IEICE TRANS. INF. & SYST., VOL.E101–D, NO.12 DECEMBER 2018

PAPER *Special Section on Parallel and Distributed Computing and Networking*

Parallel Precomputation with Input Value Prediction for Model Predictive Control Systems

Satoshi KAWAKAMI†a), Takatsugu ONO†, *Members*, Toshiyuki OHTSUKA††, *Nonmember*, *and* Koji INOUE†, *Member*

SUMMARY We propose a parallel precomputation method for real-time model predictive control. The key idea is to use predicted input values produced by model predictive control to solve an optimal control problem in advance. It is well known that control systems are not suitable for multi- or many-core processors because feedback-loop control systems are inherently based on sequential operations. However, since the proposed method does not rely on conventional thread-/data-level parallelism, it can be easily applied to such control systems without changing the algorithm in applications. A practical evaluation using three real-world model predictive control system simulation programs demonstrates drastic performance improvement without degrading control quality offered by the proposed method.

key words: *parallel precomputation, input value prediction, approximate computing, model predictive control, real-time system*

1. Introduction

Control systems are ubiquitous in real-world applications. This is because a wide range of products, such as electronic devices in houses, automobiles, aircraft, and manufacturing equipment for petrochemical plants make extensively use automatic control technologies. For instance, modern automobiles contain a number of automatic controllers that are used for fuel saving, emissions reduction, and driver assistance such as in anti-lock brake systems. Clearly, improvements in control systems can play an important role in facilitating people in performing everyday tasks.

Figure 1 shows a feedback control system consisting of a controller, plant (controlled object), sensor, and actuator. The input values of the controller (denoted as *reference* $r(t)$) represent an objective state of the plant and the sensed data (denoted as *system state* $x(t)$) indicate the current state of the plant. The output of the controller (denoted as *system input* $u(t)$) is fed to the actuator for controlling the plant. One of the main objectives of this control is to minimize the difference between $r(t)$ and $x(t)$ in real time.

MPC (model predictive control) is a control method and has attracted considerable attention in recent years in the area of control technology, since it can be effectively applied to complex nonlinear systems. The model used in MPC is

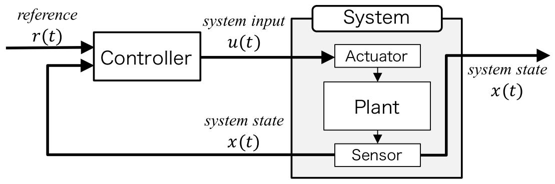


Fig. 1 Block diagram of feedback control system.

generally intended to represent the behavior of a complex dynamical system (or plant) by predicting the changes in $x(t)$. Based on this predicted behavior, MPC finds the optimum $u(t)$. This approach provides the ability to account for system dynamics by predicting system future behavior. Although MPC can potentially be used in a wide range of applications, the additional complexity of its algorithm hinders its application to satisfy the real time constraint. To predict the behavior of a system, it is required to solve optimization problems for each sampling time. The computational complexity of the optimization problem in MPC with N variables is $O(N^3)$ [1]. This is the main reason MPC has been mainly applied only in the processing industry at facilities such as chemical plants and oil refineries, which require second-order real-time operations. Since advanced nonlinear complex control systems, e.g., automatic driving systems and submarine cable laying systems, tend to require millisecond- and nanosecond-order sampling time for critical real-time control, the implementation of an ultra-high-performance MPC execution platform is a considerable challenge. Another advantage of high speed MPC execution is to provide the capability of applying a fine-grain complex optimization strategy to improve control quality in a given sampling time.

On the other hand, after the microprocessor had reached the limit of the operating frequency improvement in early 2000, multi-core processors have become mainstream and shifting to the many-core era [2], [3]. In on-chip parallel processing, the degree of parallelism significantly impacts performance improvement in general. In the control system, however, the sequential process is dominant for the entire process because there is dependency for the input value to start the process given in time series. Therefore, extracting traditional thread- and data-level parallelism from MPC applications is inherently difficult.

To address this issue, we propose a parallel precomputation method of MPC applications on many-core proces-

Manuscript received December 18, 2017.
Manuscript publicized September 18, 2018.

†The authors are with the Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka-shi, 819–0395 Japan.
††The author is with the Graduate School of Informatics, Kyoto University, Kyoto-shi, 606–8501 Japan.
a) E-mail: satoshi.kawakami@cpc.ait.kyushu-u.ac.jp

DOI: 10.1587/transinf.2018PAP0003

sors. We attempt to effectively use computational resources (cores) by taking into account the theory behind MPC. The contributions are as follows.

- We analyze MPC source codes developed for real control systems and observe unavoidable sequential operations that come from the MPC algorithm nature occupy about 70% of total execution, i.e., the maximum speed-up we can expect is only 1.4X even if an infinite number of cores is assumed. This result clearly shows that current many-core parallel execution platform cannot satisfy real-time constraint for future critical MPC applications.
- We then investigate required computation time to improve control quality at a given sampling time. An aggressive parameter tuning (increasing the number of partitions to solve MPC optimization problem explained in Sect. 2.5) for temporally fine-grained strategy is assumed. It is observed that such parameter tuning significantly improves control efficiency by achieving MPC execution speed-up.
- To bridge the performance gap, we propose a parallel precomputation method for MPC applications. The key idea behind this method is to predict $x(t)$, which will be fed to the controller as input in the future. In addition, prediction accuracy of input value can be improved by introducing a dedicated thread (IPNP) for value prediction. Such input value prediction makes it possible to apply aggressively parallel precomputation to exploit the many-core potential.
- Execution performance and control quality of proposed method are evaluated by using three actual MPC control applications. The results show that our approach can achieve almost linear scalability to the number of cores used. It is also demonstrated that such performance gain can be translated to control quality (defined in Sect. 2.3) improvement in the range from 1.2X to 6.2X.

To our knowledge, this is the first attempt at accelerating the execution of MPC by efficiently using many-core processors. The organization of this paper is as follows. In Sect. 2, the theory and implementation of MPC are explained to clarify the basic feature which is a trade-off between control quality and computation time, and the bottlenecks of sequential execution. Section 3 presents the proposed parallel precomputation method, and Sect. 4 discusses the potential for performance improvement. Section 5 presents related work, and finally, Sect. 6 concludes the paper.

2. MPC (Model Predictive Control)

2.1 Theory

MPC is a control method that determines *system inputs* based on the prediction of the future behavior of a plant [4]. Figure 2 illustrates the theory behind MPC. The $x(t_n)$, which represents the current state of the plant, is fed to the

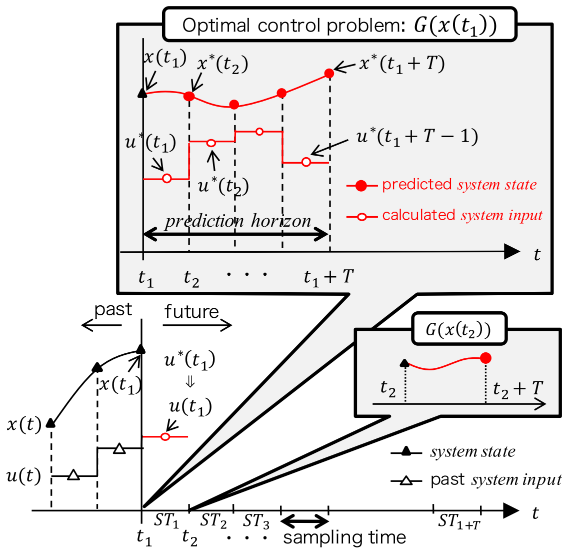


Fig. 2 Overview of model predictive control.

controller at every *sampling time* denoted as ST_n. The controller is required to calculate the associated $u(t_n)$ before the next $x(t_{n+1})$ arrives; otherwise, it fails the requirement of real-time operation. In other words, MPC has to determine $u(t_n)$ by solving the optimal control problem $G(x(t_n))$ for each sampling period. The most important feature of MPC is that an optimal control problem is solved by considering the plant's future behavior throughout a time range called the *prediction horizon* T. For instance, at time t_1 in Fig. 2, the controller receives the *system state* $x(t_1)$, and $G(x(t_1))$ process starts calculating its associated optimal *system input* $u(t_1)$. In this process, the controller attempts to find the optimal set of *system inputs* denoted as $u^*(t_1), u^*(t_2), ..u^*(t_1+T-1)$, which make the plant transit to the target state in minimum time, by predicting the value of the *system state* denoted as $x^*(t_2), x^*(t_3), ..x^*(t_1+T)$ for *prediction horizon* $[t_1, t_1+T]$. Finally, $u^*(t_1)$ is used as $u(t_1)$. At the next time step t_2, $u(t_2)$ is calculated in the same manner for $[t_2, t_2+T]$.

MPC can precisely control the behavior of nonlinear systems by accurately formulating a dynamic model of the plant. We consider a general nonlinear system to be controlled and define it as:

$$\dot{x}(t) = f(x(t), u(t), t) \qquad (1)$$

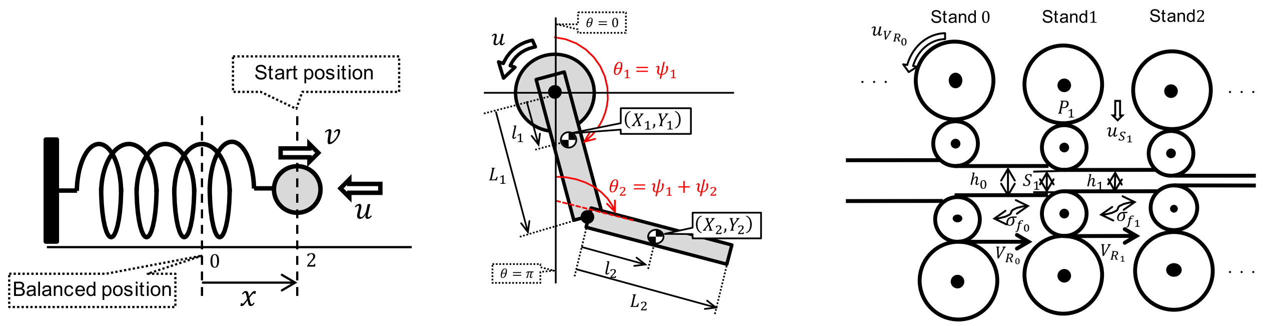
where $x(t) \in \mathbb{R}^n$ is the state vector and $u(t) \in \mathbb{R}^m$ is the vector of *system inputs*. Using Eq. (1), the optimal control problem $G(x(t))$ is formulated as follows [4]:

Minimize

$$\phi(x^*(t+T), t+T) + \int_t^{t+T} L(x^*(\tau), u^*(\tau), \tau)d\tau$$

subject to

$$\dot{x}^*(\tau) = f(x^*(\tau), u^*(\tau), \tau)$$
$$x^*(t) = x(t)$$



(a) Nonlinear spring **(b)** Arm-type pendulum **(c)** Tandem cold mill

Fig. 3 Benchmark applications which simulate MPC control systems.

$$x_{min} \le x^*(\tau) \le x_{max}$$
$$u_{min} \le u^*(\tau) \le u_{max}$$

where τ ($t \le \tau \le t + T$) is a variable and $x^*(\tau)$ and $u^*(\tau)$ are functions of that variable, $L()$ is the stage cost during T, and $\phi()$ is the termination cost. Designers carefully formulate $L()$ and $\phi()$ based on the physical behavior of the system. When the *system state* value at a certain time t is given, the objective function is minimized in the range of evaluation interval, $[t, t + T]$. The stage cost expresses penalties for the undesirable behavior of a system. Ideally, MPC should have infinite T ($T = \infty$) for the proof of stability in the whole control system. However, it is impossible to find a general solution for the optimal control problem for $T = \infty$. Instead, MPC takes $\phi()$ that attempts to approximate the cost of the interval $[t + T, \infty)$ into account.

MPC cannot mathematically prove whether a nonlinear system with constraints will stabilize or not [5]. In MPC, the proof of stability is equal to find a general solution for the optimal control problem in the case of infinite T. Although it is known that stability of unconstrained nonlinear systems can be guaranteed [6], almost existing real-world systems are suffered from some constraints. Mathematical prove of stabilization of MPC is still an open problem as other optimal control methodologies in the field of control theory, so that researchers attempt to apply MPC and tune its implementation on a one by one [7], [8].

2.2 Benchmark Applications

We have developed three real-world MPC simulation C programs which simulate nonlinear spring, arm-type pendulum, and tandem cold mill control systems in Fig. 3 (a), (b) and (c). We have chosen them to include various types of realistic MPC applications that require millisecond-order critical responses. A fast solution method of MPC, which is called C/GMRES (Continuation and Generalized Minimum RESidual method [9]), is adopted to three control systems. All C programs are automatically generated from Mathematica programs in which all control systems are described based on each state equation considering physical behavior by using AutoGen [10]. We carefully have formulated every state equation with experts of modern control theory. As for details, refer to the appendix[†] to be mentioned later in Appendix A.

Table 1 Parameters list for benchmark applications

Benchmark	Nonlinear spring	Arm-type pendulum	Tandem cold mill
Prediction horizon	1000[ms]	500[ms]	100[ms]
Sampling time	10[ms]	1[ms]	1[ms]
Simulation time	20[s]	10[s]	35[s]
Initial state	$x = 2$	$\theta_1 = \pi$ $\theta_2 = \pi$	$h_1 = 3.4[mm]$ $\sigma_{f_0} = 20[MPa]$
Target state	$x = 0$	$\theta_1 = 0$ $\theta_2 = 0$	$h_1 = 3.4[mm]$ $\sigma_{f_0} = 20[MPa]$
Control performance	$\frac{1}{settling\ time}$	$\frac{1}{settling\ time}$	$\frac{1}{variance}$

Table 1 shows key parameters list for these benchmark applications. The nonlinear spring control system is intended to stop at a position of equilibrium by applying an external force to a free vibration spring with a weight. The initial static state is a positive position ($x = 2$), and the target state is to stop on the balanced position ($x = 0$). The arm-type pendulum control system tries to make two arms stand upright. The inner arm is only controlled by a DC motor. In the initial state, both arms are hanging down ($\theta_1 = \theta_2 = \pi$), and the MPC tries to make them the target state ($\theta_1 = \theta_2 = 0$). The tandem cold mill is a metal forming process in which metal is passed through several pairs of rolls to reduce thickness and to make the thickness uniform at low temperature. This simulation program focuses on one stand, though a tandem cold mill control system generally has several stands. The purpose of this program is to keep the thickness and tension constant. Hence, the initial state is same as the target state ($h_1 = 3.4$, $\sigma_{f_0} = 20$).

2.3 Performances Definition

In a control system, the final goal is to stabilize the desired state. This means control performance is one of the most important factor as an evaluation index. The other important factor is computation time, which is influenced by the control method, its algorithm, and hardware specification of the

[†]Since they are not open source benchmark applications, we disclose the state equations to ensure reproducibility of the experiment.

execution platform. We define **control performance** and **computation performance** as evaluation indexes.

There are two control performances we use depending on benchmark applications. One is defined as $1/\textit{settling time}$, where the *settling time* means the time required for the response curve to reach and stay within a range of ±5%. Figure 4 (a) shows that *system state* converging near the reference value at the *settling time* and never over ±5% after that. In nonlinear spring and arm-type pendulum systems, since initial state and target state are different as shown in the Table 1, $1/\textit{settling time}$ is used for control performance. The other is defined as $1/\textit{variance}$, where the *variance* is defined as:

$$\textit{variance} = \frac{\sum_{i=1}^{N}(x_{ref} - x_i)^2}{N}$$

where x_{ref} is the reference value, x_i is the *system state* at sampling step i, N is the total number of sampling-time windows, and *variance* indicates the mean square difference between reference and *system state* in Fig. 4 (b). In tandem cold mill system, the purpose of control is to keep initial state. Hence, $1/\textit{variance}$ is used for control performance. Computation performance is defined as $1/\textit{computation time}$ in all applications, where the computation time means the time from getting *system state* $x(t)$ to finishing a calculation of optimal control problem and then outputting associated *system input* $u(t)$.

2.4 Limitation of Traditional Parallel Execution

We now discuss the difficulty in parallelizing numerical calculations in MPC by using traditional data- and thread-level parallelism. In the optimal control problem, the predicted *system input* $u^*(t)$ is discretized by a suitable partitioning (hereinafter called **partition number**) as follows:

$$t = \tau_0 < \tau_1 < \ldots < \tau_n = t + T$$

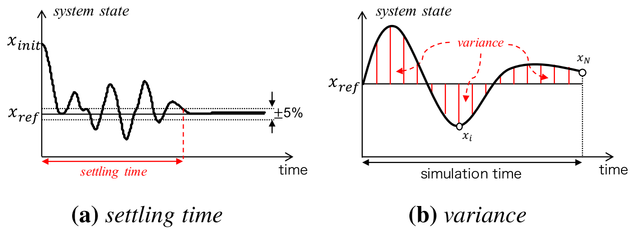
The T is divided into n subintervals $[\tau_i, \tau_{i+1}]$, $0 \le i \le n-1$. On each subinterval, it is required to compute the trajectories $x_i(t)$ by solving the following ordinary differential equation (ODE):

$$\dot{x}_i(t) = f(x_i(t), u_i(t), t)$$
$$x_{i+1}(t) = x_i(t) + \dot{x}_i(t)(\tau_{i+1} - \tau_i)$$

Note that every ODE calculates $x_{i+1}(t)$ by using $x_i(t)$ (and $\dot{x}_i(t)$) as the initial values. Therefore, each subinterval has a dependency relationship with the previous one. Since the numerical calculation in MPC is in the form of sequential, it is fundamentally hard to parallelized. With the performance evaluation of benchmark applications given in Sect. 2.2, the computation time of this sequential parts accounts for about 70% of the entire computation time on average. This means the maximum speedup we can expect is only 1.4X ($\approx 1/0.7$) even if an infinite number of cores is assumed. For instance, 4X computation performance improvement is simply required on a real-time control system that has a four-times higher sampling rate of system state $x(t)$, and the potential of traditional parallel executuions does not reach to the requirement. This is the main reason many-core processors cannot be easily applied to such control systems.

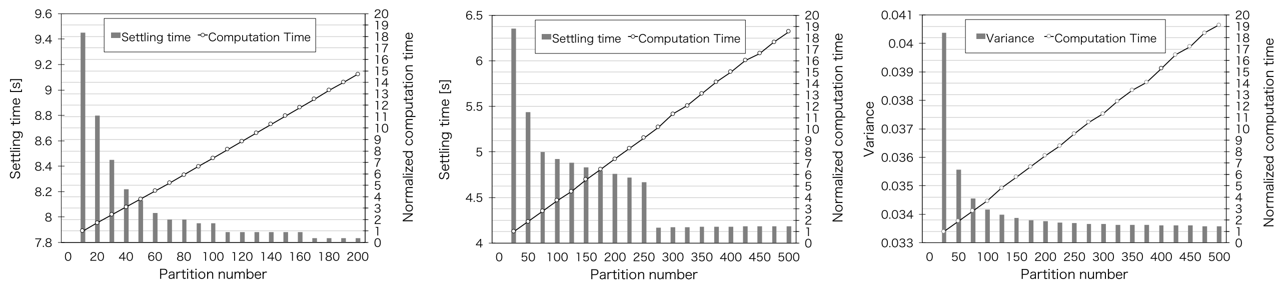
2.5 Impact of Improving Computation Performance

High-speed MPC executions contribute to improving the control performance even at a given sampling rate that can be satisfied real-timeliness in an MPC implementation. Enlarging the partition number that discretizes the *prediction horizon* T makes it possible to apply a temporally fine-grained optimization for system control. Figures 5 (a), (b), and (c) show control performance sensitivity to the partition number for three benchmark applications described in Sect. 2.2. The left y-axis in each graph shows *settling time* or *variance* (depends on the application), i.e., the lower values, the higher control performance. It is observed that the control performance can be improved by means of increasing the partition number for all cases. In the nonlinear



(a) *settling time* **(b)** *variance*

Fig. 4 Control performance



(a) Nonlinear spring **(b)** Arm-type pendulum **(c)** Tandem cold mill

Fig. 5 Trade-off control performance and computation performance.

spring and tandem cold mill control, the *settling time* and *variance* are continuously reduced, roughly 20% reduction by increasing the partition number from 10 to 170 in the nonlinear spring and from 25 to 400 in the tandem cold mill control. In the arm-type pendulum control, the *settling time* drastically decreases when the partition number reaches to 75 and 275, about 20% and 35% reduction, respectively. Such stepwise improvement comes from the behavior of free-motion and motor-controlled arms. To reach the objective state, both arms have to be stabilized at an appropriate angle and timing. Once the system control misses an opportunity to stabilize the two arms, it takes a long time to find the next suitable situation. Such behavior tends to produce a kind of sweet spots in terms of the partition number.

Although increasing the number of partitions provides better control performance, such improvements require the significant cost in terms of computation time. The right y-axis in each graph in Figs. 5 reports computation time required. All results are normalized to the case for processing the smallest number of partitions. We see from the graphs that the computation time linearly increases in proportion to the partition number in all applications. To achieve the 20% or 35% of the *settling time* or *variance* reduction as mentioned above in three benchmark applications, from the range of 3X to 15X computation cost is required. It is clear that the 1.4X small performance improvement achieved by the traditional thread- and data-level parallel acceleration cannot provide enough capacity to cover such computation performance requirement.

3. Parallel Precomputation with Input Value Prediction for MPC

3.1 Concept

In on-chip parallel processing, the degree of parallelism generally has a strong impact on the potential for computation performance improvement. In MPC systems, however, sequential operations dominate the entire process, as explained in Sect. 2.4. Meanwhile, to control advanced systems with high efficiency, the demands of MPC applications are becoming more severe. To deal with the issue and expand the applicability of many-core processors to control systems, we propose a many-core execution method that allows for accelerating MPC executions. MPC systems have the following two features.

1. Particular processes are executed in time series **independently**. This means that each process can start when the associated input values for the controller (the *system state* values from corresponding sensors in the system) become available.
2. In each process, MPC **accurately predicts the behavior of the target plant** (the *system state* values from the system) then finds the optimum strategy to control the actuators.

These two points are essential to the construction of the proposed method for accelerating MPC execution. As explained above, MPC inherently has the ability to predict input values for execution that will be fed into the controller in the future. The main idea behind our approach is to use the predicted input values not only for control optimization but also for process-level precomputation. Our approach has two advantages. (1) When a real-time restriction is severe on a certain execution environment, our method can achieve high performance processing by using more cores. (2) Even though a real-time restriction has been satisfied, control performance can be improved by increasing the partition number and compensating for an increase in computation time by our method.

3.2 Parallel Precomputation

The execution of each process in real-time MPC has to be completed before the next input arrives, as shown in Fig. 6 (a), where $x(t_n)$ is the input at time t_n for the controller, and ST_n and $u(t_n)$ are the sampling-time window and output to the actuator associated with $x(t_n)$, respectively. As explained in Sect. 2.1, $G(x(t_n))$ is solved in each ST_n. Figure 6 (b) shows the case with the conventional sequential execution method. In this scenario, we assume that the execution in each ST_n takes almost double the maximum amount of time allowed. Although it is possible to apply conventional thread-level parallel execution, we cannot expect a significant computation performance improvement due to the lack of parallelism. Since there exist the 70% sequential ODE calculations in MPC execution as explained in Sect. 2.4, a 4-core platform illustrated in Fig. 6 (c) can yield up to 1.4X ($\approx 1/0.7$) increase in speed that is insufficient to

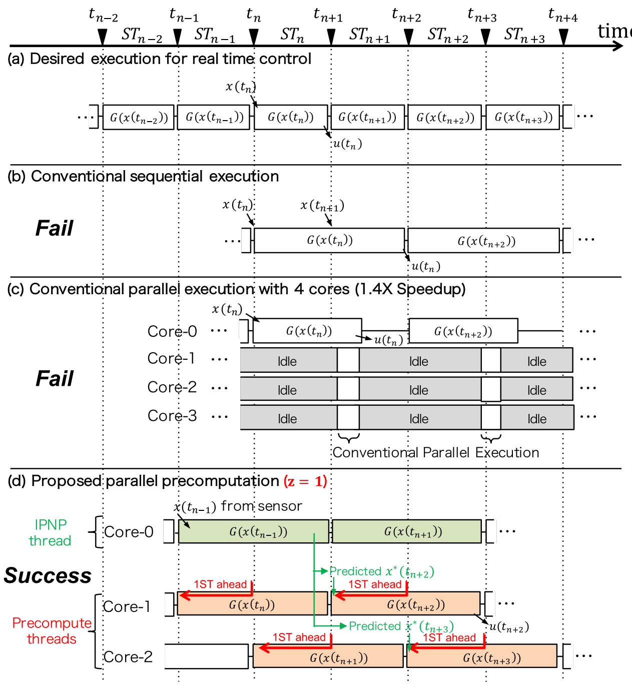


Fig. 6 Overview of parallel precomputation.

achieve real-time operation.

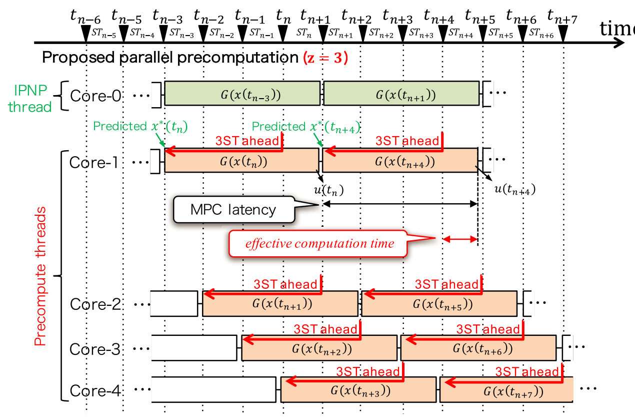
Figure 6 (d) presents a conceptual representation of the proposed method, assuming a dual-core processor for simplicity. We represent the features of our method by parameter z, which is the degree of precomputation in terms of the number of sampling-time windows. This is an indicator of how long in advance each process can start before the corresponding actual input arrives. Since $z = 1$ in Fig. 6 (d), $G(x(t_n))$ of precompute threads starts at t_{n-1} on core-1. Although the actual input value for ST_n (it means $x(t_n)$) is not available at that time, we use a predicted input value ($x^*(t_n)$), which has been computed in a previous sampling-time window. Core-2 similarly starts $G(x(t_{n+1}))$ at time t_n by using the predicted input value $x^*(t_{n+1})$.

A straightforward implementation of the proposed precomputation is to keep using the predicted input values for system control, i.e., an optimal control problem is solved by using previously predicted input values and its execution results are fed to consecutive precomputations as newly predicted ones. Since the dynamic behavior model constructed in MPC is quite accurate but not a perfect prediction, such *prediction chain* causes an accumulative error problem, finally failing to control the system. To overcome this issue, we introduce a dedicated thread to break the prediction chain, called input-predicting non-precompute (IPNP) thread, that uses not the predicted inputs but the real sensor values obtained from the target system. The IPNP solves the optimal control problem and just update predicted values, i.e., the output of IPNP $u(t)$ is not transferred to the system actuator. Precompute threads that use predicted input values are prohibited to update predicted values. IPNP and precompute threads are executed in parallel. For instance, in Fig. 6 (d), IPNP thread is executed based on an actual sensor value $x(t_{n-1})$ at time t_{n-1} and produce predicted input values $x^*(t_{n+2})$ and $x^*(t_{n+3})$, which are used in precompute threads at time t_{n+1} ($G(x(t_{n+2}))$) and t_{n+2} ($G(x(t_{n+3}))$).

3.3 Computation Performance Improvement

With our method, each process is executed on a single core, so that its computation time is the same as MPC latency (the time period spent for the execution of $G(x(t))$) with the conventional sequential method. However, the *effective computation time* (the time period from getting $x(t)$ until finishing the execution of $G(x(t))$) is reduced almost half if we set the precomputation parameter $z = 1$, resulting in a two-fold speedup over the conventional single-core execution method. This improvement of computation performance makes it possible to satisfy the real-time operation constraints, namely that $u(t)$ is generated before the next input arrives.

In case of a more performance-critical MPC application, aggressive precomputation can be possible. Potentially, the proposed method can start a precomputation when the first prediction of associated input values is carried out. For instance, in Fig. 2, $x^*(t_2), x^*(t_3), \ldots, x^*(t_1 + T)$ are predicted by solving $G(x(t_1))$. This means that even $G(x(t_{1+T}))$



t_{n-6} t_{n-5} t_{n-3} t_{n-2} t_{n-1} t_n t_{n+1} t_{n+2} t_{n+3} t_{n+4} t_{n+5} t_{n+6} t_{n+7}
ST_{n-5} ST_{n-4} ST_{n-3} ST_{n-2} ST_{n-1} ST_n ST_{n+1} ST_{n+2} ST_{n+3} ST_{n+4} ST_{n+5} ST_{n+6}
time
Proposed parallel precomputation (z = 3)
IPNP thread
Core-0
$G(x(t_{n-3}))$
$G(x(t_{n+1}))$
Predicted $x^*(t_n)$
Predicted $x^*(t_{n+4})$
3ST ahead
Core-1
$G(x(t_n))$
$G(x(t_{n+4}))$
$u(t_n)$
$u(t_{n+4})$
MPC latency
effective computation time
Precompute threads
Core-2
$G(x(t_{n+1}))$
$G(x(t_{n+5}))$
Core-3
$G(x(t_{n+2}))$
$G(x(t_{n+6}))$
Core-4
$G(x(t_{n+3}))$
$G(x(t_{n+7}))$

Fig. 7 The effective computation time with aggressive parallel precomputation ($z = 3$).

can be executed speculatively at any time after the generation of $x^*(t_2), x^*(t_3), \ldots, x^*(t_1 + T)$ in ST_1. This feature allows us to increase z. Figure 7 gives an overview of aggressive precomputation as an example ($z = 3$), where we assume a more performance-critical MPC application than that in Fig. 6 (d). Since the required sampling-time window is a fourth that for the system considered in Fig. 6 (a), even the two-fold increase in speed achieved with the proposed execution method (Fig. 6 (d)) fails the requirement of real-time operation. By using four cores and starting the execution of each process three sampling-time windows in advance, as shown in Fig. 7, we can achieve a four-fold increase in speed; thus, satisfying the real-time operation requirements. Note that $z + 1$ cores are needed for precomputation like pipeline execution since actual computation time does not change as stated above and one more core is needed for IPNP thread. In conventional execution method, computation time is "MPC latency", meanwhile the *effective computation time* is "MPC latency $-$ ($z * ST$)". The *effective computation time* can be shortened in a quarter in Fig. 7. We can denote the improvement of computation performance as $Speedup = z + 1$. Theoretically, regardless of the prediction accuracy, we can achieve perfect scalability, which is one of the most important metrics of many-core systems.

3.4 Implementation

We have implemented the proposed execution framework presented in Fig. 6 (d) by using *Pthreads*. Figure 8 shows the data flow in our implementation. The IPNP thread introduced in Sect. 3.2 operates as follows.

IT.1: Obtain the current *system state* x from the sensors.

IT.2: Solve the MPC optimal problem with the real inputs.

IT.3: Update the *predicted value table* by overwriting with the predicted *system states* x^* calculated in IT.2.

The *predicted value table* has only one predicted values $x^*(t)$ for each t and is implemented in software. Although the table ideally holds predicted values at all sampling time steps,

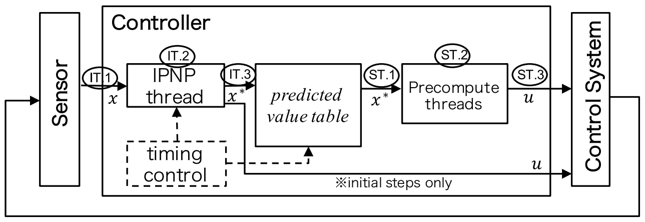


Fig. 8 Data flow in proposed method with IPNP thread

size of the table will be infinite in systems that need to be controlled endlessly. To reduce the table size, we have to consider (1) how far the predicted values are required in advance and (2) how long the values need to be kept. Regarding (1), IPNP thread produces $z+1$ predicted values for precompute threads to be executed near future, e.g., the IPNP thread $G(x(t_{n-1}))$ predicts two values ($x^*(t_{n+2})$ and $x^*(t_{n+3})$) in Fig. 6 (d). The answer for (2) is until finishing the each process with associated input values. For example, during the IPNP thread $G(x(t_{n-1}))$ calculates, precompute threads ($G(x(t_n))$ and $G(x(t_{n+1}))$) are not completed. The table have to keep $z + 1$ predicted values ($x^*(t_n)$ and $x^*(t_{n+1})$). Hence, the total number of memory entry required to implement the predicted value table can be expressed as $2 \times (z + 1)$. For example, if a system requires $z = 10$ precomputation to satisfy real-timeliness and two 8-byte variables are used to represent the system status, the total capacity of the table is $2 \times (10 + 1) \times 16 = 352$ bytes, which is negligibly small. Predicted values are written into the table with the index of $t \bmod 2\times(z+1)$ at sampling time t by the IPNP thread. Since precompute threads use the most recently predicted values, every IPNP thread greedily writes predicted values into the table, i.e., no synchronization is required between the IPNP and precompute threads.

The precompute threads that generates signals to control the actuator works as follows.

ST.1 Obtain x^*s from the *predicted value table*.
ST.2 Speculatively solve the MPC optimization problem with the predicted input values.
ST.3 Output u to the actuator.

Note that during initial steps until the $z + 1$ step, conventional sequential threads were executed instead of precompute threads to solve a cold start problem of *predicted value table*.

4. Performance Evaluation

4.1 Methodology

One of the main discussion points for parallel precomputation is the trade-off between computation performance and control performance. Regardless of the prediction accuracy, computation time becomes shorter in proportion to the precomputation parameter z (or the number of cores) and control performance can be improved by changing the partition number as shown in Sect. 2.5. If the input prediction is inaccurate, as the computation time is shortened, the quality of actuator control may be deteriorated, so the *settling time* and *variance* increase. Therefore, the parameter z is an important knob to adjust this trade-off.

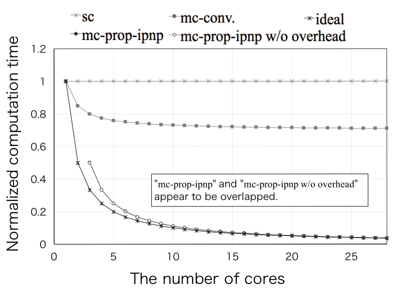
Firstly, we clarify the improvement of computation performance when given a certain number of cores. Although computation time can be shortened according to z, additional computation time (overhead) is needed for the extra procedures such as read from *predicted value table* in the proposed method. Hence, we evaluate the computation time with the overhead. Secondly, we evaluate how much the prediction accuracy deteriorates according to the number of cores by measuring the absolute values of the difference between sensor values and predicted values. Finally, we demonstrate the control performance efficiency of our parallel precomputation framework by assuming a given sampling time.

To clarify the accuracy sensitivity to parameter z and the overhead of computation time, we conduct computation and control performance analysis by implementing the proposed method on three MPC benchmark applications (Sect. 2.2) by using Pthreads. The computational environment is Intel Xeon Phi with 8-GB GDDR memory, and we use a native execution model that directly executes applications on an Intel Xeon Phi co-processor without a Linux Host computer. Intel Xeon Phi is unsuitable considering actual embedded processors because it has many cores and even one core of Xeon Phi exhibits higher performance (frequency is nearly 1 GHz) than embedded processors such as ARM, for real-time applications (frequency is nearly 200 MHz). The main point of this section, however, is to clarify the effectiveness of our method. Since the benchmarks are simulation programs, the execution platform has no effect on control performance. Assuming an embedded many-core processor, the cache size is smaller than Phi, and it is not equipped with high-performance NoC. However, since the MPC code size and working set size are small, almost all memory accesses hit the cache even in embedded processors. Also, there is very little inter-thread communication (predicted value table access only), so the difference in NoC performance does not affect. Therefore, even assuming an embedded processor, I think that the same tendency as Phi's result can be observed in this experiment.

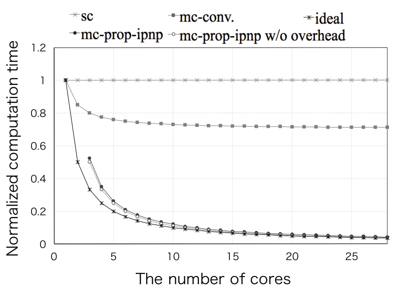
4.2 Experimental Result

Computation Performance Improvement

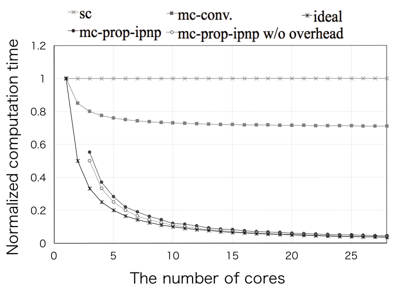
Figure 9 (a), (b) and (c) show computation time sensitivity to the number of cores in three benchmark applications respectively. The y-axis shows whole computation time of MPC calculation normalized by single-core execution (denoted by sc). The mc-conv means multi-core execution using conventional data-/thread-level parallelism which assumes Amdahl's law speedup with 70% sequential processing accounts as described in Sect. 2.4. The ideal indicates the perfect scalability where normalized computation



(a) Nonlinear spring

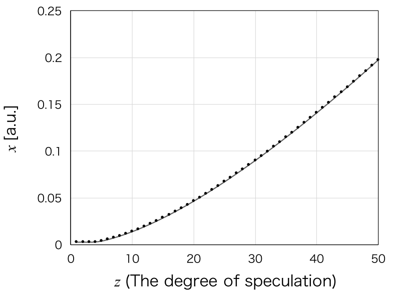


(b) Arm-type pendulum

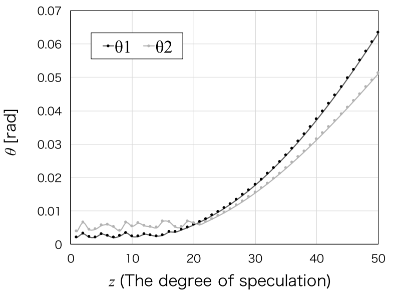


(c) Tandem cold mill

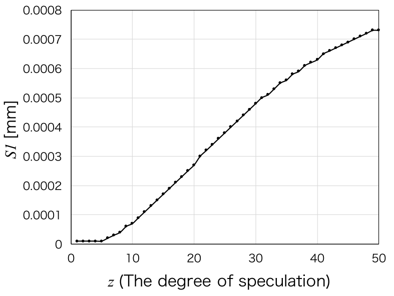
Fig. 9 Computation time sensitivity to the number of cores.



(a) Nonlinear spring



(b) Arm-type pendulum



(c) Tandem cold mill

Fig. 10 Absolute values of the difference between sensor values and predicted values.

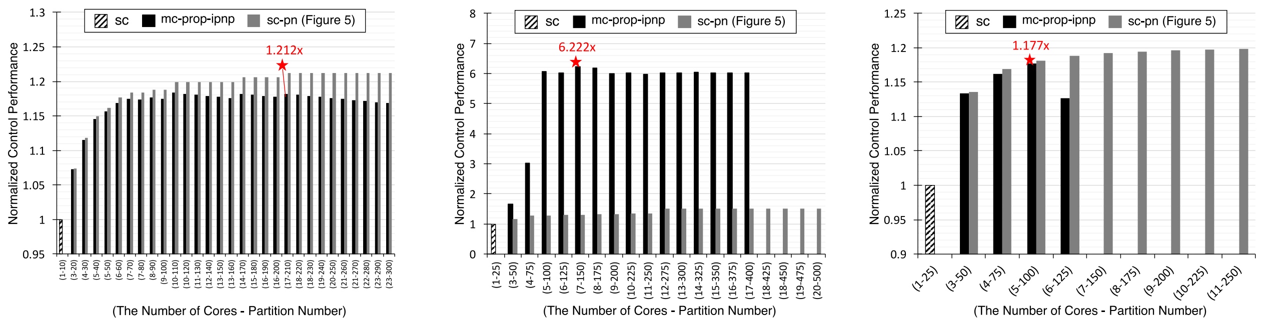
time can be reduced $1/N$ (N is the number of cores) and implies the upper limit of the direct benefit by parallel processing. The mc-prop-ipnp is computation time including overhead with our proposed method. Our method can need at least three cores because one core is used for IPNP thread and minimum $z = 1$ requires two cores. The parameter z can be increased according to the number of cores, and it presupposes that $z * sampling\ time$ does not exceed the MPC latency. Although our method with a such z can be executed theoretically, the computation time is 0 (computation performance is ∞). It is clear that overhead is negligible for every application. Hence, when a certain MPC application is given, if we can use enough number of cores our method achieve desired computation performance improvement. In other words, mc-prop-ipnp can almost achieve perfect scalability (*normalized computation time* $= 1/(N-1)$) at a large number of cores. This method, unlike previous related work, is adopted to any MPC applications without changing algorithm, and not only part of MPC execution time but whole computation time can be reduced like an ideal improvement of parallel processing. Even when shorter sampling-time is required, real-time restriction can be satisfied by the proposed method.

Control Performance Improvement

To clarify the input value prediction accuracy based on MPC algorithm, we got the log information of sensor values and predicted values at every z. Figure 10 (a), (b) and (c) show the absolute values of the difference between sensor values $x(t_n)$ and predicted values $x^*(t_n)$ for all applications. Each plot point indicates the maximum difference values (i.e., $max(|x(t_n) - x^*(t_n)|)$, where $1 \leq n \leq N$, N is the total number of sampling-time windows.) at a certain z. Although Fig. 10 only show the part of state value in each application, it is clear that every difference value increases together with z. This trend is also seen in other state values of every application. For example, the difference values are relatively small in the small number of cores, and then suddenly increase exponentially in Fig. 10 (b). Since it depends on each system that how much difference can be tolerated, we comprehensively evaluate the proposed method by measuring control performance.

Figures 11 (a), (b) and (c) show normalized control performance associated with the combination of the number of cores and the partition number (concatenated by the hyphen). For instance, "(4-30)" denotes that quad cores are used with the partition number of 30. All results are normalized to the control performance achieved by the single-core execution *sc* which is used as a baseline in this evaluation. The evaluation results reported in Fig. 5 is also plotted in this figure as *sc-pn*. The number of cores in the x-axis combination is determined to compensating for an increase in execution time. For example, when the partition number is 100, it takes 7.5X computation time as showed in Fig. 5 (a). In such case, 8X computation performance improvement ($z = 7$) is required, so that 9 cores needed. That is why the combination in Fig. 11 appears irregular.

As shown in Fig. 11, the appropriate configuration of the number of core and partition number (mp-prop-ipnp) can provide about a maximum of 1.2x, 6.2x, and 1.2x improvement in control performance for each benchmark



(a) Nonlinear spring **(b)** Arm-type pendulum **(c)** Tandem cold mill

Fig. 11 Control performance improvement by using the proposed method and changing partition number.

program, respectively. This means that higher control performance can be achieved by increasing the number of cores. This improvement can not be achieved by the conventional parallelized, since computation performance improvement is limited up to 1.4x as shown in Fig. 9 and partition number cannot be increased. The results indicate that our method provides a new opportunity to improve control performance by means of increasing partition number when real-time restriction is severe in a certain execution environment. In large number of cores, e.g., (18-425) in Figs. 11 (b), the system state vibrates or diverges denoted by no-bar of mp-prop-ipnp; as a result, the system cannot be stable within the simulation time. It is because the effect of inaccurate prediction appears according to increasing parameter z.

Figures 11 (a) (c) show that our method have a negative effect and it gradually has a large impact on control performance on a large number of cores. The difference between mc-prop-ipnp and sc-pn means control performance degraded due to the proposed method. In tandem cold mill, especially the system becomes unstable at (7-150). Since the benefit of increasing the partition number outweigh the negative effects, these control performance is improved in the range of a small number of cores.

As for the arm-type pendulum control system, Fig. 11 (b) shows higher control performance of mc-prop-ipnp than sc-pn. Although predicting input values make control performance worse intuitively, this is not always true. There are cases that exploiting predicted input values accelerates control performance. The breakdown of 6.222x control performance improvement at (7-150) in Fig. 11 (b) is: 1.3x ($\approx$ 6.4/4.8) by increasing the number of partitions and 4.8x (1.3x$*$4.8x$\approx$ 6.2x) by using the predicted input values instead of the real inputs. MPC attempts to globally optimize the plant control by locally optimizing an objective function in each finite interval (*prediction horizon*). There is a possibility in such optimization scheme that "error" on input values improve or degrade the control performance, and this is a theoretical nature of MPC. That is why predicted input values possibly contribute to performance improvement. This can be seen in Fig. 11 (b). In our scheme, the input value prediction causes the "error". The "error" affects in an enhancement of the control performance from (5-100) to (17-400), meanwhile, it produces the opposite results over (18-425).

The proposed method has low responsiveness because it does not adjust to real behavior after an optimal control problem starts. Regardless of the purpose and noise, the proposed method cannot adapt to unpredicted behavior. For example, the proposed method could not stabilize the tandem cold mill control system even at 2x speedup when the input rotation velocity V_{r_0} changes irregularly. However, this also implies the possibility of the proposed method adapting to a system that includes a disturbance if MPC includes disturbance behavior model. Solving this problem and robustness improvement are future works.

5. Related Work

There have been a number of studies on the application of MPC to real-world systems (i.e., gasoline engines and processors for thermal and energy management) [8], [11]. Since MPC can be applied in nonlinear complex systems, it has the potential to become a major control methodology in future high-performance embedded systems. In this section, we categorize acceleration techniques of MPC executions and clarify the advantages of our precomputation method over other methods.

MPC Parallelization: Although the key concept of parallel precomputation method for MPC has considered in [12], it mainly lacks four critical points. First, the related paper does not discuss the impact of real implementation. The authors developed performance models and theoretically (not physically on a real machine) estimated computation performance. Second, since the used performance model does not directly consist of the accuracy of input value prediction, real impacts of prediction accuracy is not clear (the authors indirectly analyzed the impact of prediction accuracy). Third, the discussed precomputation method cannot solve the prediction-chain problem explained in Sect. 3.2, the control performance is significantly degraded. Fourth, only computation performance is discussed, i.e. no discussion for control performance that is critical for control system. Against to the previous research, our paper covers all of the critical points. We have introduced the IPNP thread to break the prediction-chain, and have implemented the precomputation framework on a real machine. Our evaluation results demonstrate that the IPNP thread based precomputation scheme can improve both computation and control performance on a physical implementation.

A few related studies proposed parallelized MPC implementations [13], [14]. Longo et al. proposed a parallel move-blocking MPC algorithm, in which multiple small optimization problems are executed in parallel [13]. Soudbakhsh et al. used a sparse format of derived matrices [14]. Unlike such methods, our method does not require any algorithm-level modification, just executing speculatively with predicted input values. Combining the straightforward parallel implementation with our parallel precomputation method may have significantly improved MPC performance, and evaluating such hybrid implementation is for future work.

Algorithm optimization: Parallel-in-Time is well known as a parallelization method of time-dependent processing [15]. This method is applied to a wide range of systems such as ordinary differential equations (e.g., molecular dynamics) [16], [17] and partial differential equations (e.g., hydrodynamics) [18], [19]. In addition, the method is often applied in the field of high-performance computing which have large size simulation problems [20]–[23]. However, problems of this method have been pointed out in the past and the positive effect can not always be obtained. Parallel-in-Time requires to define an approximate calculation instead of the original processing. Since it is very difficult to define approximate calculations in a general form, the scope of application is limited. Furthermore, since the cost of approximate calculation and the cost of this iterative calculation are required, this method is effective only when the speedup can be sufficiently increased against to the original processing. Namely, it is difficult to achieve high scalability. On the other hand, our method can be applied to all kind of system if it is controlled by MPC. In addition, our method can achieve almost perfect scalability as shown in Fig. 9.

Hardware Acceleration: To improve MPC execution performance, hardware-based acceleration methods have been discussed. Dimitriou et al. proposed an application-specific processor targeting MPC executions, in which a matrix coprocessor is implemented [24]. Wills et al. proposed a custom architecture and demonstrated the implementation as an application specific integrated circuit (ASIC) [25]. Using field-programmable gate array (FPGA) devices is another alternative for hardware implementation [26]–[28]. Such hardware-based accelerations can significantly improve performance by sacrificing flexibility. Since the dynamic model embedded in a controller depends on the target plant, MPC execution platforms should support programmability. Unfortunately, ASIC implementation cannot satisfy such a requirement. Although FPGAs have much better flexibility than ASICs, the design cost is still much higher than a pure software approach.

Speculative execution: Thread-level speculation (TLS) has been extensively investigated in terms of software solutions [29]. Speculative threads can be defined or identified using software [30]–[35] or hardware [36]–[38]. To break the data dependency chain, value prediction techniques have also been extensively investigated [39]–[43]. Researchers have also proposed architectural supports for efficient TLS executions [44]–[48]. Against traditional TLS executions, our method differs on two points. First, MPC involves repeatedly executing the optimization problem when new input data arrive, and an instance of the optimization problem is assigned to a thread. Thus, we do not need to consider control dependency. Second, our method attempts to solve data dependency by exploiting the predictability of the input value that inherently exists in the MPC theory. Traditional TLS methods predict memory or register values then speculatively execute a part of the sequential code. Unlike traditional TLS methods, our method is quite simple.

Run-ahead execution: Run-ahead execution is a type of precomputation method [49]–[54]. By pre-computing a kernel code, we can prepare for efficient main thread executions, e.g., training branch predictor and prefetching data from off-chip main-memory to on-chip caches. Even though the concept of our method is similar to that of the run-ahead method, our purpose was not preparing for the main thread executions but accurately outputting the precomputed results for system control.

Approximate computing: Approximation is one of the most promising approaches to achieve energy-efficient real-world computing and many researchers have recently demonstrated that the concept of approximation performs well for vision computing [55]–[58]. They defined the criteria to quantitatively represent the quality of computation for each target application and attempted to reduce a number of operations. Our method allows a reduction in precision, i.e., accepting not using perfectly predicted input values.

6. Conclusions and Future Work

We have proposed a high-performance many-core execution method for accelerating MPC computation. The key idea is to use predicted *system state*, which is produced with MPC and precompute an optimal control problem. The results of experiments using real-world MPC applications clearly indicate the outstanding execution performance without lack of control performance for a few benchmarks. In addition, our method contributes to improving control performance by changing the processing load of MPC and compensating for an increase in execution time.

Future work is to establish a more effective method of predicting input candidates. This method will be helpful for not only more aggressive precomputation but also improving the robustness of the system including noise. Also, since power consumption is a first-order function of the design constraints in modern embedded systems, we will also consider a low-power technique for real-time MPC applications.

References

[1] Y. Wang and S. Boyd, “Fast model predictive control using online optimization,” IEEE Trans. Control Syst. Technol., vol.18, no.2, pp.267–278, March 2010.

[2] H. Xu, J. Tanabe, H. Usui, S. Hosoda, T. Sano, K. Yamamoto, T. Kodaka, N. Nonogaki, N. Ozaki, and T. Miyamori, “A low power many-core soc with two 32-core clusters connected by tree based noc for multimedia applications,” Proc. 2012 Symposium on VLSI Circuits, pp.150–151, 2012.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, “Tile64 - processor: A 64-core soc with mesh interconnect,” Proc. IEEE International Solid-State Circuits Conference, Digest of Technical Papers, pp.88–598, 2008.

[4] J.B. Rawlings and D.Q. Mayne, Model predictive control: theory and design, Nob Hill Publishing, 2009.

[5] J.M. Maciejowski, Predictive control: with constraints, Pearson education, 2002.

[6] A. Jadbabaie, J. Yu, and J. Hauser, “Unconstrained receding-horizon control of nonlinear systems,” IEEE Trans. Autom. Control, vol.46, no.5, pp.776–783, 2001.

[7] H. Seguchi and T. Ohtsuka, “Nonlinear receding horizon control of an underactuated hovercraft,” International journal of robust and nonlinear control, vol.13, no.3-4, pp.381–398, 2003.

[8] H.J. Ferreau, G. Lorini, and M. Diehl, “Fast nonlinear model predictive control of gasoline engines,” Proc. 2006 IEEE International Conference on Control Applications, pp.2754–2759, 2006.

[9] T. Ohtsuka, “A continuation/gmres method for fast computation of nonlinear receding horizon control,” Automatica, vol.40, no.4, pp.563–574, 2004.

[10] T. Ohtsuka and A. Kodama, “Automatic code generation system for nonlinear receding horizon control,” Transactions of the Society of Instrument and Control Engineers, vol.38, no.7, pp.617–623, 2002.

[11] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, “Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller,” IEEE Trans. Parallel Distrib. Syst., vol.24, no.1, pp.170–183, Jan. 2013.

[12] S. Kawakami, A. Iwanaga, and K. Inoue, “Many-core acceleration for model predictive control systems,” Proc. First International Workshop on Many-core Embedded Systems, MES ’13, New York, NY, USA, pp.17–24, ACM, 2013.

[13] S. Longo, E.C. Kerrigan, K.V. Ling, and G.A. Constantinides, “Parallel move blocking model predictive control,” Proc. 50th IEEE Conference on Decision and Control and European Control Conference, pp.1239–1244, 2011.

[14] D. Soudbakhsh and A.M. Annaswamy, “Parallelized model predictive control,” Proc. 2013 American Control Conference, pp.1715–1720, 2013.

[15] J.L. Lions, Y. Maday, and G. Turinici, “A “parareal” in time discretization of pde’s,” Comptes Rendus de l’Académie des Sciences - Series I - Mathematics, vol.332, no.7, pp.661–668, 2001.

[16] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah, “Parallel-in-time molecular-dynamics simulations,” Physical review E, Statistical, nonlinear, and soft matter physics, vol.66, p.057701, 12 2002.

[17] Y. Maday and G. Turinici, “A parallel in time approach for quantum control: the parareal algorithm,” Proc. 41st IEEE Conference on Decision and Control, vol.1, pp.62–66, Dec. 2002.

[18] D. Ruprecht and R. Krause, “Explicit parallel-in-time integration of a linear acoustic-advection system,” Computers & Fluids, vol.59, pp.72–83, 2012.

[19] D. Samaddar, D.E. Newman, and R. Sánchez, “Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm,” Journal of Computational Physics, vol.229, no.18, pp.6558–6573, 2010.

[20] E. Aubanel, “Scheduling of tasks in the parareal algorithm,” Parallel Computing, vol.37, no.3, pp.172–182, 2011.

[21] W.R. Elwasif, S.S. Foley, D.E. Bernholdt, L.A. Berry, D. Samaddar, D.E. Newman, and R. Sanchez, “A dependency-driven formulation of parareal: Parallel-in-time solution of pdes as a many-task application,” Proc. 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers, MTAGS ’11, New York, NY, USA, pp.15–24, ACM, 2011.

[22] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon, “A massively space-time parallel n-body solver,” Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’12, Los Alamitos, CA, USA, pp.92:1–92:11, IEEE Computer Society Press, 2012.

[23] R. Speck, D. Ruprecht, M. Emmett, M. Bolten, and R. Krause, “A space-time parallel solver for the three-dimensional heat equation,” Parallel Computing: Accelerating Computational Science and Engineering (CSE), vol.25, pp.263–272, 2014.

[24] P.D. Vouzis, L.G. Bleris, M.G. Arnold, and M.V. Kothare, “A system-on-a-chip implementation for embedded real-time model predictive control,” IEEE Trans. Control Syst. Technol., vol.17, no.5, pp.1006–1017, Sept. 2009.

[25] A.G. Wills, G. Knagge, and B. Ninness, “Fast linear model predictive control via custom integrated circuit architecture,” IEEE Trans. Control Syst. Technol., vol.20, no.1, pp.59–71, Jan. 2012.

[26] K.V. Ling, S.P. Yue, and J.M. Maciejowski, “A fpga implementation of model predictive control,” Proc. 2006 American Control Conference, pp.1930–1935, 2006.

[27] J.L. Jerez, G.A. Constantinides, and E.C. Kerrigan, “An fpga implementation of a sparse quadratic programming solver for constrained predictive control,” Proc. 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp.209–218, 2011.

[28] J.L. Jerez, G.A. Constantinides, and E.C. Kerrigan, “Fpga implementation of an interior point solver for linear model predictive control,” Proc. 2010 International Conference on Field-Programmable Technology, pp.316–319, 2010.

[29] L. Hammond, M. Willey, and K. Olukotun, “Data speculation support for a chip multiprocessor,” Proc. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.58–69, 1998.

[30] M.K. Prabhu and K. Olukotun, “Exposing speculative thread parallelism in spec2000,” Proc. Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp.142–152, 2005.

[31] N. Vachharajani, R. Rangan, E. Raman, M.J. Bridges, G. Ottoni, and D.I. August, “Speculative decoupled software pipelining,” Proc. 16th International Conference on Parallel Architecture and Compilation Techniques, pp.49–59, 2007.

[32] M. Gupta and R. Nim, “Techniques for speculative run-time parallelization of loops,” Proc. IEEE/ACM Conference on Supercomputing, p.12, 1998.

[33] M. Cintra and D.R. Llanos, “Toward efficient and robust software speculative parallelization on multiprocessors,” Proc. Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp.13–24, 2003.

[34] I. Park, B. Falsafi, and T.N. Vijaykumar, “Implicitly-multithreaded processors,” Proc. 30th Annual International Symposium on Computer Architecture, pp.39–51, 2003.

[35] J.G. Steffan, C. Colohan, A. Zhai, and T.C. Mowry, “The stampede approach to thread-level speculation,” ACM Transactions on Computer Systems, vol.23, no.3, pp.253–300, Aug. 2005.

[36] P. Marcuello, A. González, and J. Tubella, “Speculative multithreaded processors,” Proc. 12th International Conference on Supercomputing, pp.77–84, 1998.

[37] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, “Trace processors,” Proc. 30th Annual ACM/IEEE International Symposium

on Microarchitecture, pp.138–148, 1997.
[38] A. Roth and G.S. Sohi, "Speculative data-driven multithreading," Proc. Seventh International Symposium on High-Performance Computer Architecture, pp.37–48, 2001.
[39] Y. Sazeides and J.E. Smith, "The predictability of data values," Proc. 30th Annual ACM/IEEE International Symposium on Microarchitecture, pp.248–258, 1997.
[40] P. Marcuello, J. Tubella, and A. González, "Value prediction for speculative multithreaded architectures," Proc. 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp.230–236, 1999.
[41] Y. Sazeides, S. Vassiliadis, and J.E. Smith, "The performance potential of data dependence speculation & collapsing," Proc. 29th Annual ACM/IEEE International Symposium on Microarchitecture, pp.238–247, 1996.
[42] B. Calder, G. Reinman, and D.M. Tullsen, "Selective value prediction," Proc. 26th Annual International Symposium on Computer Architecture, pp.64–74, 1999.
[43] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value locality and load value prediction," Proc. Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pp.138–147, 1996.
[44] M. Cintra, J.F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," Proc. 27th Annual International Symposium on Computer Architecture, pp.13–24, 2000.
[45] H. Akkary and M.A. Driscoll, "A dynamic multithreading processor," Proc. 31st Annual ACM/IEEE International Symposium on Microarchitecture, pp.226–236, 1998.
[46] M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing architectural bottlenecks to the scalability of speculative parallelization," Proc. 28th Annual International Symposium on Computer Architecture, pp.204–215, 2001.
[47] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," IEEE Trans. Comput., vol.48, no.9, pp.866–880, Sept. 1999.
[48] M.J. Garzarán, M. Prvulovic, J.M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering memory state for thread-level speculation in multiprocessors," Proc. 9th International Symposium on High-Performance Computer Architecture, pp.191–202, 2003.
[49] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," Proc. 28th Annual International Symposium on Computer Architecture, pp.2–13, 2001.
[50] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," Proc. 11th International Conference on Supercomputing, pp.68–75, 1997.
[51] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," Proc. Ninth International Symposium on High-Performance Computer Architecture, pp.129–140, 2003.
[52] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic speculative precomputation," Proc. 34th Annual ACM/IEEE International Symposium on Microarchitecture, pp.306–317, 2001.
[53] O. Mutlu, H. Kim, and Y.N. Patt, "Techniques for efficient processing in runahead execution engines," Proc. 32nd Annual International Symposium on Computer Architecture, pp.370–381, 2005.
[54] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," ACM SIGARCH Computer Architecture News, vol.29, no.2, pp.14–25, 2001.
[55] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," Proc. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp.449–460, 2012.
[56] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," Proc. Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.301–312, 2012.
[57] M. Samadi, J. Lee, D.A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," Proc. 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp.13–24, 2013.
[58] M. Samadi, D.A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.35–50, 2014.

Appendix A: Appendix

A.1 Nonlinear Spring Control System

The nonlinear spring control system simulation program is intended to stop at a position of equilibrium by applying an external force to a free vibration spring with a weight. In this program, the state value is $x = [\dot{x}, \dot{v}]^T$, and x indicates the distance from a balanced position up to a certain weight. The controller applies force u to reach the target value (balanced position) from the initial value. A schematic of a nonlinear spring is shown in Fig. 3 (a). When the weight position x is greater than zero, the spring is right-side balanced. If x is less than zero, its position is too left from the target balanced position. The initial static state is a positive position ($x = 2$), and the target state is to stop on the balanced position ($x = 0$). We formulate the equation of state for the nonlinear spring system. The velocity of weight is expressed as

$$\dot{x} = v$$

The acceleration of weight is then expressed as

$$\dot{v} = \{1 - a_0(x^2 + v^2)\}v - x + u,$$

where u is an external force (*system input* in Fig. 1). The inequality constraint is converted to equality constraint by using a dummy variable d.

$$u^2 + d^2 - u_{max}^2 = 0$$

This means $|u| \leq u_{max}$ and u_{max} is set to 0.5.

A.2 Arm-Type Pendulum Swing-Up Control System

This section describes the details of an optimal control problem for an arm-type pendulum swing-up control system. Figure 3 (b) shows the pattern diagram of the system. In the initial state, both arms are hanging down ($\theta_1 = \theta_2 = \pi$), and the MPC tries to make them stand upright, as in the target state ($\theta_1 = \theta_2 = 0$). We now formulate the equation of state. The center of gravity of each arm (X_1, Y_1) is expressed as

$$\begin{cases} X_1 = l_1 \sin\theta_1 \\ Y_1 = l_1 \cos\theta_1 \end{cases}, \begin{cases} X_2 = L_1 \sin\theta_1 + l_2 \sin\theta_2 \\ Y_2 = L_1 \cos\theta_1 + l_2 \cos\theta_2 \end{cases} \quad \text{(A· 1)}$$

The kinetic energy W, potential energy U, and loss energy

D are expressed as

$$\begin{cases} W = \sum_{i=1}^{2} \left\{ \frac{1}{2} m_i \left(\dot{X}_i^2 + \dot{Y}_i^2 \right) + \frac{1}{2} J_i \dot{\theta}_i^2 \right\} \\ U = \sum_{i=1}^{2} m_i g Y_i, \ D = \sum_{i=1}^{2} \frac{1}{2} \mu_i \dot{\psi}_i^2 \end{cases} \tag{A· 2}$$

where m_i is the mass of each arm, J_i is the moment of inertia around the center of gravity, g is the gravitational acceleration, μ_i is the viscous friction coefficient, and ψ is the relative angle. In the optimal control problem based on Eqs. (A· 1) and (A· 2), the objective function J is obtained as

$$J = \frac{1}{2} x^T (t+T) S_f x(t+T) + \int_t^{t+T} \left(\frac{1}{2} x^T(\tau) Q_x(\tau) + \frac{r_1}{2} u^2(\tau) - r_2 v(\tau) \right) d\tau$$

where the state values x are $\left[\theta_1 \ \theta_2 \ \dot{\theta}_1 \ \dot{\theta}_2 \right]^T$, S_f and Q are positive semi-definite matrices, and r_1 and r_2 are positive real numbers. The constraint condition is defined as

$$u^2 + v^2 - u^2{}_{max} = 0$$

where u is the input voltage to the motor and has a constraint $|u| \le u^2{}_{max}$. In this program, u_{max} is $2.5V$ and v is a dummy input for the constraint.

A.3 Tandem Cold Mill Control System

Tandem cold mill is a metal forming process in which metal is passed through several pairs of rolls to reduce thickness and to make the thickness uniform at low temperature. This simulation program focuses on one stand, though a tandem cold mill control system generally has several stands. The purpose of this program is to keep the thickness and tension constant when the sheet velocity changes. Figure 3 shows the pattern diagram of the tandem cold mill control system. Focusing a stand1, the entry-side thickness of the metal plate is h_0, roll gap is S_1, exit-side plate thickness is h_1, and rolling load is P_i. Interstand tension σ_{f_0} is controlled by adjusting the rotation velocity V_{R_0}. We now formulate the equation of state for this system. State variables x and *system input* u are expressed as

$$\begin{aligned} x &= [x_1, x_2, x_3, x_4]^T := [S_1, \dot{S}_1, \sigma_{f_0}, V_{R_0}]^T \\ u &= [u_1, u_2]^T := [u_{S_1}, u_{V_{R_0}}]^T \end{aligned} \tag{A· 3}$$

where u_{S_1} is a command value that implies plate thickness considering the tuning rate in the servo system. Similarly, $u_{V_{R_0}}$ is the velocity command value. The equation of state is shown as follows:

$$\dot{x} = \begin{bmatrix} x_2 \\ f_2 \\ f_3 \\ -(1/T_m) x_4 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ K_g/T_g & 0 \\ 0 & 0 \\ 0 & 1/T_m \end{bmatrix} \begin{bmatrix} u_{S_1} \\ u_{V_{R_0}} \end{bmatrix}$$

$$f_2 = -\frac{K_g}{T_g} x_1 - \frac{1}{T_g} x_2 - \frac{K_g}{T_g} \frac{k}{M_0} P_i$$

$$f_3 = \frac{E}{L} \{ \frac{h_1}{h_0} (1 + f_i) V_{R_1} - (1 + f_{i-1}) x_4 \}$$

Table A· 1 Parameters for tandem cold mill system

K_g [-]	AGC gain	10
T_g [s]	Time constant of hydraulic servo	0.01
k [-]	Tuning ratio	0.4
M_0 [N/mm]	Mill modulus	$4.9 * 10^6$
T_m [s]	Time constant of mill motor	0.003

where P_i and forwarding slip ratio f_i are nonlinear functions that have interstand tension and rotation velocity as input values. In addition, Table A· 1 shows each parameter for this system.

$$\begin{aligned} P_i &:= P_i(\sigma_{f_{i-1}}, \sigma_{f_i}, h_i, h_{i-1}, V_{R_i}) \\ f_i &:= f_i(\sigma_{f_{i-1}}, \sigma_{f_i}, h_i, h_{i-1}, V_{R_i}) \end{aligned}$$

As mentioned above, the purpose of this control system is to keep the thickness and tension constant; however, we cannot monitor the thickness as an observed value in Eq. (A· 3). Therefore, we also need to formulate thickness h_i on stand i. Considering h_i is always controlled near target value h_i^r, h_i can be approximately represented as $h_i^r + \Delta h_i$ (Δh_i denotes the infinitesimal difference from the target value).

$$h_i \approx h_i^r + \frac{P_i(\sigma_{f_{i-1}}, \sigma_{f_i}, h_i, h_{i-1}, V_{R_i}) - M_0(h_i^r - S_i)}{M_0 - (\partial P_i / \partial h_i)|_{h_i = h_i^r}}$$

This function explicitly calculates h_i. The input rotation velocity is given in Fig. 3 (e.g., V_{r_0} is input for a stand1). The target values are thickness $h_1^r = 3.4$ mm and tension $\sigma_{f_0}^r = 20$MPa.

Satoshi Kawakami received the B.E. and M.E. degrees in Information Science and Electrical Engineering from Kyushu University in 2012 and 2014, respectively. During 2014-2016, he worked in Bosch Corporation as an engineer. He is currently a Ph.D. student and technical staff in Kyushu University. His research interests include parallel computing and nanophotonic computing. He is a member of ACM and IPSJ.

Takatsugu Ono received a Ph.D. degree from Kyushu University, Japan, in 2009. He was a researcher for Fujitsu Laboratories Ltd., Kawasaki, Japan, and engaged in developing a server for a data center. He is currently an assistant professor in the Faculty of Information Science and Electrical Engineering at Kyushu University. His research interests include the area of memory architecture, secure architecture, and supercomputing. He is a member of the IEEE, IPSJ, and IEICE.

Toshiyuki Ohtsuka received the B.E., M.E. and Ph.D. degrees from Tokyo Metropolitan Institute of Technology, Japan, in 1990, 1992 and 1995, respectively. From 1995 to 1999, he worked as an Assistant Professor at the University of Tsukuba. In 1999, he joined Osaka University, and he was a Professor at the Graduate School of Engineering Science from 2007 to 2013. In 2013, he joined Kyoto University as a Professor at the Graduate School of Informatics. His research interests include nonlinear control theory and real-time optimization with applications to aerospace engineering and mechanical engineering. He is a member of IEEE, SICE, ISCIE, JSME, JSASS, and AIAA.

Koji Inoue received the B.E. and M.E. degrees in computer science from Kyushu Institute of Technology, Japan in 1994 and 1996, respectively. He received the Ph.D. degree in Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan in 2001. In 1999, he joined Halo LSI Design & Technology, Inc., NY, as a circuit designer. He is currently a professor of the Department of I&E Visionaries, Kyushu University. His research interests include power-aware computing, high-performance computing, secure computer systems, 3D microprocessor architectures, multi/many-core architectures, nano-photonic computing and quantum computing.