Deploying Hard Real-time Control Software on Chip-multiprocessors

Dai N. Bui
University of California, Berkeley
EECS Department
Berkeley, U. S. A.
daib@eecs.berkeley.edu

Hiren D. Patel
University of Waterloo
ECE Department
Waterloo, Canada
hdpatel@uwaterloo.ca

Edward A. Lee
University of California, Berkeley
EECS Department
Berkeley, U. S. A.
eal@eecs.berkeley.edu

***Abstract*—Deploying real-time control systems software on multiprocessors requires distributing tasks on multiple processing nodes and coordinating their executions using a protocol. One such protocol is the discrete-event (DE) model of computation. In this paper, we investigate distributed discrete-event (DE) with null-message protocol (NMP) on a multicore system for real-time control software. We illustrate analytically and experimentally that even with the null-message deadlock avoidance scheme in the protocol, the system can deadlock due to inter-core message dependencies. We identify two central reasons for such deadlocks: 1) the lack of an upper-bound on packet transmission rates and processing capability, and 2) an unknown upper-bound on the communication network delay. To address these, we propose using architectural features such as timing control and real-time network-on-chips to prevent such message-dependent deadlocks. We employ these architectural techniques in conjunction with a distributed DE strategy called PTIDES for an illustrative car wash station example and later follow it with a more realistic tunnelling ball device application.**

***Index Terms*—Real-time software, Chip-multiprocessors, Discrete-Event.**

I. Introduction

The use of multiprocessors for real-time control systems requires distributing the real-time software on the various processing nodes, and coordinating their execution and communication with a protocol. Two common protocols are time-triggered and event-triggered. With either of these protocols, it is critical to ensure that the system is deadlock free. In this paper, we focus on the design of reliable real-time systems on chip-multiprocessors (CMPs) using the event-triggered protocol; in particular, the discrete-event (DE). However, we do not consider a monolithic event queue because of its inability to exploit parallelism, and its susceptibility to being a single point of failure. Alternatively, we concentrate on distributing the discrete-event execution across multiple processing nodes. In doing this, we study the effects of distributing DE with a deadlock avoidance mechanism known as the null-message protocol (NMP) [1], [2]. Specifically, we evaluate a potential message-dependent deadlock [3], [4] problem that arises even when NMP is used.

The architecture we use is a CMP with multiple processing nodes connected via a network-on-chip (NoC). A processing node connects to a network interface, which is directly connected to the NoC interconnect. Designers of NoCs often assume that packets transmitted to a processing node are always consumed immediately. With this assumption, the designer provides guarantee that sent packets are always delivered. This means that once a packet is sent, it will reach the destination within a finite amount of time. Therefore, there is no deadlock or livelock in the network that may cause a packet to never reach its destination. However, in implementations of CMPs, processing nodes (i.e. CPUs) have limited memory and processing resources; therefore, processing nodes cannot always consume packets as soon as they arrive. If too many packets arrive at a processing node during an interval, then they are usually queued up in the network. This results in a blocking effect in the network, which might cause the system to deadlock entirely or partially. This is called *message-dependent deadlock* [3], [4].

In the case of NMP, each processing node regularly sends null messages to some other processing nodes to update those nodes about the sender's physical time. It is very possible that when too many null messages (packets) are sent to the same receiving node, if that node is busy doing some task then it cannot process these messages. Therefore, these null messages fill up the input buffer at the network interface of the node. This congestion prevents other non-null messages from being processed quickly as well. This temporary blocking effect is problematic for real-time systems because it might cause the system to miss its real-time deadline simply due to a congestion caused by null messages. Moreover, if the buffer capacity reaches its maximum, then the null messages can result in blocking the entire network; essentially, a deadlock.

We show that distributing DE on a CMP using the NMP protocol can result in message-dependent deadlocks. These deadlocks arise from two factors: 1) bursty traffic due to network jitter and 2) mismatch in transmit and receive rates. We identify these factors via an analytical model based on the ideas of network calculus [5] and real-time calculus [6]. Then, we discuss architectural features that can address these factors. In particular, we present an implementation that combines a real-time network-on-chip communication network that eliminates jitter [7] with a real-time processor architecture called the Precision Timed (PRET) machine [8] with instructions for timing control. We show that by using timing instructions in software we can enforce a bound on the message transmit-

ting/receiving rates between nodes. This in turn provides a bound on the number of messages at the destination nodes within an interval of time. Furthermore, instead of employing NMP, we propose the use of a programming model called Programming Temporally Integrated Distributed Embedded Systems (PTIDES) [9] that eliminates the need to send null messages over the network.

A. Organization

This paper is organized as follows: in Section III, we give a brief overview of network on-chip interconnection. The Section IV is dedicated to building an analytical model for communication network. We then discuss the requirements for the implementation from the analytical model in Section V. An illustration of the MDP is demonstrated in Section VI. Finally, Section VII is dedicated to showing a more realistic distributed DE example on a multicore system.

II. Related Work

Current approaches that employ CMP architectures with network-on-chips to avoid deadlock either increase the number of virtual channels, buffer sizes [10], or they use a deadlock resolution mechanisms [11]. However, simply increasing the buffer size and number of virtual channels without considering the transmit/receive rates is not safe. This is because it is possible for a node to continuously send more packets than the destination node can receive resulting in congestion and potentially message-dependent deadlocks. Deadlock resolution mechanisms are often complicated since they require an end-to-end flow control mechanism as in TCP/IP so that a sending node has to resend a packet when this packet is *killed* by the deadlock resolution mechanism. However, for real-time control systems, simply dropping packets may have adverse effects on the application's timing requirements, which can cause the application to miss critical deadlines.

Although message-dependent deadlock (MDP) happens *infrequently*, for safety-critical and real-time control systems, any possibility of a deadlock has to be excluded completely. In [12], whenever an MDP occurs and it is detected, an intermediate node has to consume some messages, store them in its local memory and then resend those messages when the network is freed. This mechanism is unsuitable for real-time systems. In the active message communication model [13], deadlock is avoided by making the receiving nodes always sink a message when it arrives. However, to successfully implement that, receiving nodes have to be fast enough to process all received messages before a new message arrives. As we can see flooding other nodes with null messages as in the NMP might hinder this approach. A recent work [14] on solving the problem using end-to-end flow control mechanism is only suitable for multimedia applications.

III. Interconnection Network on Chip

A. Network on Chip

Network on-chip (NoC) is a new design paradigm for system-on-chip (SoC) [15], [16]. Network-on-chips often uses wormhole packet switching [17]. A packet is divided in to smaller data units called flits (flow units) as in Figure 1(a). The *head* flit contains routing and other information for routers to route the packet. In wormhole switching, buffers, i.e. in a router, are allocated to flits rather than packets. So a packet is sent from one router to another router gradually flit by flit. Thus, a packet can span over multiple routers and buffers causing blocking to other packets.

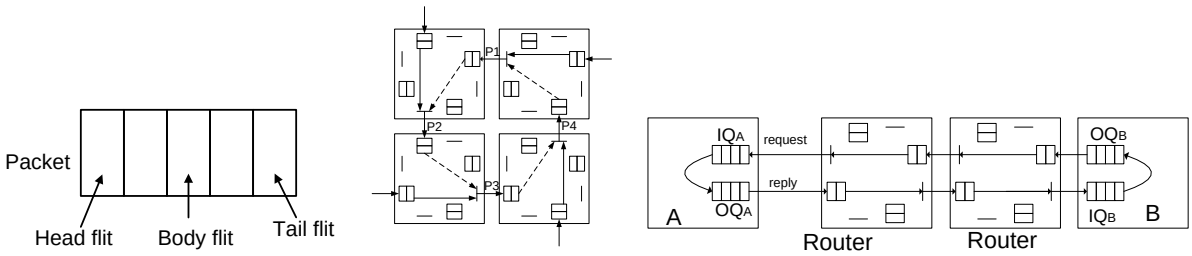
B. Deadlock-free Interconnection Network

An interconnection network used to connect processing elements such as the network on-chip in Figure 4(a) is composed solely of routers. The interconnection network is deadlock free if the routing function in the routers of the network does not cause any *routing-dependent deadlock* as in Figure 1(b), in which packets create a cyclic loops [18]. In Figure 1(b), deadlock happens when four packets P_1, P_2, P_3, P_4 wait for buffer space occupied by each other in a loop and all buffers are full, therefore no packet can advance.

C. Message-Dependent Deadlock

In a multicore system that uses network on-chip interconnection, although the communication network is deadlock free, message-dependent loops created by processing nodes might cause deadlocks in the multicore system [3], [4], [11]. This deadlock is sometimes called *request* $\prec$ *reply* dependency deadlock. Intuitively, the processing nodes process requests then sometimes send out a reply message. This *request* $\prec$ *reply* dependency might form cyclic dependency loops in the whole systems as in Figure 1(c). Different form routing-dependent deadlocks, in the message-dependent deadlock, the message-dependent loops are created at processing nodes. For example, when node A has a new pending request req_B from node B but it first has to send out a reply rep_A of some previous request from node B to free its internal memory in order to consume req_B. However, rep_A cannot be sent out due to buffers in the network are full that need node B to consume some messages to free the network buffer. However, node B also cannot consume any messages since its internal memory is full and it cannot send out a message to node A because the buffers in the network are full. Both node A and B wait for each other to consume packets but none of them can then the deadlock happens.

The progress of deadlock formation is as follows. Let IQ_A, OQ_A, IQ_B, OQ_B be input, output queues of nodes A, B respectively. When node A sends packets to node B, if it sends packets faster than node B can process then the packets will queue up at output links and buffers at routers around node B. When the buffers at routers around node B are full, this effect will block other normal packets. Other normal packets then fill up buffers at other routers. Gradually, this congestion will propagate to output of node A, then input of node A. Then node A cannot send/receive and any packet. At this point, none of the processes in the system can make progress as there are not enough memory space therefore the system becomes



(a) Format of a packet. (b) Routing-dependent deadlock. (c) Message-dependent deadlock.

Fig. 1. Interconnection network

deadlocked. This is deadlock phenomenon is not just a system with a congested network.

IV. Analytical Model

We construct our analytical model similar to real-time [6] and network calculus [5]. We use this model to identify message-dependent deadlocks and to specify the conditions that must be preserved in order to avoid them. Later, we will show implementation techniques that aid designers in satisfying these conditions.

A network constitutes of a set of processing nodes connected via an interconnection network. Let V be the set of processing nodes and $C \subset \mathbb{N}$ be the set of virtual channel indexes representing the connections between the nodes. The set of flows in the network $F \subseteq V \times V \times C$ describes paths on the interconnection network that is used for communication between nodes. One flow $f \in F$ is a 3-tuple, $f = (s, d, c)$, which denotes a path from a source node s to a destination node d with virtual channel index $c \in C$. It is typical for a node to have additional buffer space reserved for a particular flow, which we denote by B_f.

The source node s can transmit $x_f(t)$ amount of data at a time instant $t \in \mathbb{R}$ on flow f. Similarly, $a_f(t)$ defines the amount of data arriving at the destination node at time instant t on flow f. Note that both $x_f(t)$ and $a_f(t)$ describe the maximum amount of traffic being transmitted and received. The destination node d consumes a certain amount of data from flow f, which we describe as $r_f(t)$. This is the minimum amount of data that can be processed by the destination at time t. Since we are interested in network traffic over a period of time, we augment transmission, arrival and processing rates to support time intervals. Hence, for a time interval $[t_0, t_1]$ where $t_0, t_1 \in \mathbb{R}$, $X_f(t_0, t_1) = \int_{t_0}^{t_1} x_f(t)dt$ is the maximum amount of data transmitted, $A_f(t_0, t_1) = \int_{t_0}^{t_1} a_f(t)dt$ is the maximum amount of data arrived, and $R_f(t_0, t_1) = \int_{t_0}^{t_1} r_f(t)dt$ is the minimum amount of data the node processes.

We characterize the amount of data in a flow as $N_f(t_0, t_1) = A_f(t_0, t_1) - R_f(t_0, t_1)$. It is possible for a flow to become congested. This occurs when more data arrives at the destination node than the amount of data processed. This is, of course, if the flow does not allocate sufficient buffer space on it.

Definition 1: A message-dependent deadlock (MDP) occurs when there exists a cyclic dependency between multiple congested flows such that none of the nodes in the dependency can consume and produce data on its flows.

To avoid MDPs, we must satisfy condition (1) for every flow within the cyclic dependency. This condition ensures that adequate buffer space is allocated to a flow using the arrival and processing rates.

$$A_f(t_0, t_1) - R_f(t_0, t_1) \leq B_f, \forall t_0, t_1.\ t_1 \geq t_0 \geq 0 \quad (1)$$

Notice that condition (1) is both sufficient and necessary to exclude data congestion resulting in MDPs. However, enforcing this condition is nontrivial. This is because it requires controlling A_f and R_f. Moreover, A_f depends on the properties of the underlying communication infrastructure such as the routing, switching and flow control policies. To address this issue, we abstract the communication infrastructure details to two main constituents: the minimum network delay D_f and the maximum possible jitter Δ_f for a flow f.

Observation 1: By using the minimum network delay D_f and the maximum possible jitter Δ_f on flow f, we can derive A_f using the transmission rate X_f.

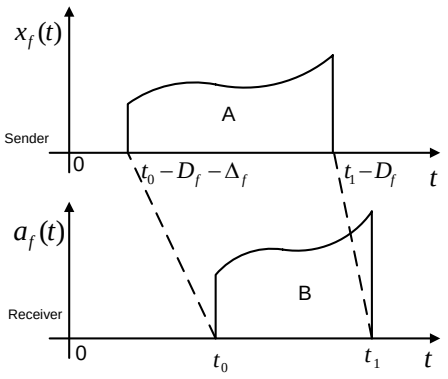


Fig. 2. Effect of traffic distortion on send/arrival rates.

We describe the intuition behind observation 1 using Figure 2, which shows the relationship between the transmission and arrival of data for an interval $[t_0, t_1]$. The bottom graph shows the amount of data arriving between the interval $[t_0, t_1]$ at the destination node. To encode this in terms of the transmission, we observe that the data arriving at the destination

node at t_0 is sent at the **latest** from the source node at time $t_0 - D_f - \Delta_f$. This takes into account the network delay and the jitter. For t_1 however, the **earliest** that the source can transmit the data is $t_1 - D_f$. Notice that we take conservative bounds. Hence, the maximum amount of traffic arriving at a destination node d in between $[t_0, t_1]$ must be transmitted by source node s within the interval $[t_0 - D_f - \Delta_f, t_1 - D_f]$. Intuitively, the area of the region B is equal to the area of the region A in the graphs in Figure 2. We use these bounds to describe the following: $\int_{t_0-D_f-\Delta_f}^{t_1-D_f} x_f(t)dt = \int_{t_0}^{t_1} a_f(t)dt$. And since $A_f(t_0, t_1) = \int_{t_0}^{t_1} a_f(t)dt$ and $X_f(t_0 - D_f - \Delta_f, t_1 - D_f) = \int_{t_0-D_f-\Delta_f}^{t_1-D_f} x_f(t)dt$, this allows us to describe the following condition:

$$A_f(t_0, t_1) = X_f(t_0 - D_f - \Delta_f, t_1 - D_f) \quad (2)$$

Combining conditions (1) and (2), $\forall t_0, t_1 \in \mathbb{R}$ such that $t_1 \geq t_0 \geq 0$, we obtain

$$X_f(t_0 - D_f - \Delta_f, t_1 - D_f) - R_f(t_0, t_1) \leq B_f \quad (3)$$

Condition (3) captures the criterion for avoiding MDPs with respect to transmission rate, processing capabilities, minimum network delay and the jitter in the network.

A. Factors that Contribute to Message-Dependent Deadlocks

We identify two factors that contribute to MDPs: 1) temporary or permanent mismatch between transmit/receive rates, and 2) bursty traffic caused by message jitter in the network.

1) Mismatch of Transmit/Receive Rates: It is clear from condition (3) that if sufficient buffer capacity is not allocated, then whenever data is transmitted faster than it is processed, the unconsumed packets can overflow the buffers resulting in MDPs.

2) Bursty Traffic due to Increased Jitter: If the jitter Δ_f increases then the amount of traffic $X_f(t_0 - D_f - \Delta_f, t_1 - D_f)$ may also increase as the length $t_1 - t_1 + \Delta_f$ of the interval $[t_0 - D_f - \Delta_f, t_1 - D_f]$ increases, accordingly, the amount of traffic sent during the interval is increased. The message jitter occurs because of the best-effort routing schemes often employed in network-on-chip architectures.

Figure 3 illustrates how bursty traffic can arise from message jitter [19]. This occurs even when the sender guarantees that packets are transmitted at regular intervals, after traversing through three routers, the intervals between them may be reduced. This appears as bursty traffic to the destination node. This phenomenon happens because packets have to compete for resources such as buffers and physical links in a network. This causes a packet's arrival to get closer to the previous one when the previous packet has to wait for resources. For a large network-on-chip composed of hundreds of nodes, packets might have to traverse several hops, which may increase the message jitter and result in severe bursty traffic. This effect can then cause a node to temporarily be flooded with messages; thus, the external network may be blocked resulting in system deadlock.

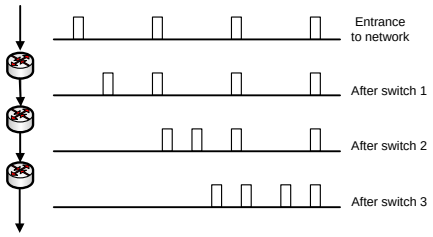


Fig. 3. Jitter of messages.

V. Requirements from an Implementation

Our analytical model in Section IV makes four assumptions. We list these assumptions and describe how an implementation can provide the necessary techniques to satisfy the assumption. In the following Section we explain these techniques in further details.

Assumption 1: The maximum network jitter Δ_f for a flow f is known.

Assumption 2: The minimum network delay D_f for a flow f is known.

Assumption 3: It is possible to control $R_f(t_0, t_1)$, which is the processing capability of a node for flow f between some time interval $[t_0, t_1]$.

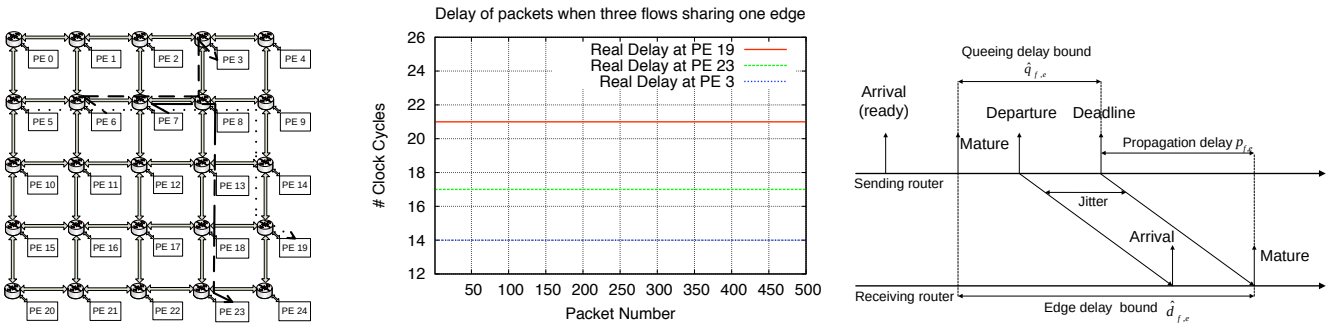
Assumption 4: It is possible to control $X_f(t_0, t_1)$, which is the amount of data being transmitted by a processing node on flow f between some time interval $[t_0, t_1]$.

A. A Known Maximum Jitter via Guaranteed-service On-chip Communication

We need to satisfy assumption 1 to guarantee that condition (3) holds. Fortunately, there are guaranteed-service on-chip communication networks that provide specific values for the jitter. The networks ensure that messages reach their destination within a known amount of time regardless of other traffic on a network. For example, [7], [20] guarantee that their network guarantees no jitter ($\Delta_f = 0$). There are other real-time on-chip communication networks guarantee an upper-bound on Δ_f such as the Æthereal architecture [20] where a time-division multiplex access method is applied to guarantee the real-time delay of real-time packets.

We use the real-time network on chip communication interconnect presented by Bui et al. [7]. This interconnection network requires a clear specification of the set of possible flows, and based on that, a suitable path is found in a system to meet the real-time constraint of each flow if and only if such a that path exists. This mechanism divides the end-to-end delay of a flow into local delays at each hop of the path of the flow. The scheduling mechanism at each router will hold packets forwarded from its previous routers for the rest of the packet's local delay if the packet is forwarded early. Thereby, all the jitters are absorbed at routers.

For example, Figure 4(a) describes three real-time flows on a network for a simple traffic controller example. The traffic pattern of each flow is characterized by the maximum packet length in flits of each flow and the minimum interval



(a) A traffic Scenario. (b) Message delay of the three real-time flows. (c) Timing diagram of packet forwarding.

Fig. 4. Real-time communication network

between two successive transmissions of packets. For the traffic controller example, the specifications for the three flows are as follows: $F_1 = (PE_7 \rightarrow PE_{23}, 5$ flits, 21cycles), $F_2 = (PE_6 \rightarrow PE_3, 3$ flits, 19 cycles) and $F_3 = (PE_5 \rightarrow PE_{19}, 4$ flits, 17 cycles). The packets sent from source nodes of the three flows reach their destination nodes within a bounded amount of time regardless of other traffic as in Figure 4(b). The figure shows that it will take the packets of one real-time flow the same amount of time to reach their destination.

Figure 4(c) shows the timing diagram of a packet at a node. Consider a flow f along a path π_f and let $e_n(v_n, v_{n+1})$ be the n^{th} edge on the path and the upper bound on the queueing delay is denoted by $\hat{q}_{f,e}$, which is guaranteed by a scheduling algorithm. Let the maturation time of a packet be the latest time a packet p is expected to arrive at the node denoted by $m_{f,n}^{(p)}$. The deadline of a packet p of flow f at node v_n is defined as follows:

$$T_{f,n}^{(p)} = m_{f,n}^{(p)} + \hat{q}_{f,e_n} \tag{4}$$

The actual sending time of the packet is denoted by $S_{f,n}^{(p)}$ and the jitter for each packet is computed as follows:

$$j_{f,n}^{(p)} = T_{f,n}^{(p)} - S_{f,n}^{(p)} \tag{5}$$

The jitter measures the length of time from the departure of a packet to its deadline. Denoting by $r_{f,n+1}^{(p)}$ the arrival time of packet p at next node v_{n+1}, the following holds: $m_{f,n+1}^{(p)} = r_{f,n+1}^{(p)} + j_{f,n}^{(p)}$. Combining this last equation with Equations (4) and (5), we obtain:

$$j_{f,n+1}^{(p)} = r_{f,n+1}^{(p)} + j_{f,n}^{(p)} + \hat{q}_{f,e_{n+1}} - S_{f,n+1}^{(p)} \tag{6}$$

This equation suggests that the maturation time of a packet can be computed using only two local parameters: the jitter from the previous node, and the upper bound on the queueing delay. Moreover, these parameters are used to compute the jitter value to be sent to the next node on the path. Using the jitter information allows to have a completely distributed scheduling algorithm.

As we can see that, if a packet p is forwarded before mature time $m_{f,n}^{(p)}$ of the packet at a node v_n, the next node v_{n+1} will hold the packet until that mature time so that the local delay of the packet on the edge $e_n(v_n, v_{n+1})$ is always at equal to $\hat{q}_{f,e}$ therefore the end-to-end delays of all packets of a flow f are the same.

B. Analyzing Minimum Network Delay

In order to address assumption 2, we compute D_f experimentally for the underlying communication network.

The minimum network delay D_f for a flow f is straightforward to compute as the minimum delay of a packet of a flow f is simply the delay when there are no other packets on the network. This delay can be computed statically based on each specific network architecture. For example, the delay D_f of a best effort flow can be analysed as follows: the head flit travels from the input queue to the output queue of a router in s cycles where s is the number of pipeline stages of the router. Then it takes the packet's head flit 1 cycle to travel the a link from the output queue of the router to the next router's input queue. Therefore the end-to-end delay of the head flit of a packet on a path is $(s+1) * h$ where h is the number of hops on the path. The remaining flits of the packet follow suit.

C. Controlling Processing Capability and Transmission Rates via Timing Instructions

To control the processing capability (assumption 3) and the maximum amount of data transmitted (assumption 4) from a processing node, we propose using predictable real-time architectures that provide repeatable timing behaviors. One such architecture is the Precision Timed (PRET) architecture [8]. The objective of PRET is to exhibit predictable and repeatable timing behaviors. Predictable timing behaviors simplify the process of analytically determining the worst-case execution time of a program. This is beneficial for addressing assumption 3 because the worst-case execution time gives a conservative upper-bound on the execution time of the program.

The processing capability of a node for flow f between some time interval $[t_0, t_1]$, $R_f(t_0, t_1)$ can also be measured by using the PRET architecture [8]. The PRET architecture

using interleaving pipeline architecture to avoid memory access latency effects. Therefore, the worst-case execution time of a segment code can be measure, hence the $R_f(t_0, t_1)$ processing capability during in interval $[t_0, t_1]$ can be derived. For example, for the following source code:

```
1 while(not_terminated) {
2     receive(source_node, message);
3     do_computation();
4 }
```

The worst-case execution time of the `do_computation()` function will determine the worst-case interval between two message receiving commands, hence $R_f(t_0, t_1)$ is derived.

Repeatable timing behaviors, on the other hand, means that we observe the same timing behaviors for a program even with varying input stimuli. PRET support predictability and repeatability through judicious redesign of the architecture and extensions that support timing instructions [8], respectively. It is specifically designed for hard real-time systems.

One of the central features of PRET is the instruction-set architecture extension to include timing instructions. These timing instructions allow programmers to directly specify their timing requirements with their functionality. An example of our use of a timing instruction is the *deadline* instruction. This instruction guarantees the minimum interval between transmitting packets. For example, Listing 1 represents a typical program segment used to transmit packets.

Listing 1. Code template without timing instruction

```
1  while(not_terminated) {
2    ...
3    send(dest_node, message1);
4    i = 0;
5    while(i < 100) {
6         ...
7         if(some_condition)
8            break; // Changes execution time
9         ...
10        i++;
11   }
12   send(dest_node, message2);
13   ...
14 }
```

Note, however, the interval between message transmissions is determined by the execution time of the code segment between the two send commands (lines 3 and 12). This execution time may also vary due to data-dependent control flow paths. An example of this is shown in lines 7 and 8 where `break` may reduce the execution time between the first and second send commands. This means that messages are sent to the receiving node faster. Then, the receiving node might be flooded with messages if its processing capability and buffer capacity are not adjusted accordingly. If the receiving node is flooded with messages, a potential MDP might occur. This is because the receiving node may perceive quicker send commands from the source node as bursty traffic at its end.

We address this issue by making minor modifications to the program code for the PRET architecture as shown in Listing 2.

Listing 2. Code template with timing instruction

```
1  while(not_terminated) {
2    ...
3    //interval to next send command approx. 2000 cycles
4    DEADLINE(period);
5    send(dest_node, message1);
6    i = 0;
7    while(i < 100) {
8         ...
9         if(some_condition)
10           break;
11        ...
12        i++;
13   }
14   //interval to next send command approx. 2000 cycles
15   DEADLINE(period);
16   send(dest_node, message2);
17   ...
18 }
```

In the modified code, we insert two `DEADLINE` instructions with the `period` as an argument. For this example, we specify the `period` to be 2000 cycles to ensure that the interval between the two message send commands to the destination node is never less than 2000 cycles regardless of processor speed and/or the lengths of execution paths of the program. Of course, this requires that the worst-case execution time of the program segment between the two send commands is less or equal to the `period`. As the interval between two send commands is guaranteed to be always larger than some certain value, a node will never send messages faster than it is allowed; thereby, the destination node is never flooded with messages. For example, if a transmitting node sends messages of packet size p and we know the following:

- The interval between messages $\frac{p((t_1-D_f)-(t_0-D_f))}{X_f(t_0-D_f,t_1-D_f)} \geq 2000$ is guaranteed by the deadline instructions.
- $\Delta_f = 0$ is guaranteed by the real-time communication network with jitter control.

Then, $X_f(t_0 - D_f, t_1 - D_f) \leq p\frac{t_1-t_0}{2000}$, which means that the receiving node's processing capability is

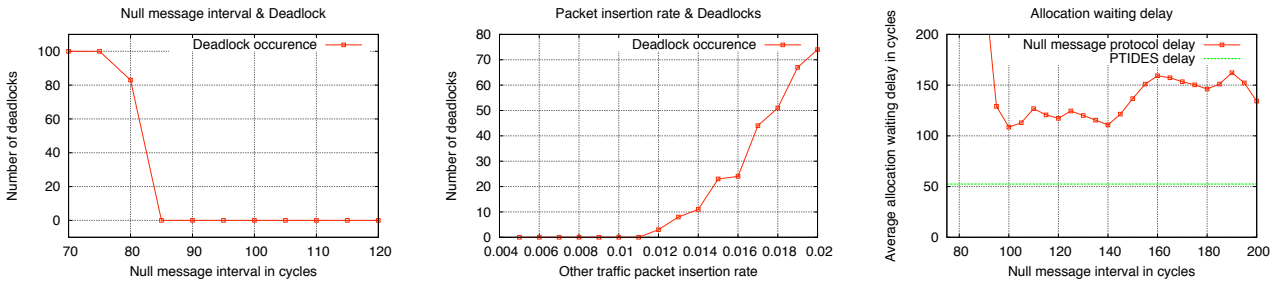
$$R_f(t_0, t_1) \geq p\frac{t_1 - t_0}{2000} - B_f \Leftrightarrow B_f \geq p\frac{t_1 - t_0}{2000} - R_f(t_0, t_1)$$

This indicates that the sufficient and necessary condition (3) in section IV is satisfied.

We can restrict the sending rates with the PRET architecture; however, other alternatives can be used as well. For example, timer counters with predictable architectures may be used.

VI. An Illustrative Example of MDP: Car Wash Stations

We borrow the *car wash* [2] example as shown in Figure 6 to describe a DE system. In this example, there are five processing components: a source, an attendant, two car wash stations denoted by CW_1 and CW_2, and a sink. The source forwards cars to the attendant that dispatches cars to the car wash stations. The attendant dispatches a waiting car to the car wash station that is idle earliest. A request message for another car from the car wash station informs the attendant that the car wash is idle. This message contains a timestamp denoting the physical time at which the car wash station became idle. The attendant uses this timestamp to compare and identify the car wash station to allocate the waiting car. Once a car



(a) Effect of null message interval on deadlock frequency.

(b) Effect of other traffic load on deadlock frequency of NMP.

(c) Average allocation delay by two approaches.

Fig. 5. Configurations and evaluation results

wash completes its process, it sends the car to the sink. In doing so, it sends a message with the timestamp at which the car completed its wash. The sink then orders the washed cars according to their completion timestamp order.

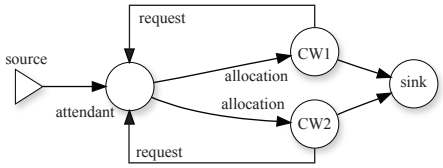


Fig. 6. Message flow of the car wash example.

Implementing this DE car wash system on a single processor is straightforward. It requires ordering an event-queue on timestamps, and the front of the event-queue contains the next event to process. Therefore, it is easy to select the next event to process. On a multiprocessor system, however, the processing components are distributed, and each of the processors has its own ordered event-queues. Since none of the individual processors have global knowledge of all the events at any point in time, it is difficult to determine the earliest event to process.

In the case of the car wash example, let us assume that the processing components are distributed on a separate processor, and they communicate with each other over a network that exhibits variable latencies. Now, suppose that the attendant is biased and sends the waiting cars only to CW_1. Upon completing the washes, CW_1 sends the cars to the sink, but since the sink is unaware of this biased routing, it will wait for a message from CW_2. In fact, this will cause the system to deadlock because the sink will not be able to complete the wash until it has successfully ordered the cars based on their timestamps, and to do this, it requires a message from CW_2. To address this issue, the null message protocol (NMP) [1], [2] was proposed.

NMP solves the deadlock issue by periodically sending *null* messages from one processing node to another if there is no real message to send. The null messages update the receiving nodes with the latest physical time of the sending node. The car wash example with NMP would then require CW_2 to send null messages to the sink periodically. This allows the sink to compare the timestamps of the messages from CW_1 and the null messages from CW_2, and if it determines that the message from CW_1 has an earlier timestamp, then that car is completed first. For example, CW_1 sends a car c_1 to the sink with a timestamp t_1 and CW_2 periodically sends null messages with timestamps to the sink. If the sink receives a null message n_2 with timestamp $t_2 > t_1$, then the sink node knows that there is no pending car from CW_2 until t_2, therefore the car c_1 can be sent out. So the system cannot be deadlocked.

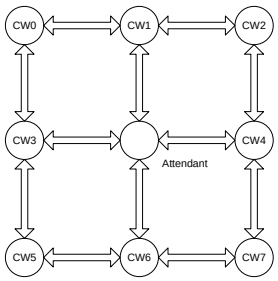


Fig. 7. Simulation scenario for car wash example

To demonstrate the potential deadlock of NMP on a multicore system, we set up a simple simulation scenario of the car wash example. This scenario is shown in Figure VI, where there is a 3×3 network. The source and attendant share the same center node. The outer nodes are 8 car wash nodes. For the sake of simplicity, we discard the sink node. We map one task onto one processor and we minimize the total communication latency between the cores.

The simulation steps are as follows: 1) A car wash node sends a *request* message to the attendant node whenever it is idle (not busy washing any car). The request message also contains a timestamp of the time of the node when the request message is sent. 2) The attendant node will allocate a car to a washing node by an *allocation* message whenever it receives a request message. However, the attendant node requires that cars be allocated to washing node in an increasing order of the timestamp in request messages. This means that, if there are 2 request messages m_1 and m_2 from node n_1, n_2 with timestamp $t_1 < t_2$ respectively, although m_2 arrives at the attendant node *before* m_1 due to different network delays, node

n_1 is still allocated a car before n_2. 3) A car wash node, whenever allocated a car, will wash a car within a specified amount of time. When it finishes washing the car, it sends another request message to the attendant node to ask for new cars to wash.

A. NMP Deadlock Scenario

The car wash example in Figure 6 is vulnerable to MDPs if there are several washing nodes and those nodes frequently send out null messages to update the attendant and sink nodes about their current progress time. If the attendant and sink nodes at some time receive too many null messages from the washing nodes, partially due to the traffic pattern distortion of a packet switching network that cause time intervals between messages to become smaller as in Figure 3 in [19], then those receiving nodes cannot process all null message packets on-time. This, coupled with some other bursty traffic like memory access traffic, might cause congestion at the links around those receiving nodes similar to the phenomenon in Section III-C. This congestion then might causes those receiving nodes to be unable to send out messages, car assignment messages in case of the attendant node and car delivery messages in case of the sink node. Since those nodes cannot send out messages, they cannot free their internal buffers to receive more packets. Till this time, the system becomes deadlocked.

Consider another situation when each washing node is supposed to send null messages every 85 cycles, but due to improper timing or decreased workload, it sends messages faster at a rate of 80 cycles then deadlock can happen quickly within 50,000 cycles after that. To avoid this situation, we can use an architecture like PRET [8] that does not allow some work (sending message) to be done faster than needed as in Section V-C.

Figure 5(a) shows the interaction between the null message interval and deadlock frequency. There is a sharp threshold where deadlock turns from never happening to happening frequently. This occurs because car wash nodes send null messages faster than the attendant node can handle. For example, if car wash nodes send null messages at the rate 1 null message per 80 cycles, since there are 8 car wash nodes, null messages will arrive at the attendant node every 10 cycles. Sometimes some request messages arrive at the attendant node also, so null messages and request messages will arrive at the attendant node every interval less then 10 cycles. Because the attendant can process one message in 10 cycles, it cannot process all arrival messages, the condition (3) is violated. The mismatch between the arrival rate and consumption rate at the attendant node might cause the system to be deadlocked, we can avoid this deadlock beforehand by setting the interval between null messages to a larger enough value, say 85 cycles to satisfy (3).

B. PTIDES Execution Strategy

As null messages can cause deadlocks to a system without a proper implementation, a good DE execution strategy can eliminate such type of messages. In this section, we evaluate a new execution strategy called Programming Temporally Integrated Distributed Embedded Systems - PTIDES [9] as a replacement for NMP. For PTIDES, instead of waiting for null messages from washing nodes to the attendant node, the attendant node uses the passage of real-time. Please note that PTIDES does not eliminate the deadlock problem by itself, rather, it eliminates the null messages that potentially cause deadlocks.

We briefly explain the basic PTIDES execution strategy [21] in the context of the car wash example. PTIDES requires a strict packet delay bound to guarantee the discrete-event semantics. A guaranteed service on network on-chip architecture in Section V-A can be used as the underlying communication for PTIDES on a network on-chip multicore system. Different from the NMP, PTIDES does not use null messages to avoid protocol deadlock. Instead, PTIDES uses the delay bound of a message in a network to guarantee the DE semantics. Suppose that a request message m_i sent from a car wash node CW_i will reach the attendant node within the delay bound $d(CW_i)$.

The attendant node receives a request message m_1 from car wash CW_1 with timestamp t_1. The attendant node knows that it is safe to dispatch a car to CW_1 when: 1) Either the attendant node has received all request messages from other nodes and all the other request messages have timestamp greater than m_1. 2) Or current physical time $\tau \geq t_1 + d(CW_i) \forall i$ and all received messages have timestamp greater than t_1.

By using guaranteed service mechanism in [7], the attendant node knows for sure that a request message sent from a washing node will never be delayed by the network more than some max_delay. The same configuration is applied and we never find any deadlock. The average allocation waiting delay is about 52 cycles in comparison with that of null message protocol at about always more than 100 cycles for any variation of interval between null messages as in Figure 5(c). The buffer at the attendant node is enough to store all the request packets from car wash nodes.

VII. Distributed DE On Multicore Example: Tunnelling Ball Device

For a more realistic example, we describe a distributed control example that we call Tunnelling Ball Device (TBD). The TBD is a device that controls a spinning disc with two holes on it. Balls are dropped from a tower and the control system needs to vary the speed of the disc such that balls reach the disc right on one of the holes on the disc in order to *tunnel* through the hole. To accomplish this, the TBD has two sensors: one to detect the time the ball starts to fall and another to measure its speed. Using these parameters and the height of the tower, we compute the time that the ball will reach the disc and rotate it to the hole.

The control system consists of three main processes as shown in Figure 8. The *encoding* process receives encoding events from the motor spinning the disc. If the disc rotates a full round, it will produce 200 ticks corresponding to 200 encoding events. The position of the disc is measured by the number of *ticks* the encoding process counts. From the

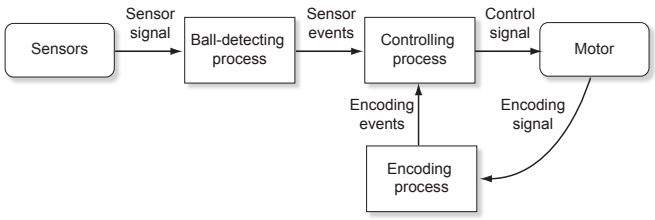


Fig. 8. Computation model of the TBD.

encoding events, it measures the current speed of the disc as well as the position of the hole on the disc. The controlling process uses those parameters to control the spinning speed of the disc so that the hole on the disc will be on the path of the ball when the ball reaches the disc to allow the ball to tunnel through the disc.

The encoding process mainly detects ticks from the encoding sensor of the motor. Each time it detects a tick, it knows that the motor has rotated by some angle, then it modifies its current information about the position of the motor's rotor. Finally, it sends an event with the timestamp at which the tick occurs and the position of the rotor to the controlling process.

The controlling process controls the spinning speed of the motor by simulating the theoretical positions of the motor at each specific time. The theoretical positions of the motor are compared against the real positions at the same time of the motor based on the timestamps and positions of events sent from the encoding processes.

The ball-detecting process checks for signals from sensors that detect a dropping ball. The dropping ball events are then timestamped and forwarded to the controlling process. The controlling process then computes the time the ball will reach the disc to adjust its theoretical spinning speed so that the hole on the disc is aligned with the ball's path. The controlling process then automatically aligns the real position of the disc with the theoretical position.

A. Multicore Mapping

We map the TBD example onto a multicore simulation environment that we developed. We instantiate multiple PRET processor [8] and connect them with our interconnection network-on-chip [7]. Each processes from Figure 8 is mapped onto a separate PRET processor. The most interesting process in the TBD example is the control process. The sketch of the control program of the control process is as follows:

In the Listing 3, the speed of the motor spinning the disc is controlled and varied using deadline instructions. First, we set the theoretical speed of the disc by setting the period between the each iteration increasing the theoretical position of the disc. Then control the speed of motor spinning the disc to make the theoretical position of the disc and the real position of the disc as close as possible. In each iteration, we compare the theoretical position of the disc and the real position of the disc to determine the control value for controlling the motor by a PID control. For example, if we want to increase the speed of the motor, we just need to decrease the period.

Listing 3. Code for the Controlling process

```
initialize();
while(not_terminated) {
    DEADLINE(period);
    //the event that increases the theoretical position
    theorital_disc_pos++;
    //get the timestamp of the event
    theoretical_disc_pos_increase_time = get_time();
    check_msg_from_sensors();
    if(ball_dropped) {
        //adjust the theoretical period (speed)
        period = compute_period();
    }
    //Find  real disc position
    while(1) {
        receive_encoding_events();
        for all encoding events e {
            if(timestamp(e) <=
                theoretical_disc_pos_increase_time) {
                real_disc_pos = value(e);
            } else
                //skip compare with the real disc position
                break;
        }
        //no encoding events before theo. time
        if(get_current_time() >
             theoretical_disc_pos_increase_time +
             max_delay_from_encoding_core)
             break;
    }
    //control real speed motor using theo. time
    control_value = PID(theorital_disc_pos, disc_pos);
    send_motor_control_signal(control_value);
}
```

To find the real position of the disc, we use the PTIDES mechanism to find all the encoding events before we increase the theoretical position of the disc.

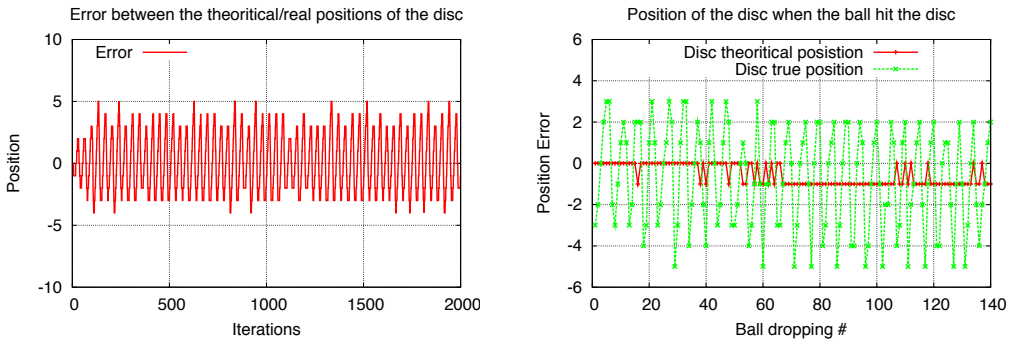
The analysis of the TBD example is similar to that of the one done in Section V-C. In our case, if the disc does not spin faster than 2 rounds per second, then there are at most 400 tick events per second sent by the encoding process. Suppose that the system is running at the speed of 100MHz. Then, the interval between two consecutive encoding events is at least 250000 cycles. Now, suppose that each encoding event is of size e, then $\frac{e((t_1-D_f)-(t_0-D_f))}{X_f(t_0-D_f,t_1-D_f)} \geq 250000$. As $\Delta_f = 0$ is guaranteed by the real-time communication network with jitter control, $X_f(t_0 - D_f, t_1 - D_f) \leq e\frac{t_1-t_0}{250000}$. This means that the receiving node's processing capability is $R_f(t_0, t_1) \geq p\frac{t_1-t_0}{250000} - B_f \Leftrightarrow B_f \geq p\frac{t_1-t_0}{250000} - R_f(t_0, t_1)$

B. Results

The Figure 9(a) shows the error between the theoretical position of the disc and the real position of the disc based on our simulation disc model. Based on that control capability, Figure 9(b) shows the capability of positioning one of the two hole on the disc so that the ball is able to tunnel through the disc. The error of the theoretical position of the disc from the exact position is at most 1 tick. Since the disc rotates a full round, there are 200 ticks, which makes 1 tick equivalent to about 2 degrees. If the control algorithm for the disc is good enough, the real error of the disc is at most 4 ticks or 7 degrees.

VIII. Conclusion

We understand that multicore systems are the future of processor technology. However, deploying real-time control software on CMPs is non-trivial because it demands certain



(a) Theoretical and real disc positions.

(b) Error from exact position of disc using resolution 200.

Fig. 9. Tunnelling ball device example results.

properties from the underling communication network and processing nodes. We identify these properties and present our implementation that consists of multiple PRET processors interconnected with a real-time network-on-chip. Our solution eliminates network jitter and variability in send and receive rates between processing nodes. Unlike the NMP protocol, we employ PTIDES to distribute the DE application onto the CMP. This has the advantage that null messages are no longer required. We are currently investigating an implementation of the presented architecture on an FPGA.

IX. Acknowledgment

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards 0720882 (CSR-EHS: PRET) and 0931843 (ActionWebs), the U. S. Army Research Office (ARO W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI FA9550-06-0312 and AF-TRUST FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

References

[1] J. Misra, "Distributed discrete-event simulation," *ACM Computing Survey*, vol. 18, no. 1, pp. 39–65, 1986.

[2] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transaction on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[3] Y. H. Song and T. M. Pinkston, "On message.dependent deadlocks in multiprocessor/multicomputer systems," in *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*. London, UK: Springer-Verlag, 2000, pp. 345–354.

[4] A. Hansson, K. Goossens, and A. Rădulescu, "Avoiding message-dependent deadlock in network-based systems on chip," *VLSI Design*, vol. 2007, pp. Article ID 95 859, 10 pages, May 2007, hindawi Publishing Corporation.

[5] R. L. Cruz, "A calculus for network delay. i. network elements in isolation," *Information Theory, IEEE Transactions on*, vol. 37, no. 1, pp. 114–131, 1991. [Online]. Available: http://dx.doi.org/10.1109/18.61109

[6] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *in ISCAS*, 2000, pp. 101–104.

[7] D. Bui, A. Pinto, and E. A. Lee, "On-time network on-chip: Analysis and architecture," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-59, May 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-59.html

[8] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2008, pp. 137–146.

[9] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–268.

[10] P. Gratz, C. Kim, R. Mcdonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architectures," in *in International Conference on Computer Design*, 2006, pp. 477–484.

[11] Y. Ho Song and T. M. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 3, pp. 259–275, 2003.

[12] J. Kubiatowicz and A. Agarwal, "Anatomy of a message in the alewife multiprocessor," in *ICS '93: Proceedings of the 7th international conference on Supercomputing*. New York, NY, USA: ACM, 1993, pp. 195–206.

[13] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1992, pp. 256–266.

[14] N. Concer, L. Bononi, M. Soulie, R. Locatelli, and L. P. Carloni, "Ctc: An end-to-end flow control protocol for multi-core systems-on-chip," in *NOCS '09: Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 193–202.

[15] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM, 2001, pp. 684–689.

[16] G. D. Micheli and L. Benini, *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[17] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

[18] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 278–287, 1992.

[19] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," vol. 10, 1995, pp. 1374–1396.

[20] K. Goossens, J. Dielissen, and A. Rădulescu, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, Sept-Oct 2005.

[21] J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler, "Execution strategies for ptides, a programming model for distributed embedded systems," in *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 77–86.