On Distributed Discrete Event Execution on Chip-Multiprocessors

Dai Bui
Hiren Patel
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-148
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-148.html

October 30, 2009

Acknowledgement

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 #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, and Toyota.

On Distributed Discrete Event Execution on Chip-Multiprocessors

Dai Bui[1], Hiren D. Patel[2], Edward A. Lee[1]
[1] University of California, Berkeley
{daib,eal}@eecs.berkeley.edu
[2] University of Waterloo
{hdpatel}@uwaterloo.ca

Abstract

Deploying real-time control systems software on multiprocessors requires distributing tasks on multiple processing elements, and coordinating their executions using a protocol. One potential protocol is the use of the discrete-event (DE) model of computation because it already defines a clear notion of the passage of time, and there is significant existing research on distributing DE. In this paper, we consider a distributed DE with null-message protocol (NMP) on a multicore system for real-time control systems. We illustrate that even with the null-message deadlock avoidance scheme in the protocol, the system may still deadlock due to inter-core message dependencies. We propose a simple analytical model to identify two central reasons for these deadlocks. They are lack of an upper bound on send and receive rates for each processing element, and an unknown upper-bound on network delay. Then, we argue that architecture features such as timing control, timing synchronization and real-time networks-on-chip can be used to prevent message-dependent deadlock. We show that we can replace NMP with a distributed DE strategy called PTIDES that helps ease the process of eliminating this deadlock problem.

1 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) [16, 2]. Specifically, we evaluate a potential message-dependent deadlock [17, 10] 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* [17, 10].

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.

There are two main reasons that contribute to this message-dependent deadlock: 1) lack of an upper-bound on send and receive rates for each processing element, and 2) an unknown upper-bound on network delay. If we can obtain these upper-bounds, then we can allocate enough processing power to process all arrival messages fast enough and buffer space to absorb immediately packets of bursty traffic, thereby guaranteeing that deadlock will never happens. We propose the use of architectural features such as timing control, timing synchronization and real-time NoCs to prevent message-dependent deadlock. In particular, we show that by using timing instructions in software we can enforce a bound on the sending/receiving rates between nodes. For this, we employ a real-time embedded processor architecture that is designed for predictable and repeatable timing behaviors called PRET [14]. When we can enforce a bound on the number of messages sent within an interval, then the number of message arrivals at a receiving node within an interval are also bounded. We propose an analytical model based on the ideas of network calculus [3] and real-time calculus [18] to derive buffer space requirements such that no message has to wait at the node's router. Furthermore, we propose the use of a programming model called PTIDES that will make it easier to avoid deadlocks of a system, but also improves performance over NMP as in the experiments in Section 6.1, in which PTIDES average waiting delay is only haft of the minimum NMP waiting delay.

This paper is organized as follows: in Section 2, we give a brief overview of related materials used in this paper including network on-chip interconnection, discrete even execution, null message protocol and message dependency deadlock. Section 3 explains the fundamental problem causing a multicore system to be deadlocked when the null message protocol is used. We then discuss some deadlock avoidance mechanism in Section 4. The evaluation section, Section 5, demonstrates an evaluation of a deadlock scenario and a performance comparison between the null message protocol and PTIDES.

2 Background

2.1 Distributing Discrete-event with NMP

The discrete-event (DE) execution semantics require that events are processed in timestamp order. An event e is defined as a timestamp and value pair (t, v) where $t \in \mathbb{R}^+$ and $v \in domain(V)$. Events are processed such that for two events e_1, e_2 with timestamps t_1, t_2, event e_1 is processed before e_2 if and only if $t_1 < t_2$.

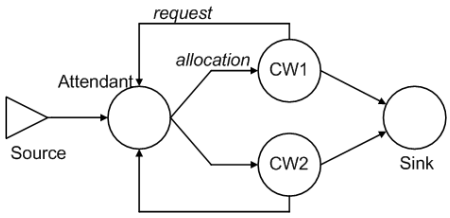


Figure 1. Message flow of the car wash example.

We borrow a *car wash* [2] example to describe a DE system as shown in Figure 1. In this example, there are five processing components. They are a source, an attendant, two car washes denoted by CW_1 and CW_2, and a sink. The source forwards cars to the attendant who dispatches cars to the car washes. This attendant follows the policy that it dispatches a waiting car to the car wash that is idle earliest. A request mesage for another car from the car wash informs the attendant that the car wash is idle. This message contains a timestamp of the physical time at which the car wash became idle. The attendant uses the timestamp to identify which car wash to allocate a waiting car. Once a car wish completes its process, it sends the car to the sink. In doing so, it attaches a message with the timestamp at which the car completed its wash. The sink then orders the cars according to the timestamps at which the cars finished their washes.

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 multiple processor system, however, the processing components are distributed, and each of the processors have their own ordered event queues. Since a processor does not have global knowledge of the events at any point in time,it is difficult to determine when it is safe to process events.

In the case of the car wash example, let us assume that each of 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 mes-

sage protocol (NMP) [16, 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 receive 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.

2.2 Interconnection Network on Chip

2.2.1 Network on Chip

Network on-chip (NoC) is a new design paradigm for System on-chip (SoC) [4, 15]. Network on-chip often uses the wormhole packet switching [5]. A packet is divided in to smaller data unit called flits (flow units) as in Figure 2(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.

2.2.2 Deadlock-free Interconnection Network

An interconnection network used to connect processing elements such as the network on-chip in Figure 5(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 2(b), in which packets create a cyclic loops [7]. In Figure 2(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.

2.3 Message-Dependent Deadlock

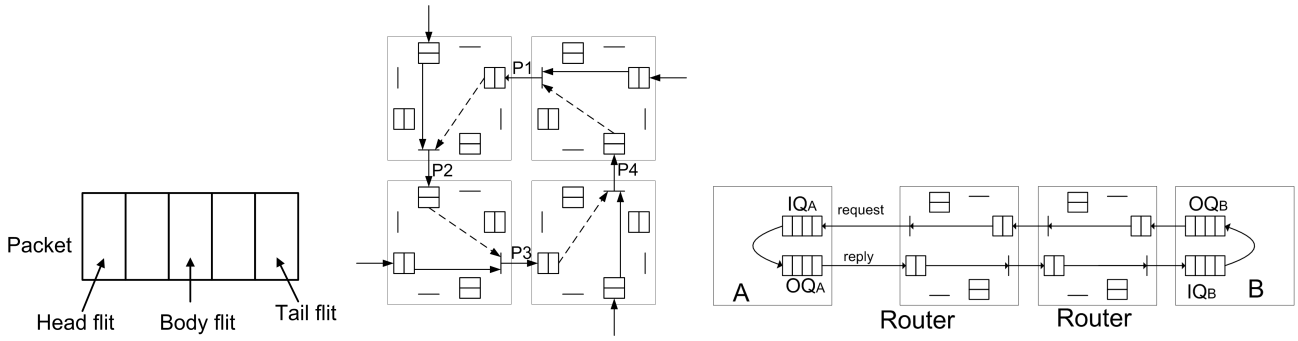
In a multicore system that uses network on-chip interconnection, although the communication network is deadlock free, message-dependent loops created by processing nodes might cause deadlocks in the multicore system [17, 10, 11]. This deadlock is sometimes called *request* $\prec$ *reply* dependency 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 2(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 to output of node A, then input of node A. Then node A cannot send/receive and any packet. At this point, the system becomes deadlocked.

3 Deadlock of Null Message Protocol

3.1 Deadlock Scenario

The car wash example in Figure 1 is vulnerable to message-dependent deadlock 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 4 in [20], 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 2.3. 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.



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

Figure 2. Interconnection network

3.2 Analytical Model

We present an analytical model to clearly describe the factors causing deadlock. Our model is based on the ideas of network calculus [3] and real-time calculus [18]. Let V be the set of nodes in a network and $C \subset \mathbb{N}$ be the set of virtual channel index. The set of flows in the network is defined as $F \subseteq V \times V \times C$. A flow $f = (s, d, c) \in F$ is a path from a source node s to a destination node d with virtual channel index $c \in C$.

For a time instant $t \in \mathbb{R}$, we define upper bound of the traffic transmitted by a source node on flow f as $x_f(t)$, and the traffic arriving at the destination node at the same time instant as $a_f(t)$. A destination node on flow f has a minimum processing capability (amount of traffic it can process) of $r_f(t)$. For a time interval $[t_0, t_1]$ such that $t_0, t_1 \in \mathbb{R}$, the upper bound of the amount of traffic transmitted by the source node is $X_f(t_0, t_1) = \int_{t_0}^{t_1} x_f(t)dt$, and the maximum amount of traffic arriving at the destination node is $A_f(t_0, t_1) = \int_{t_0}^{t_1} a_f(t)dt$. The minimum processing capability of a destination node reserved for flow f is $R_f(t) = \int_{t_0}^{t_1} r(t)dt$. At the destination node, there is buffer space reserved for the flow f, which we denote as B_f.

Congestion occurs when there are more packets arrived than consumed on a flow. We characterize the amount of traffic packets contributing to the congestion on flow f by $N_f(t_0, t_1) = A_f(t_0, t_1) - R_f(t_0, t_1)$. If a flow does not have sufficient buffer space on it, a deadlock due to unconsumed packets may occur. Therefore, we must satisfy condition (1) in order to avoid such deadlocks.

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

Notice that condition (1) is a sufficient and necessary to exclude packet congestion resulting to deadlocks. However, enforcing this condition directly from a programming model is difficult because $A_f(t_0, t_1)$ depends on the properties of the underlying communication infrastructure such as the routing and switching policies.

We can, however, control the transmission rate in a programming model; hence, we need to derive $A_f(t_0, t_1)$ using the transmission rate from a source node. To do this, we require the notion of a network delay, and the interval bounds that result in the maximum number of arrival packets at a destination node. Let D_f be the minimum network delay, and $D_f + \Delta_f$ be the maximum network delay on flow f. Note that Δ_f can be interpreted as the jitter in the network on a flow f.

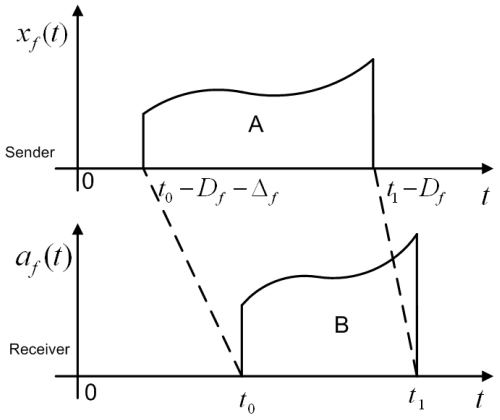


Figure 3. Effect of traffic distortion on send/arrival rates.

Figure 3 shows the relationship between the transmission and arrival of packets for an interval $[t_0, t_1]$. Notice that a packet arriving at the destination node d at t_0 can be sent *latest* from node s at time $t_0 - D_f - \Delta_f$. The traffic arriving at node d at time t_1 can be sent *earliest* at time $t_1 - D_f$. 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 reason A. We can 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), $\forall t_0, t_1 . t_1 \geq t_0 \geq 0$, 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 transmission rate, processing capabilities, and the jitter in the network. Through programming models, it is possible to control the transmission rates, the processing capability, and even allocate sufficient buffer capacity. However, this formulation requires that a bound for the network delay and jitter are determined.

3.3 Identification of Deadlock Factors

We identify two factors that contribute to the deadlock issue: 1) temporary or permanent mismatch between send/receive rates, and 2) bursty traffic caused by message jitter in the packet-switching network.

3.3.1 Mismatch of Send/Receive Rates

It is clear from condition (3) that if sufficient buffer capacity is not allocated, then whenever packets are sent faster than they are processed, the unconsumed packets can overflow the buffers and result in deadlocks. However, only controlling the transmission and processing rates is not safe because the condition (3) also depends on message jitter Δ_f.

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 4 illustrates the message jitter issue [20]. 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. The minimum delay D_f can be easily determined by sending a message in a network without any other traffic. If we can estimate Δ_f, then the upper bound on the sending rate at each source node can be derived from condition (3).

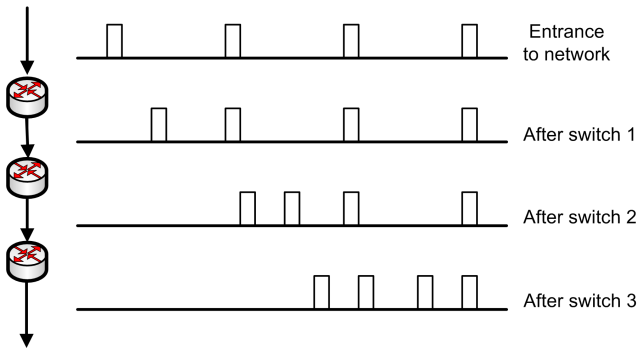


Figure 4. Jitter of messages.

4 Deadlock Avoidance Mechanisms

To build a reliable system, a high-level programming layer should know more about its underlying hardware infrastructure like buffer space, network delay and so on. High-level software programs also should not behave selfishly such as sending as many packets as fast as possible, instead, they should co-ordinate in a *timely manner*. Simply increasing buffer space without explicitly controlling the sending/receiving rates of computation nodes in a multicore system is never safe because bursty traffic might quickly fill up buffers in an interconnection network. In addtion, the varying network delays often contribute to bursty traffic that flood nodes with messages.

Our solution essentially controls and determines the rates and variables in condition(3). There are three components to our solution: on-chip real-time communication network, repeatable transmission rates at the sender, a calculus analytic model based on the ideas of network calculus [3] and real-time calculus [18] for estimating buffer capacity and processing capability. We describe controlling message jitter by using an on-chip real-time communication network, which helps with estimating message jitter Δ_f. We also show that it is necessary to control the sending rate $X_f(t_0,t_1)$ of a source node by using timing instructions of a real-time embedded processor architecture PRET [14]. Finally, from condition (3) we to derive the buffer space and processing capability from the maximum number of packet arrivals within any certain interval at a processing node.

4.1 Guaranteed-service On-chip Communication

In order to guarantee (3), we need to estimate the message jitter Δ_f. The guaranteed-service on-chip communication network such as [1, 8] can guarantee $\Delta_f = 0$. Other real-time communication on-chip can guarantee that Δ_f is smaller than a reasonable finite value. The guaranteed service communication on-chip guarantees that real-time messages will reach their destination within a certain, previously known, bounded amount of time regardless of other traffic on a network on-chip. This guarantee is essential for real-time multicore systems using network on-chip.

In the example from [1], there are three real-time flows on a network as in Figure 5(a) that are routed to share a number of links. The traffic pattern of each flow is characterized by the maximum packet length in flits of each flow and the minimum interval between two successive packets in cycles of each flow. For example, the specifications for the three flows are as follows: $F_1 = (PE_7 \rightarrow PE_{23}, 5\text{flits}, 21\text{cycles})$, $F_2 = (PE_6 \rightarrow PE_3, 3, 19)$ and $F_3 = (PE_5 \rightarrow PE_{19}, 4, 17)$. Based on those characteristics, a suitable path is found in a system to meet the real-time constraint of each flow if that path exists. The packets sent from source nodes of the three flows will reach the destination node within a bounded amount of time regardless of other traffic as in Figure 5(b). The figure shows that it will take the packets of one real-time flow the same amount of time to reach their destination.

4.2 Timing-Control Architecture

We find that the deadline instruction in real-time processor PRET [14] is a useful mechanism to control the transmission rates of a node. In the car wash example, suppose that each car wash node periodically sends null messages to the attendant node, this is called the null message flow, on condition that the interval between null messages is at least 2000 cycles. In conventional architecture, car wash nodes can be programmed as follows:

```
while(notTerminated)
{
    ...
    send(attendantNode, nullMessage);
    i = 0;
    while(i < 100)
    {
         ...
         if(someCondition)
            break;   //this can reduce execution time
         ...
         i++;
     }
     send(attendantNode, nullMessage);
     ...
}
```

In the above program, the interval between null message sends are determined by the execution time of the code segment between those send commands. As the speed of a processor is increased or there are no threads competing for execution with this thread or some execution path takes less time than usual such as the break command, the interval between the two send commands may become smaller. This means that messages will be sent to the receiving node faster than required. 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, potential deadlock might happen. Now we will modify the program to use timing instructions as follows.

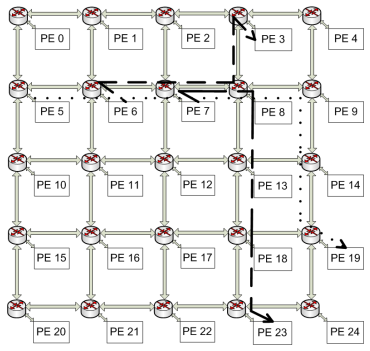
```
while(notTerminated)
{
    ...
    //interval to the next send command
    //is at least 2000 cycles
    DEADLINE(z);
    send(attendantNode, nullMessage);
    i = 0;
    while(i < 100)
    {
         ...
         if(someCondition)
            break;
         ...
         i++;
     }

    //interval to the next send command
    //is at least 2000 cycles
     DEADLINE(z);
     send(attendantNode, nullMessage);
     ...
}
```

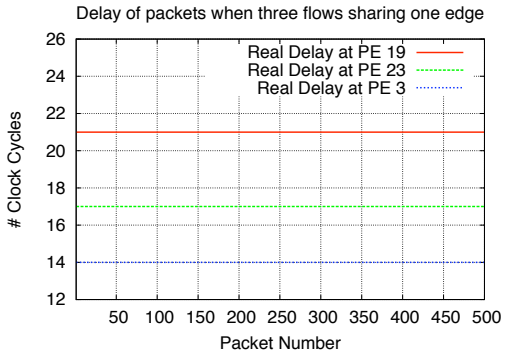
In the above code, we insert two *deadline instructions* with parameter z that converted into 2000 cycles to make sure that the interval between the two null message send commands to the attendant node is never smaller than 2000 cycles regardless of processor speed and/or current processor workload affected by other concurrent threads or the lengths of execution paths of the program. 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 for each car wash node, each null message packet size is p and the interval between null messages $\frac{p((t_1-D_f)-(t_0-D_f))}{\overline{X}_f(t_0-D_f,t_1-D_f)} \geq 2000$ is guaranteed by the deadline instructions and $\Delta_f = 0$ is guaranteed by the real-time communication with jitter control, then $\overline{X}_f(t_0 - D_f, t_1 - D_f) \leq p\frac{t_1-t_0}{2000}$, which means that if the attendant node processing capability $\overline{R}_f(t_0, t_1) \geq p\frac{t_1-t_0}{2000} - B_f \Leftrightarrow B_f \geq p\frac{t_1-t_0}{2000} - \overline{R}_f(t_0, t_1)$, then the sufficient and necessary condition (3) is satisfied.

4.3 PTIDES Execution Strategy

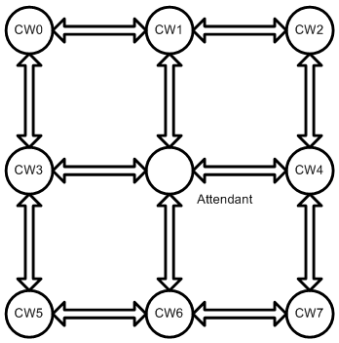
PTIDES makes it easier to satisfy condition (3) since PTIDES does not send any null messages. If we consider



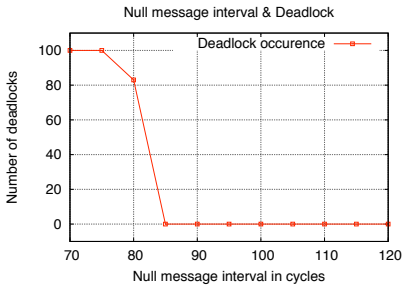
(a) A traffic Scenario.



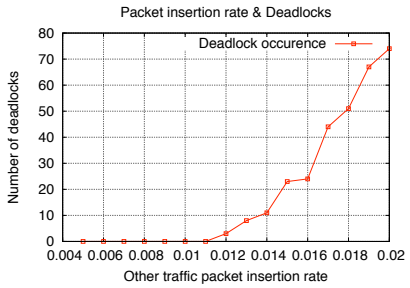
(b) Message delay of the three real-time flows.



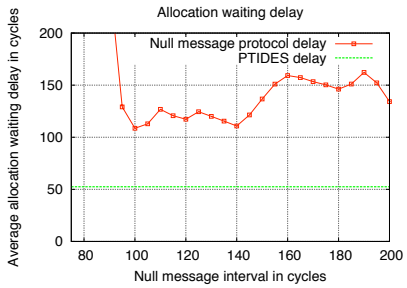
(c) A simple car wash example execution mapped on a multicore system.



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



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



(f) Average allocation delay by two approaches.

Figure 5. Configurations and evaluation results

null messages from each car wash node to the attendant node is a flow and request packets is another flow, then those flows need to satisfy the condition (3). Given that null messages are rather regular, calculating the condition (3) for all the null messages flow might be problematic when the real-time constraints of a system is also taken into account. For PTIDES, the condition (3) is computed only for request flows. Request messages are not sent as regularly as null messages, therefore, calculating the condition (3) with some real-time constraints of a system could be much more easier. As PTIDES does not need null messages, we only need to compute the condition (3) for each request flow from a car wash to the attendant. Suppose that it takes each car wash at least 10000 cycles to wash a car, this can be guaranteed by using PRET, then the interval between two consecutive request messages is at least 10000 cycles. We apply the same procedure as in Section 4.2 to derive that $\overline{R}_{req\_flow}(t_0, t_1) \geq p\frac{t_1-t_0}{10000} - B_{req\_flow}$ to satisfy (3) where p is request packet size.

We will 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 4.1 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 receive 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. The reason for the second condition is that if a car wash CW_2 sends a request message m_2 with timestamp t_2 to the attendant node. At the physical time τ, the message m_2 has not reached the attendant node. This means that $t_2 + d(CW_2) > \tau$. Furthermore, $\tau \geq t_1 + d(CW_i)\forall i$, therefore $t_2 + d(CW_2) > t_1 + d(CW_2) \Rightarrow t_2 > t_1$. Therefore it is safe to allocate a car to CW_1 without violating the DE semantics.

5 Experiments

5.1 Simulation Scenario

To demonstrate the potential deadlock of the NMP on a multicore system, we set up a simple simulation scenario of the car wash example on a network on chip using the Noxim [6] simulator. The scenario of the example is shown in Figure 5(c), 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. This example is a metaphor for a simple load balancing protocol.

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.

5.2 Simulation with the Null Message Protocol

5.2.1 Simulation specification

While the basic simulation steps of the example are as in the above section, the null message protocol to avoid deadlock is as follows: 1) A car wash node periodically sends *null* messages to the attendant node to update its current time to the attendant node via the timestamps in the messages. 2) The attendant node, whenever it receives a null message, updates its knowledge of the current time of the node that sends the null message.

It is also mandatory that the messages sent from a washing node to the center node be received in order. Those messages are routed using XY routing, so-called Dimension Order Routing (DOR) [7]. In our simulation, the network is also lightly loaded with other traffic using DyAd adaptive routing [12] with packet insertion rate at 0.005. The simulation specification is as in Table 1[1].

[1]We choose to use simulation parameters with small cycles to facilitate the process of simulating. Those parameters can be scaled to fit with real applications

	Value
Time to process a message (null, request)	10 cycles
Interval between null messages	80 cycles
Time to wash a car	301 cycles
Buffer size (at attendant node)	40 flits
Message size	2 flits
Number of cars to wash	200

Table 1. Simulation specification

5.2.2 Deadlock Characterization

First, we would like to define deadlock in our simulation experiments.

Definition 1 *A node is blocked when both its input queue and output queue are full. A system is deadlocked when all the nodes are blocked.*

With the specification in Table 1, deadlock happens frequently 70 times in 100 runs before 200 cars are washed. When we increase the interval between null messages sent by a washing node to 82 cycles, deadlock happens less frequently. If the interval is more than 85 cycles, deadlock does not occur in our simulation because the condition (3) is satisfied. However, we consider the situation when the update interval is kept at 85 cycles, but the attendant node cannot process a packet in 10 cycles anymore due to some cache misses or some increased workload, its time to process a packet is increased by 10% to 11 cycles. Immediately, the deadlock happens frequently again in our simulation.

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 [14] that does not allow some work (sending message) to be done faster than needed as in Section 4.2.

Figure 5(d) 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).

5.2.3 Effect of other network traffic on deadlock

In this section, we will evaluate the effect of other network traffic on the deadlock. Figure 5(e) shows the effect of increasing other traffic load in the network on deadlock occurrence of NMP. The simulation scenario is as follows: 1) The interval between null messages is 85 cycles. 2) The packet insertion rate of other traffic is gradually increased from 0.005 to 0.02. 3) Other parameters stay the same.

For each packet insertion rate, we simulate 100 times to get the number of deadlocks. As we can see, when we increase the packet insertion rate of other traffic, deadlock cases might grow from never to frequently. So in a more complex network with multiple $request \prec reply$ fashion transactions or during bursty memory traffic, deadlock might happens easily.

5.3 Simulation with PTIDES Strategy

For PTIDES, instead of waiting for null messages from washing nodes to the attendant node, the attendant node uses the passage of real-time. By using guaranteed service mechanism in [1], 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. Then the mechanism as described in Section 4.3 is applied. 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. The buffer at the attendant node is enough to store all the request packets from car wash nodes.

6 Related Work

There are some common approaches used in current CMP architecture with network-on-chips to avoid deadlock. They involve increasing the number of virtual channels, buffer sizes [9], or by using a deadlock resolution mechanisms [11]. However, simply increasing the buffer size and number of virtual channels without considering the send/receive rates is never safe since if a node continuously sends more packets than another node can receive, unconsumed packets will fill up buffers gradually and form a deadlock. 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 clock. It might also slow down the clock of a system [9] since deadlock detection and resolution logic is added to router pipeline stages.

Although message-dependent deadlock happen *infrequently*, for safety-critical control systems, this kind of deadlock has to be excluded completely. In [13], whenever deadlock happens and 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 not safe and is especially unsuitable for real-time systems. In the active message communication model [19], 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.

6.1 Comparison

Figure 5(f) shows the performance of the two approaches in terms of waiting delay, which is defined as the interval from when a request message is sent until the attendant node decides to allocate a car to the request node[2]. We choose this metric because it is independent from washing time. From that figure, we can also see that decreasing the interval between null messages does not necessarily mean smaller waiting delay. We can see that PTIDES execution scheme provides better waiting delay. Sending fewer packets can also reduce the power consumption of a system.

7 Conclusion

We have shown that the NMP might cause a potential deadlock in a multicore system if it is not implemented carefully. We also show that in order to exclude message-dependent deadlocks in a multicore system, a computation node needs to know the bound of the number of message arrivals within an interval of time to allocate enough buffer and processing power to process those messages, ensuring that all messages are consumed right after they have arrived. This means that sending nodes should not send messages too fast. This can be supported using PRET architecture.

PTIDES execution strategy is a suitable replacement for NMP since it can exploit the inherent time synchronization in a multicore system. PTIDES execution strategy avoids sending many null messages, which means that less power is consumed and potential deadlocks due to null messages are avoided. We expect that our techniques will not only

[2]Please note that the null message interval is for NMP only, PTIDES does not use null messages

solve the deadlock issue but also provide a better real-time guarantee.

References

[1] Dai Bui, Alessandro Pinto, and Edward A. Lee. On-time network on-chip: Analysis and architecture. Technical Report UCB/EECS-2009-59, EECS Department, University of California, Berkeley, May 2009.

[2] K. Mani. Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transaction on Software Engineering*, SE-5(5):440–452, 1979.

[3] R. L. Cruz. A calculus for network delay. i. network elements in isolation. *Information Theory, IEEE Transactions on*, 37(1):114–131, 1991.

[4] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM.

[5] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[6] Fabrizio Fazzino, Maurizio Palesi, and Davide Patti. Noxim: Network-on-chip simulator.

[7] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. *SIGARCH Comput. Archit. News*, 20(2):278–287, 1992.

[8] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, Sept-Oct 2005.

[9] Paul Gratz, Changkyu Kim, Robert Mcdonald, Stephen W. Keckler, and Doug Burger. Implementation and evaluation of on-chip network architectures. In *in International Conference on Computer Design*, pages 477–484, 2006.

[10] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007:Article ID 95859, 10 pages, May 2007. Hindawi Publishing Corporation.

[11] Yong Ho Song and Timothy Mark Pinkston. A progressive approach to handling message-dependent deadlock in parallel computer systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):259–275, 2003.

[12] Jingcao Hu and Radu Marculescu. Dyad: smart routing for networks-on-chip. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 260–263, New York, NY, USA, 2004. ACM.

[13] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the alewife multiprocessor. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 195–206, New York, NY, USA, 1993. ACM.

[14] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward 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*, pages 137–146, New York, NY, USA, 2008. ACM.

[15] Giovanni De Micheli and Luca Benini. *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[16] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Survey*, 18(1):39–65, 1986.

[17] Yong Ho Song and Timothy Mark Pinkston. On message.dependent deadlocks in multiprocessor/multicomputer systems. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 345–354, London, UK, 2000. Springer-Verlag.

[18] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *in ISCAS*, pages 101–104, 2000.

[19] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[20] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. volume 10, pages 1374–1396, 1995.

[21] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia 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*, pages 77–86, Washington, DC, USA, 2009. IEEE Computer Society.