Adaptive Communication Mechanism for Accelerating MPI Functions in NoC-Based Multicore Processors

LIBO HUANG, ZHIYING WANG, NONG XIAO, YONGWEN WANG, and QIANG DOU,
National University of Defense Technology

Multicore designs have emerged as the dominant organization for future high-performance microprocessors. Communication in such designs is often enabled by Networks-on-Chip (NoCs). A new trend in such architectures is to fit a Message Passing Interface (MPI) programming model on NoCs to achieve optimal parallel application performance. A key issue in designing MPI over NoCs is communication protocol, which has not been explored in previous research.

This article advocates a hardware-supported communication mechanism using a protocol-adaptive approach to adjust to varying NoC configurations (e.g., number of buffers) and workload behavior (e.g., number of messages). We propose the *ADaptive Communication Mechanism* (ADCM), a hybrid protocol that involves behavior similar to buffered communication when sufficient buffer is available in the receiver to that similar to a synchronous protocol when buffers in the receiver are limited. ADCM adapts dynamically by deciding communication protocol on a per-request basis using a local estimate of recent buffer utilization. ADCM attempts to combine both the advantages of buffered and synchronous communication modes to achieve enhanced throughput and performance. Simulations of various workloads show that the proposed communication mechanism can be effectively used in future NoC designs.

Categories and Subject Descriptors: C.2 [Computer Systems Organization]: Computer Communication Networks

General Terms: Design, Performance

Additional Key Words and Phrases: Multicore, NoC, MPI, adaptive, communication mechanism

ACM Reference Format:

Huang, L., Wang, Z., Xiao, N., Wang, Y., and Dou, Q. 2013. Adaptive communication mechanism for accelerating MPI functions in NoC-based multicore processors. *ACM Trans. Architect. Code Optim.* 10, 3, Article 18 (September 2013), 25 pages.

DOI: <http://dx.doi.org/10.1145/2512434>

1. INTRODUCTION

Multicore architectures with Network-on-Chip (NoC) connecting cores have been pervasively recognized as the de facto design for the efficient utilization of the ever-increasing density of transistors on a chip. Recent proposals including the 64-core Tile Processor from Tilera [Wentzlaff 2007], Intel's 80-core Terascale chip [Mattson et al. 2008],

This work is supported in part by National Core-High-Base Major Project of China under grant no. 2013ZX01028-001-002, National Natural Science Foundation of China under grant no. 61070037, no. 61025009, no. 61103016, and no. 61170045, National High Technology Research and Development Program of China under grant no. 2012AA010905, Doctoral Fund of Ministry of Education of China under grant no. 20114307120010, and Hunan Provincial Natural Science Foundation of China under no. 12JJ4070.

Authors' addresses: L. Huang (corresponding author), Z. Wang, N. Xiao, Y. Wang, and Q. Dou, State Key Laboratory of High Performance Computing and School of Computer, National University of Defense Technology, Changsha, Hunan Province, China; email: libohuang@nudt.edu.cn.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/09-ART18 \$15.00

DOI: <http://dx.doi.org/10.1145/2512434>

Arteris' NoC interconnect IPs [Arteris 2011], and NXP-Philips' AEtheral NoC [Goossens 2003] have successfully demonstrated the potential effectiveness of NoC designs.

As the number of cores increases, message passing multicore architectures are introduced as an effective way to eliminate the "coherency wall" between cores existing in conventional shared memory architectures. To reduce the gap between the programming model and the underlying NoC-based hardware, a new NoC design that incorporates message passing parallel programming models, such as MPI [Gropp et al. 1999], is regarded as a promising option. Numerous applications have been ported to or developed for the MPI standard, thus making performance optimization a necessity for multicore architectures [Murillo 2009]. Software overhead constitutes a very large percentage of message latency. This issue will increase in severity when high-speed on-chip parallel communication channels are used to transmit messages. To accelerate the software processing time, the hardware support features of NoC designs require further exploration.

To provide efficient support for MPI to boost the performance of parallel applications, we exploit the on-chip hardware available in NoC-based multicore architectures. Existing solutions [Mahr et al. 2008; Psota and Agarwal 2008; Saldaña and Chow 2006; Williams et al. 2006; Ohara et al. 2006; Verari Systems 2012] mainly focus on MPI adaption on NoC-based multicore processors such as minimal MPI function selection and MPI software stack modification. However, special techniques that can take advantage of the fine-grained, low-latency features of NoC hardware are not fully exploited. In this article, we focus on the special communication protocol for the further performance optimization of MPI functionality.

One of the key functionalities of hardware implementation is the support of MPI communication protocols. Synchronous and buffered communication protocols are the two dominant classes of communication protocols for message passing. In a synchronous system, sender and receiver block execution of the program until the information transfer is complete. In a buffered system, the sender can complete the transmission without the corresponding receive operation being executed. Thus, buffered protocols can achieve lower latencies than synchronous protocols on data transmission by eliminating acknowledgement delay.

By buffering to avoid acknowledgement delay for subsequent data transmission, the buffered protocol outperforms the synchronous protocol when buffers are plentiful, but the latter outperforms the former when buffers are limited, such that the retry mechanism is activated. Designing a single communication protocol to provide high performance for numerous system configurations and workloads is difficult. We advocate an adaptive approach to address this challenge. An adaptive scheme is desirable for two reasons. First, considering the trend toward NoC-based multicore processors, a single protocol must suffice for multiple hardware configurations and applications. Second, statically choosing between a buffered protocol and a synchronous protocol is undesirable because of the varying behavior of different workloads and the time-varying behavior within a workload. Further, a given workload's demand on buffer size varies dynamically over time.

Thus, an adaptive hybrid protocol that provides robust performance is preferable to a static selection of either a buffered or synchronous protocol. Our contribution is an adaptive communication mechanism that performs buffered communication if buffers are plentiful but performs synchronous communication if buffers are limited. Our protocol, adaptive communication mechanism (ADCM), adapts dynamically by determining the communication protocol on a per-request basis to provide robust performance. ADCM attempts to combine both the advantages of buffered and synchronous communication modes and achieves better throughput and performance. Simulations of

various workloads show that the proposed communication mechanism can be effectively used in future NoC designs.

The remainder of this article is organized as follows: Section 2 presents the background of this work. Section 3 describes the general architecture of ADCM, whereas Section 4 provides its evaluation results. Section 5 discusses related work, and Section 6 concludes this article.

2. BACKGROUND

Before discussing ADCM in detail, we describe its background. We first briefly discuss the reason behind MPI adaption into NoC-based processor designs and then present two dominant classes of communication protocols, namely, the buffered and synchronous protocols. Finally, we present the limitations of existing protocols.

2.1. Message Passing over NoC-Based Systems

The challenge of effectively connecting and programming numerous cores for an NoC-based system has received significant attention from both academia and industry. A natural choice is a cache-coherent shared memory design based on previous symmetric multiprocessor architectures. These multicore processors likely have small private L1 or L2 caches but share a large last-level cache that is kept coherent with all L1 caches. However, as the number of cores increases, the protocol overhead would rapidly grow, leading to a “coherency wall” beyond which the overhead exceeds the value of adding cores [Kumar et al. 2011]. To resolve this problem, message passing multicore architectures are introduced to eliminate cache coherence between cores.

Message passing typically makes the most sense for architectures that do not have a shared memory system, with each core having its own main memory. However, the inherent distributed and scalable nature of the NoC communication backbone and the heterogeneity of computation elements make message passing an a priori model as the number of cores increases. If the cache system does not maintain its coherency, the NoC-based system, regardless of whether it employs a shared memory, can eliminate the coherence protocol overhead while facilitating highly efficient communication between cores through message passing. Such explicit parallel programming can be implemented as MPI [Gropp et al. 1999], which is known to be portable and extensible. It has numerous tools and parallel legacy codes to facilitate the use of MPI. MPI implementations exist for most high-performance parallel computer systems, with MPICH [2012] and LAM MPI [2013] being two of the most popular. The current MPI standard is large, containing over 200 function calls. However, only several functions are essential to code a parallel application, with others facilitating its programming.

A large number of studies have attempted to fit the MPI programming model upon multicore architectures. Intel has recently released an experimental processor, called Single-Chip Cloud computer (SCC) [Howard 2010; Mattson et al. 2010; Petrović et al. 2012]. The 48-core SCC explores the message passing model, which provides an on-chip low-latency memory buffer called the Message Passing Buffer (MPB), which is physically distributed across the tiles. Each core has a small MPB (8KB), L1 and L2 caches, as well as off-chip private memories. No hardware cache coherence exists for L1 and L2 caches. In addition, the off-chip memory is physically shared, thus enabling the provision of portions of shared memory by changing the configuration. To embed MPI for NoC-based systems, a working subset of standard MPI functions should be reasonably selected, and a porting process that facilitates the use of MPI functions should be established by modifying the lower software layers of each selected MPI function. This requirement has given rise to numerous studies on MPI adaption, including ocMPI, which is a lightweight implementation of an MPSoC message passing interface [Jaume 2009], SoC-MPI library implemented on Xilinx Virtex FPGA [Mahr

Fig. 1. Basic MPI communication protocols.

et al. 2008], rMPI targeting embedded systems using MIT Raw processor [Psota and Agarwal 2008], TMD-MPI focusing on parallel programming of multicore multi-FPGA systems [Saldaña and Chow 2006], MPI communication layer for a reconfigurable cluster-on-chip architecture [Williams et al. 2006], and Lightweight MPI (LMPI) for embedded heterogeneous systems [Agbaria et al. 2006]. Previous studies have successfully demonstrated the effectiveness of adapting MPI into NoC-based multicore processors. This article is also based on the assumption that MPI is a potential programming model candidate for future multicore processors.

2.2. Communication Protocols in MPI

In this subsection, we describe two basic communication protocols, known as synchronous and buffered protocols in MPI, which will also serve as the base cases. Figures 1(a) and (b) illustrate the process of the two basic communication protocols separately. The buffered protocol is asynchronous because it enables the completion of a send operation without the corresponding receive operation being executed. Thus, the overlap of communication and computation is possible. However, this protocol assumes sufficient memory at the receiver to store all the expected or unexpected messages, which could be on the order of kilobytes or even megabytes depending on the size of messages; otherwise, buffer overflows will occur. The allocation of substantial memory for buffering may lead to wasted memory in cases where the buffer is underutilized. Thus, the programmer must avoid the corruption of the data, given that data transmission occurs in the background, and the transmission buffer can be overwritten by further computation. In an embedded system with limited resources, this protocol may not scale well.

By contrast, the synchronous protocol requires the producer first to initiate a request to the receiver. This request is called the message envelope and includes the details of the message to be transmitted. When the receiver is ready, it will reply with a clear-to-send packet to the producer. Once the producer receives the clear-to-send packet, the actual transfer of data will begin. This protocol incurs a higher message overhead compared with the buffered protocol because of the synchronization process. However, less memory is required and buffer overflows are less likely to occur because the protocol only has to store message envelopes, which are eight bytes long, in the event of unexpected messages.

2.3. Existing Problems

The buffered protocol is generally found to outperform the synchronous protocol when receiving buffers are sufficient. However, using a single buffered protocol alone would result in numerous problems. Before discussing these problems, we assume that the

Fig. 2. Correctness problems.

on-chip network is adequately robust to perform any message transmission operation to the target receivers correctly. That is, we do not need to receive the acknowledgement message to ensure correct NoC transmission. This condition is also the case for real NoC-based processor designs, which differ from large multiprocessor systems.

2.3.1. Correctness Problems. For MPI applications, messages out of order or unexpected are common [Goudy 2005]. The unexpected messages would have to be buffered, and a large number of unexpected small messages or a small number of large messages may result in buffer overflows. Such buffer overflow would give rise to a correctness problem because the transmission data are not received correctly. Figure 2 illustrates this correctness problem. In NoC-based systems with limited memory space, buffering messages would be inadequate because it would limit the scalability of NoC designs. Thus, a programmer must prevent corruption of the data. The incorrect use of the buffered protocol would result in incorrect MPI execution, which poses a programming burden. To mitigate this problem, the retry mechanism could be used. When the receiver does not have sufficient buffer to store the unexpected messages, the retry mechanism would be triggered to resend the lost messages.

2.3.2. Retry Problems. Although the retry mechanism can maintain the correctness of the buffered protocol, the utilization of the retry mechanism in the data transmission remains faced with a number of problems. First, we need to know when to trigger the retry operation. Different ways of triggering the retry mechanism would yield different performance results. Figure 3 illustrates three different timing methods for the retry operation. The first method, as shown in Figure 3(a), is to trigger the retry procedure after all the data are received, even if these data are discarded. This method is simple but may delay the transmission time when a receiving buffer is already available. The second method, as shown in Figure 3(b), is to trigger the retry procedure immediately when buffers overflow. This method is better than the first because it can send the retry message in advance. The third method, as shown in Figure 3(c), is to trigger the retry procedure right after the first packet of the message is received when the buffer is insufficient. This method appears to be an optimal approach that can trigger the retry procedure as soon as possible. However, this method may introduce additional data transmissions when buffers are not ready. The last method, as shown in Figure 3(d), is to trigger the retry procedure only when the receiver has sufficient buffers to complete the data transmission.

Determining what to retry is also an important problem. Different degrees of data granularity for retry also yield different performance results. Figure 4 illustrates three

Fig. 3. Retry problems: timing exploration.

Fig. 4. Retry problems: data granularity exploration.

basic methods for retry. The first method, as shown in Figure 4(a), is called *retry whole*, which sends the retry packet to resend the whole message. Using this method, the receiver does not need to record any data received from previous data transmission. However, this method introduces additional data transmission. The second method is called *retry following*, as shown in Figure 4(b), which sends the retry packet only for data that have not been received because of the lack of buffers. That is, part of the data can be received when the receiving buffers are available. This method utilizes the available network bandwidth and receiving buffer more efficiently than the first method. The last method is called *retry discarded*, as shown in Figure 4(c), which sends only the data that were not received, thus further reducing the size of data to be transferred. This method slightly differs from the *retry following* method in that it requires the receiver to buffer any message data once the buffer is available.

These methods have their own advantages and disadvantages. To achieve an optimized retry mechanism, further exploration of these methods is required. In this article, we propose that these timing exploration methods be combined with data granularity exploration to establish an optimized approach.

2.3.3. Performance Problems. The buffered protocol does not always outperform the synchronous protocol. In some cases, the buffered protocol is slower than the synchronous protocol. Figure 5 illustrates such a case, where the buffers are not ready to receive messages immediately. After three attempts at message transmission, the sender successfully sends its data to the receiver. However, the synchronous protocol only needs send data once, after which the receiver can trigger the retry procedure immediately when buffers are available. The buffered protocol has two other disadvantages. First, this protocol may introduce a large amount of network traffic, which would increase

Fig. 5. Performance problems.

Fig. 6. Ideal communication protocol that can optimally address different cases.

network contention and consequently result in performance degradation. Second, the buffered protocol should send the confirmation message to the sender to release the sending buffer after activation of the retry mechanism. This process may also degrade performance because of the additional message transmission and delayed sending buffer release.

2.4. Motivations

After considering the aforementioned problems, we attempt to establish an ideal communication protocol that can address these issues. Figure 6 illustrates the protocol that ideally has prior knowledge of the buffer usage and is capable of performing the corresponding operations. The figure has three main timelines: the first is D_0 arriving at time AT_{D_0} , which is the arriving time of first communication data D_0 at the target node; the second is D_n arriving at time AT_{D_n} , which is the arriving time of the last communication data D_n at the target node; and the last is $D_0(\text{retry})$ arriving at time $AT_{D_0(R)}$, which is the arriving time of the first communication data D_0 for the first retry operation at the target node. These three timelines will help determine the appropriate operation for different cases.

Figure 6(a) shows an ideal case where the receiving buffer is sufficiently large ($n+1$ free buffers are available before AT_{D_0}). We simply use the buffered communication protocol to deal with this transaction. Figure 6(b) shows a case where the receiving

buffer is limited ($n + 1$ free buffers are only available before AT_{D_n}). That is, the receiving buffer is not ready to receive the data, but will be ready soon. In such cases, the sender can initially send the data using the buffered protocol, such that the receiver will receive the data using available receiving buffers until no buffers are left. Once buffers become available, the data will be received. After $n + 1$ buffers are ready, the receiver will trigger the retry mechanism for discarded data using the retry discarded method illustrated in Figure 4(c). Given that this process can ensure that the data will be successfully received, the *Confirm* message is unnecessary. This approach outperforms the two basic protocols and will have less network traffic than the buffered protocol. Figure 6(c) shows a case with a more limited number of free buffers wherein only $n + 1$ free buffers are available between AT_{D_n} and $AT_{D_0(R)}$. In such cases, we use the retry following method. That is, the receiver triggers the retry operation immediately after receiving data D_0 for the following $n + 1 - k$ data. Compared with the retry discarded method, this approach may send some data twice. However, this method is capable of immediately triggering the retry mechanism. Figure 6(d) shows the last case where the free buffers are limited and will be ready only after $AT_{D_0(R)}$. A simple solution for this issue is to use the synchronous protocol. Although the buffered protocol may achieve better performance than the synchronous protocol in some cases, it may introduce a considerable amount of network traffic that will likely degrade overall performance.

The proposed ideal protocol improves the two basic protocols in terms of performance and network traffic. However, it is based on the knowledge of accurate current and future buffer usage information. This requirement is impossible for practical implementation but sets a goal that we can attempt to achieve. In the following sections, we will introduce ADCM, which enables the hardware implementation of the proposed ideal protocol with a number of simplifications.

3. ADAPTIVE COMMUNICATION MECHANISM

3.1. Goal and Approach

The goal of ADCM is to minimize average data transmission latency. Given an infinite buffer, using the buffered protocol would achieve this goal by eliminating all waiting time for receiver acknowledgement. However, a finite buffer may result in buffer overflow and retry delays that outweigh the benefit of eliminating waiting time. Nevertheless, retry delay only dominates when the receiving buffer on the target NoC node is highly occupied. The mechanism we propose for ADCM uses feedback to keep the buffer utilization below a critical level and thus mitigate retry delays. Our mechanism uses a combined local-remote estimate of buffer utilization to keep utilization below a prespecified threshold by dynamically adjusting the probability of buffered transmission.

Our adaptive communication implementation uses a simple mechanism to estimate the buffer utilization and correspondingly adjust the communication protocol. The mechanism can be discussed from two aspects: sender and receiver. The sender operations comprise three parts: (1) estimating buffer utilization for the target receiving node, (2) determining whether to trigger buffered communication, and (3) reacting according to the messages from the receiving node.

First, the sender uses the information on buffer utilization along with the receiving message as a local estimate of target buffer utilization. Although this static and somewhat obsolete information does not capture future buffer utilization, it is easy to obtain and correlates strongly with future buffer utilization partly because of the buffered nature of the requests that are most likely to cause buffer occupation. Each core uses a simple, signed, saturating utilization counter to calculate whether the buffer utilization is above or below a static threshold. When the counter is sampled, a positive

value indicates that the buffers used are more than the threshold, and a negative value means that the buffer used less than the threshold. The counter is initially reset to the maximum value. Second, a generated message is transmitted through the buffered or synchronous protocol with a probability proportional to the policy counter. The node sends the messages using the synchronous protocol if the policy counter is smaller than the random number; otherwise, the node uses the buffered protocol. Finally, the sender would react according to the feedback message. If buffered communication packets are generated, then a message for a retry operation may be received until a confirmation message is received. If synchronous communication packets are generated, then the data are sent after an answer packet is received.

The receiver operations comprise two parts: (1) estimating own buffer utilization and (2) performing the corresponding receiving operations. This adaptivity is reflected on the time of triggering the retry mechanism and the granularity of data retry. First, the receiver estimates the buffer utilization according to the number of free buffers and future trend of utilization. Second, the receiver generates a message according to different timings and level of data granularity. Given that the time of this retry can be regarded as the answer of the target node, the sender may not need to wait for the confirmation message.

Although ADCM is broadly inspired by the credit-based flow control often used in NoCs, solely using the sender-to-receiver credit counts at a large granularity (perhaps 1KB per credit) to manage flow control in MPI communication is difficult because the MPI communication information, such as message size, is transparent to the NoC credit-based flow, which only knows the packet size. Thus such flow control logic would be implemented in an upper layer such as an MPI Engine (ME), as described in the following section.

3.2. Baseline MPI-Accelerated NoC Design

This subsection describes the baseline communication architecture of processors. This design aims at accelerating the processing performance of MPI primitives in future massive multicore architectures by way of underlying hardware support. These multicore architectures usually present a mesh-type interconnect fabric. No hardware cache coherence exists for the L1 and L2 caches, similar to the Intel SCC processor. A number of factors have to be considered in the performance improvement of MPI primitives through hardware support. The basic design discussed in this article includes two main hardware techniques for the acceleration of MPI primitives: NoC design and the ME [Saldaña and Chow 2006]. Figure 7 shows the block diagram of the baseline implementation architecture with 4×4 2D mesh topology. The underlying NoC design is the actual media used to transfer messages, which can be designed with consideration for MPI communication. Each node also has an ME between the core and Network Interface (NI), which is used to execute corresponding instructions for MPI primitives.

By directly executing the MPI primitives and interrupt service routines, ME reduces the context switching overhead in the cores and can accelerate software processing. The engine also performs message buffer management for the cores as well as fast buffer copying. This engine transfers messages to and from dynamically allocated message buffers in the memory to avoid buffer copying between system and user buffers. This process also eliminates the need for the sending process to wait for the release of the message buffer by the communication channel. The engine also reserves a set of buffers for the incoming messages. Using the aforesaid methodologies, the long message transmission protocol can be simplified, consequently reducing transmission latency.

The ME architecture is shown in Figure 8. This design provides hardware support to address the communication protocol used in MPI implementation. Primary functionalities include serving as a middle layer between the processor core and the interface

Fig. 7. Block diagram of baseline communication architecture.

Fig. 8. Block diagram of MPI engine architecture.

of NoC. The ME will receive the message send requests from the PE core and handle various messages from other processor cores. Two sources can trigger ME to work: the local processor core and the Network Interface (NI). The local processor core may request the ME to perform MPI primitive functions, such that the associated communication data are transferred through this interface. Another source is NI, which may request ME to receive messages from the on-chip network and perform corresponding operations for handling the received messages.

ME generally comprises three key components: PreProcessing Unit (PPU), Parameter Registers (PR), and MPI Processing Unit (MPU). PPU is used to translate the instructions from the processor core into control signals and to generate the message passing parameters used for data transfer that are temporarily stored in PR. Another important task of PPU is to exchange data with the CPU cache for reading or writing communication data. PR includes several registers in the ME. When MPI functions are performed, ME first receives the message parameters from PE core and then updates these registers. MPU is the key component of ME that performs the actual operations for MPI primitive functions, as shown in Figure 9(a). The execution of these functions is generally performed in two separate pipelines: the send and receive pipelines. The send pipeline is used for active operations such as *MPI_Send*. The receive pipeline is used for passive operations such as *MPI_Receive*.

(a) conventional MPI processing unit

(b) modified MPI processing unit for ADCM

Fig. 9. Block diagram of MPI processing unit.

3.3. ADCM Architectural Support

In this subsection, we describe the proposed ADCM architectural support based on the MPI-accelerated NoC designs.

3.3.1. ADCM Hardware. To support the adaptive communication protocol, hardware modification primarily can be applied to the MPU. The conventional MPU is capable of handling unexpected messages and dividing large messages into smaller size packets, as shown in Figure 9(a). The MPU generally comprises five main components organized in two separate pipelines: message packetizing and building units for the send pipeline as well as packet reception, expectation, and response units for the receive pipeline.

The MPU also has an internal buffer to store unexpected messages. Thus, sending and receiving can be performed simultaneously. Figure 9(b) shows the block diagram of the proposed modified MPU. Two small logical units with associated registers are added into MPU, which are shaded in the figure. For the sending pipeline, the Sending Policy Unit (SPU) is added. To support the sending policy generation, a Receiving Buffer Credit (RBC) table is also added. The SPU generates the selection results according to the RBC table. The SPU comprises two subcomponents: the sending policy generator, which implements the adaptive sending protocol algorithm (buffered or synchronous protocol) and the preparation unit, which generates corresponding header data according to the selection result and then sends these data to the packetizing unit. The RBC table is the basis for the selection of different protocols. This table includes two fields for each target NoC node: *Counter* and *UD*. The 2-bit *UD* field is used to indicate the trend of the receiving buffer number, such that “00” indicates that the buffer number is unchanged, “01” indicates that the buffer number is increased, and “10” indicates that the buffer number decreased. The *UD* field is an important parameter for protocol selection because it can predict available buffers more accurately. The 8-bit *Counter* field stores the value of the receiving buffer number available for other nodes, and their initial values are set as maximum. The update operation of the RBC value is triggered by the response packet from the corresponding node. The upper 2 bit can be used to specify the buffer number in different angularities, that is, “00” for 0KB to 1KB, “01” for 1KB to 4KB, “10” for 4KB to 8KB, and “11” for 8KB to 32KB. The last 6 bits are used to indicate the buffer number in that granularity. This process is sufficiently accurate for storing the buffer number information and for reducing the hardware overhead of the RBC table. Each core would have an RBC for every possible NoC node. The value of *Counter* is directly calculated from the received value for the send operation, and the value of *UD* is calculated from records of previous buffer number changes. These two parameters serve as the available information for the estimation algorithm of buffer usage.

For the receiving pipeline, the Receiving Policy Unit (RPU) is added. The RPU also includes two subcomponents: the receiving policy generator, which implements the adaptive receiving protocol algorithm according to the receiving packets and local receiving buffers; and the preparation unit, which generates corresponding header data according to the receiving policy and sends these data to the packetizing unit. These header data would specify the type of the response packets, the RBC data, and so on. To access the buffer utilization information, a local receiving buffer credit register is configured. This register records the number of free buffers and the utilization trend of buffers, the fields of which have the same meaning as those of SPU. The *UD* is set according to the comparison result of the current number and previous stored number of free receiving buffers. Notably, the receiving buffers are counted as an unexpected queue, as shown in Figure 9. Evidently, the expected packets can be handled promptly by the processor core and would not occupy the configured free buffers.

3.3.2. Adaptive Algorithm Implementation. The implementation of the adaptive algorithm is an important component of the ADCM approach for determining how to integrate the buffered and synchronous protocols. Our main goal is to improve the performance of MPI functions. However, hardware complexity is also an important measure in our design. Thus, the proposed algorithm should facilitate an optimal trade-off between performance and hardware complexity. This algorithm can be implemented in two parts from sender/receiver perspective: the adaptive sending and receiving algorithms.

Figure 10 shows the main frame of these two parts. The first part is the adaptive sending algorithm, as shown in Figure 10(a). This algorithm is relatively simple because it only needs to select the communication protocol from two options for sending messages. This algorithm accepts two inputs: the *RBC* table information and the data

```

1: // Note: RBC: receiving buffer credit table for all other nodes
2: //   Data: data to be sent
3: void adaptive_sending (RBC, SendingData) {
4:   if (RBC[dst][9:2] > len(SendingData) )           // Enough free buffers
5:     Buffered_sending();                             // Send using buffered protocol
6:   if (RBC[dst][9:2] >  $T_{s\_threshold}$  && RBC[dst][1:0] == '01' ) // buffer increase
7:     Buffered_sending();                             // Send using buffered protocol
8:   else                                              // Other cases: little buffer, buffer decrease
9:     Synchronous_sending();                         // Send using synchronous protocol
10: }

```

(a) adaptive sending protocol algorithm

```

1: // Note: LocalRBC: Local receiving buffer credit
2: //   ReceivingData: data received
3: void adaptive_receiving (LocalRBC, ReceivingData) {
4:   if ( is_confirm_received )                       // Confirm signal received
5:     release_sendingbuffer();                       // Release the sending buffers
6:   else if ( is_synchronous_protocol )              // Synchronous data received
7:     ReceiveNormally();                             // Send answer or Receive Data
8:   if ( is_buffered_protocol ) {                   // This packet is in buffered protocol
9:     if ( is_retry_operation ) {                   // This packet has retry data
10:      if ( is_speculative_retry ) {                // The retry is speculative
11:        if ( is_enough_buffer ) {                 // Enough buffers
12:          ReceiveNormally();                       // Receive normally
13:          Confirm();                             // Send confirm signals
14:        }
15:      else {                                       // Not enough buffers
16:        Receive_with_FreeBuffer();                 // Receive with free buffers
17:        TriggerRetryFollowing_when_BufferReady();   // Retry
18:      }
19:    }
20:  else                                           // Not speculative retry operation
21:    ReceiveNormally();                           // Receive normally with enough buffers
22: }
23: else if ( LocalRBC[9:2] > len (ReceivingData) ) { // Enough buffers
24:   ReceiveNormally();                           // Receive with enough buffers
25:   Confirm();                                   // Send confirm signals
26: }
27: else if ( LocalRBC[9:2] >  $T_{r\_threshold}$  && LocalRBC[1:0] == '01' ) {
28:   Receive_with_FreeBuffer();                   // Receive with free buffers
29:   TriggerSpeculative_RetryFollowing();          // Speculative retry
30: }
31: else {                                       // Other cases
32:   Receive_with_FreeBuffer();                   // Receive with free buffers
33:   TriggerRetryFollowing_when_BufferReady();     // Retry following
34: }
35: }

```

(b) adaptive receiving protocol algorithm

Fig. 10. Adaptive algorithm for ADCM.

to be sent. The algorithm compares the RBC value of the receiving node with the length of sending data flits. If the estimated number of receiving buffers is adequate to store all the data, the buffered protocol is simply chosen for message transmission. If the estimated number of free receiving buffers is insufficient for storing all the data but is likely to increase, the buffered protocol is also selected. This process is based on the following assumption: the buffers could be free after a short time when data are received. Nevertheless, a minimal number of current free buffers to make more free buffers is required for this case, which we call $T_{s_threshold}$. Generally, we can set this

value as half of the sending data size, as used in this article. For other cases such as when the number of free buffers is less than $T_{s_threshold}$ or the estimated number of receiving free buffers is likely to decrease, the synchronous protocol is chosen.

Figure 10(b) shows the adaptive receiving protocol algorithm. Similar to the sending algorithm, the receiving algorithm also accepts two data sources (*localRBC* and receiving data) as input. This algorithm is implemented in the following two steps: (1) analyze the receiving data and determine the next step of communication; and (2) send the response message to the data sender according to the decision of step 1. Considering that various types of messages could be received, this procedure is more complex for handling different messages. Line 4 verifies whether the confirmation message is received. Receipt of the confirmation message indicates that the data transmission is successfully completed. The response operation is very simple in that the receiver only needs to release its sending buffers for the storage of a specific transmission. Line 6 verifies whether the messages are received using the synchronous protocol. Such messages are of two types: request message or data message. To deal with the request message, the receiver could send the answer message only after it already has sufficient free buffers that can be reserved for this transmission. Receipt of the data message indicates that the receiving free buffer is sufficient for the synchronous protocol. Thus, the receiver can receive the data without the consideration of free buffers.

Line 8 verifies whether the message is received using the buffered protocol. Various cases should be considered to maximize the utilization of the buffered protocol to improve performance. Line 9 initially verifies the retry operation cases. The algorithm supports two different retry operations: *ready retry following* and *speculative retry following*. The *ready retry following* requires the receiver to trigger the retry operation only after the free buffers are ready to receive. If the free buffers are inadequate, buffers would be reserved only when they are free. To reduce the transmission delay further, *speculative retry following* is also integrated. For such type of retry operation, the receiver does not need to wait until the free buffer is ready before triggering the retry operation upon finding that the number of free buffers is insufficient for receiving the message data. Line 10 verifies whether the retry mechanism is speculative, whereas lines 10 to 21 do the respond operations accordingly. Notably, if the free buffers remain insufficient for receiving the *speculative retry following* message data, the *ready retry following* mechanism is triggered in following transmission.

Lines 23 to 34 deal with the normal first-time data message reception (no retry operation) using the buffered protocol, which can be handled in three cases: 1) the number of receiving free buffers is adequate, 2) the number of free buffers is more than $T_{r_threshold}$ and is likely to increase, and 3) other cases that do not belong to the other two cases. The first case is very simple in that the message only has to be received normally, after which the confirm signal is sent to the sender. The second case is related to the *speculative retry following*, which is used when current free buffers are insufficient for storing the received data, but the RPU predicts that the number can be adequate when the data transmission is again processed by the retry operation. Thus, RPU speculatively triggers the retry operation immediately after it finds insufficient free buffers. The last case is related to the *ready retry following*, which is the worst case. The synchronous protocol is likely to be used when the free buffers will be ready only after a long time. Such policy can save traffic workload if *speculative retry following* is triggered and is likely to improve performance.

Based on the description of the ADCM adaptive algorithm, we can see that the integration of buffered and synchronous protocols combines the best features of the two protocols to achieve ideal communication when the buffer prediction is correct. Otherwise, optimal performance is achieved by dynamically adjusting the response operations.

Fig. 11. Packet format.

3.3.3. Packet Format. Given that the communication protocol is implemented by hardware, the conventional packet format of the network should be modified to provide the receivers more information to perform adaptive operations. Figure 11 illustrates the packet format of ADCM. For comparison, a conventional NoC packet format is also shown in the figure. We explain the packet fields as follows: In the conventional 69-bit packet format shown in Figure 11(a), the packet comprises the header flit followed by payload flits. Two additional 3-bit heads are *Type* and *ID* (Identity) bits. The source and target address of the packet are included in the header flit. Passing a communication segment of the NoC, each packet has the same local identity number (ID-tag) for differentiation from other packets. The local ID-tag of the data flits of one packet will vary over different communication segments to provide a scalable concept.

To support the adaptive protocol for NoC transmission, additional bits should be integrated into the conventional packet format, called *ADCM*, which is shaded in Figure 11(b). The specific bits of *ADCM* are listed in the right figure, which comprises five control fields: *P*, *Type*, *R*, *S*, and *RBC*. The control bit *P* indicates whether the transmission is under the buffered or synchronous protocol. *P* also determines the following steps for handling the message: This bit will be set by the sender according to different buffer usage scenarios. The *Type* field specifies the packet type for ADCM: *data*, which is the actual packet for sending data that are located in the following data flits; *request*, which is the control packet for requesting data receiving under the synchronous protocol and does not contain actual data of the message; *answer*, which corresponds to the *request* packet under synchronous protocol; *retry*, which is the control packet for requesting a retry operation under the buffered protocol; and *confirm*, which is used to confirm to the sender that the receiver has received the message. The *R* field indicates whether the receiving data is in the retry operation. The *S* field is the speculative bit used to indicate whether the retry operation for sending data is speculative. If speculative retry is observed, the receiver would send the confirmation packet to the sender to complete the data transmission. The *RBC* field is used to provide the *localRBC* for the sender to estimate future buffer usage information.

Such *ADCM* packet format would not introduce a burden for packet encoding and transmission delay because additional flits would not be required. Furthermore, ADCM provides adequate information for the sender and receiver to determine the adaptive operation.

3.4. Comparison with Ideal Protocol

To reduce the hardware complexity and establish a practical design, the ADCM approach does not perform the same dynamic operations as the ideal protocol discussed in Section 2.4. ADCM mainly differs from ideal protocol in the following aspects.

—*Buffer usage information.* In ADCM, the current and future buffer usage information are estimated based on previous buffer usage information, which is inaccurate because the buffers can considerably change based on the node receiving the data.

Table I. Communication Design Configurations

Designs	Parameter	Measure
Processor design	frequency	1 GHz
	Cores	Alpha 21264
	L1 instruction cache	2-way 16KB
	L1 data cache	2-way 32KB
	L2 Cache	16-way 1MB
NoC design	Topology	2D Mesh 4×4
	Basic routing	Dimension-ordered XY
	Number of ports	5 ports
	VCs per port	4 VCs
	Buffers per VC	16 buffers
	Channel width/flit size	128 bits
	Packet size	8 flits
	MPU receiving buffer	32 buffers
	MPU unexpected buffer	32 buffers

Such inaccuracy may cause the ADCM to perform operations that are less optimal compared with those performed by the ideal protocol. This inaccuracy is mainly attributed to the long network latency of information transmission and the estimation algorithm. To improve estimation accuracy, we can perform optimization from two aspects: hardware support for efficient buffer usage transmission and accurate buffer usage estimation of local buffers such as some hints from programs. These optimizations are out of scope of this article and will be our future work.

- Retry granularity*. In ADCM, the retry following method is used. This process may introduce more network traffic compared with the retry discarded method used in the ideal protocol. However, the retry following does not need to record the address of discarded data and does not possess a complex control logic to send/receive these data. Furthermore, retry following can trigger the retry mechanism immediately after finding that an insufficient number of free buffers are available. This would improve the performance in some cases.
- Retry timing*. The selection of retry timing methods is based on the estimation of buffer usage information. Appropriate retry timing would facilitate performance improvement. In ADCM, the *speculative retry following* method (i.e., retry immediately method in Figure 3(b)) is used when estimates show sufficient free buffers. Otherwise, the *ready retry following* method in Figure 3(b) is used. Given that the buffer usage estimation information in ADCM is inaccurate, ADCM may make a wrong decision to result in poorer performance than the ideal protocol in some cases.

4. EVALUATION

4.1. Methodology

To evaluate the proposed communication design, we implemented its architecture using a SystemC-based cycle-level NoC simulator augmented with ADCM, which is modified from a NIRGAM simulator [Jain et al. 2007]. ADCM is based on the ME architecture. The simulator models a detailed pipeline structure for the NoC router and ME. We can change various network configurations, such as network size, topology, buffer size, routing algorithm, and traffic pattern. Table I lists the NoC design configurations in this study. The receiving and unexpected buffers in the NoC design are both 32 buffers, each having a 128-bit size. We compare the characteristics of the proposed communication architecture based on the ADCM scheme against the conventional ME design. The basic router is representative of conventional NoC designs, which originally has a four-stage

router pipeline. To shorten the pipeline, the basic router uses lookahead routing [Peh and Dally 2001] and the speculative method [Galles 1996].

Each simulation experiment is run until the network reaches steady state. The time for initializing ME is not considered part of message transmission. For the sake of comprehensive study, numerous validation experiments were performed for several combinations of workload types and network sizes. In the following section, the capability of the proposed communication design will be assessed for different traffic patterns including synthetic traffic and real application traffic.

The synthetic traffic patterns used in this research are the round-trip, uniform random, and hotspot traffic patterns to achieve a more specific evaluation for different traffic patterns. Each simulation runs for 1×10^6 cycles. To obtain stable performance results, the initial 1×10^5 cycles are used for simulation warmup, and the following 9×10^5 cycles are used for analysis. When destinations are chosen randomly, we repeat the simulation run five times and obtain the average of values obtained in each run. The MPI communication packet data size is also chosen randomly, which ranges from 128 bits (1 flit) to 1024 bits (8 flits).

We also studied the ADCM approach using real application communication traffic. Traces for the baseline conventional implementations were gathered on a full-system multicore simulator M5 [Binkert et al. 2006]. We model our target multicore systems with the Alpha Instruction Set Architecture (ISA), which is the most stable ISA supported in M5. Each core is modeled with two-way 16KB L1 Icache, two-way 32KB Dcache, and 1MB L2 cache. We use the NAS Parallel Benchmarks (NPB 2.4) suites as application traffic to evaluate the ADCM design. The applications used to perform the experiments are a subset of the A class NPB, a well-known, allegedly representative set of application workloads often used to assess the performance of parallel computers. These applications include three kernels Conjugate Gradient (CG), Integer Sort (IS), discrete 3D fast Fourier Transform (FT), as well as two pseudo applications Block Tridiagonal solver (BT) and Scalar Pentadiagonal solver (SP). Considering that the current baseline hardware implementation of MPI functions only supports point-to-point MPI communication, other communication types such as collective communication are performed through these basic MPI communications.

4.2. Synthetic Traffic Results

The synthetic traffic represents different communication patterns that facilitate the evaluation of the ADCM approach. In the following text, we will analyze the experimental results from the aspects of bandwidth, traffic, and delay. Higher MPI bandwidth indicates that the communication design can achieve higher execution performance, which has been adopted as the evaluation metric in numerous experiments for MPI implementation.

4.2.1. Round-Trip Traffic Pattern. We first record the time taken for a number of round-trip message transfers. The randomly generated message (i.e., destinations of unicast messages at each node are selected randomly) is sent to the target node by the *MPI_Send* instruction. When the target node receives this message, it will first perform the *MPI_Receive* instruction and then return the message back to the source node immediately without changing anything by the *MPI_Send* instruction. Considering that receivers prepost *MPI_Receive* instructions, this round-trip test can help us to obtain the maximum network bandwidth of the communication system. Assuming that the maximum capacity of the L1 cache in the multicore processor is 32KB, the maximum length of message triggered by MPI primitive instructions should be set to 16KB (for send and receive operations).

Fig. 12. Bandwidth comparison under round-trip traffic pattern.

Fig. 13. Bandwidth comparison for prepost receive ratio variation.

Figure 12 shows the bandwidth results of different protocols under a round-trip traffic pattern with varying message size from 1B to 16384B. As the message size increases, the communication bandwidth rapidly increases, such that more time can be used for real data transmission. It can be seen from the figure that the buffered protocol achieves significantly higher bandwidth than the synchronous protocol primarily because the buffered protocol does not need the hand-shaking process or retry operations for its prepost receives. As expected, the proposed ADCM approach achieves the same bandwidth as the buffered protocol. With sufficient buffers and prepost receive operations, the buffered protocol exhibits a special case of ADCM.

To understand the buffered protocol's disadvantages and ADCM's advantages further, Figure 13 illustrates how the bandwidth varies in terms of percentage of prepost receive operations with 4KB message. Prepost receive indicates that such receive operations are preposted before the time messages arrive. The synchronous protocol outperforms the buffered protocol when 58% or less receives are preposted. The ADCM approach achieves better communication bandwidth by dynamically performing corresponding protocol behavior based on buffer usage and communication demand.

4.2.2. Hotspot Traffic Pattern. For a hotspot traffic pattern, we set one or two network nodes as the hotspot nodes to which other nodes send data messages with a greater probability. To simulate the unexpected cases of receiving messages, *MPI_Receive* is executed later than *MPI_Send* (uniform random ranging from 1 to 256 cycles) with a 30% possibility. Unlike round-trip traffic, we use the average network traffic and

Fig. 14. Network traffic comparison under hotspot traffic pattern.

Fig. 15. Message delay comparison under uniform random pattern.

message delay in the hotspot and real traffic scenarios. We first evaluate the protocols through network traffic, which is identified as the number of bytes transmitted through the underlying NoC. The synchronous protocol serves as the baseline design, that is, the protocol is normalized to one in different hotspot scenarios. Figure 14 illustrates the network traffic comparison results for different communication protocols. The buffered protocol entails significantly more network traffic than the synchronous protocol for the retry mechanism, which is approximately 29% for the single-hotspot scenario and approximately 43% for the double-hotspot scenario. The ADCM approach minimizes the network traffic overhead. This traffic overhead can primarily be attributed to the wrong buffer usage prediction or partial data retry operations.

Figure 15 illustrates the average message delay comparison results for different protocols. We define message delay as the time interval between the send and receive operations for each data unit. The ADCM approach has the least message delay compared with the two other protocols. ADCM outperforms the synchronous protocol in terms of the elimination of the hand-shaking process in most cases and is better than the buffered protocol in terms of the reduction of retry delay. The message delay in the double-hotspot scenario is longer than that in the single-hotspot scenario primarily because of increased retry delay and network traffic contention. ADCM minimizes this negative impact, achieving the best trade-off between message delay and network traffic.

Fig. 16. Network traffic comparison under real traffic.

Fig. 17. Message latency comparison under real traffic.

4.3. Real Traffic Results

We run application traffic to evaluate ADCM. These application benchmarks were selected for their large number of message transmissions and unexpected receives. Considering that the on-chip buffers are limited for preserving these data, a pure buffered protocol is unsuitable. Figure 16 shows the network traffic comparison results for different benchmarks. The buffered protocol required an average of approximately 22% more network traffic than the synchronous protocol. This network traffic overhead is decreased to 7% when using ADCM.

Figure 17 shows the message delay comparison results for different benchmarks. The buffered protocol has 34% lower message delay than the synchronous protocol, whereas ADCM has 42% lower message delay than the synchronous protocol on average. This reduced message delay can facilitate application performance improvement. The *CG* benchmark achieves the lowest message delay reduction because of its short message size and relative low unexpected receives. In conclusion, the real traffic results demonstrated that the proposed ADCM approach achieves the best performance metric with minimized network traffic overhead.

The preceding memory access latency and network traffic measurements are more related to Instructions Per Cycle (IPC) and are beneficial only in sequential processing. In a parallel environment, the valid performance metric to be used is the total execution time. Nevertheless, these two measurement improvements would result in overall performance improvement. To determine the execution time of an application, we added the information on computation execution time into the real application trace. Figure 18 shows the performance results of different communication protocols in terms

Fig. 18. Execution time comparison under real traffic.

Fig. 19. ADCM prediction accuracy for the tested benchmarks.

of execution time. The primary effect of the ADCM approach is the reduction in MPI transmission delay, which would result in application speedup. Compared with the synchronous protocol, ADCM has greater execution time reduction of approximately 30% on average. ADCM also has a performance advantage over the buffered protocol of approximately 4% execution time reduction on average. That is, using ADCM, the programmer can avoid the burden of handling the buffers in the MPI application to achieve even better performance than the buffered protocol.

Figure 19 illustrates the ADCM prediction accuracy for the tested benchmarks. Prediction accuracy is defined as the ratio that can use receive buffers appropriately. This result can help us understand the effectiveness of the prediction mechanism. The figure shows that ADCM can achieve an average of approximately 87% prediction accuracy for the NPB benchmarks. That is, in most cases, ADCM can make the best use of the receiving buffer. A wrong prediction may trigger unnecessary retry operations or other operations used in a synchronous protocol, which is why ADCM also has small network traffic overhead.

4.4. Sensitivity Analysis

To analyze the performance of ADCM further, this subsection presents the performance results of varying values of ADCM design parameters. Figure 20 shows the sensitivity results to receiving buffer sizes of 16, 32, and 64KB. We can find that a buffer with 64KB achieved the best performance with the largest memory consumption. The 32KB buffer is approximately 38% higher than 16KB buffer but approximately 15% lower than 64KB buffer. Considering the memory resource for multicore systems, we set the size of the hardware management buffer to 32KB in this work.

Fig. 20. Comparison results for varying sizes of receiving buffer.

Fig. 21. Comparison results for varying value of $T_{s_threshold}$.

Figure 21 shows the sensitivity results to $T_{s_threshold}$ parameter for the percentage of 40, 50, and 60. The optimal value of $T_{s_threshold}$ can be changed for different applications. If the value is set too large, then the communication would act as a synchronous protocol with minimal performance benefits. If the value is set too small, then the communication would act as a buffered protocol with numerous retry operations resulting in performance degradation. $T_{s_threshold}$ at a percentage of 50 has a evident performance advantage over a percentage of 40 on an average of 19%. However, this case does not hold true in the comparison with a percentage of 60. This finding demonstrates the application-specific features of $T_{s_threshold}$ parameter. In this article, we choose a percentage of 50 as a reasonable configuration value for the evaluation.

4.5. Hardware Overhead Analysis

The proposed ADCM approach is implemented based on a conventional MPU. The introduced hardware realization overhead includes the ADCM control logic and RBC table. To estimate hardware cost, we implemented the ADCM hardware in Verilog and performed logic synthesis by using the Synopsys Design Compiler to obtain the area information. We used TSMC 90nm CMOS generic process technology for logic synthesis. The hardware overhead for the entire proposed ME hardware, including PR, PPU, and MPU, is listed in Table II. For the impact of the proposed modifications, approximately 3% area overhead for ADCM is observed. This small hardware overhead of ADCM results in significant network traffic and message delay reduction, which is demonstrated to be an effective communication protocol.

Table II. Hardware Estimation for the ME in Propose ADCM

Units	PR	PPU	MPU	Total
Area (gate count)	975	1325	9570	94
Ratio (%)	8.2%	11.2%	80.8%	100%

5. RELATED WORKS

A natural way of providing MPI functionality on multicore processors is to port conventional MPI implementations such as MPICH [2012]. However, this approach is unsuitable for on-chip systems that have limited resources. Some implementations have been ported for high-end embedded systems with large memories such as MPI/PRO [Verari Systems 2012]. However, such implementations are likewise unsuitable for NoC systems. Thus, a large number of studies have focused on the MPI adaption for multicore systems [Ohara et al. 2006; Williams et al. 2006; Mahr et al. 2008; Psota and Agarwal 2008; Saldaña and Chow 2006; Agbaria et al. 2006].

To improve the performance of MPI applications, a large number of works on software optimization and hardware support have been introduced. For multiprocessor systems, MPI optimization is an extensively investigated domain along various directions, optimizing implementation on a generic architecture such as a cluster [Hoefer et al. 2007] or on a specific machine [Feind and McMahon 2006].

Ogawa and Matsuoka [1996] used compiler modifications to optimize MPI. The compiler would recognize the MPI calls in a program, perform a static analysis to determine which arguments are static, and then create specialized MPI functions for that program. Faraj and Yuan [2005] presented a method for automatically optimizing the MPI collective subroutines. Karwande et al. presented a method for compiled communication (CCMPI), which applies more aggressive optimizations to communications with information that is known at compile time [Karwande et al. 2005]. The works presented in Liu et al. [2004] and Hoefer et al. [2007] used hardware multicast in native InfiniBand to improve the performance of MPI broadcast operation. However, these approaches cannot be directly used in NoC infrastructure under different constraints. For on-chip multicore systems, Peng et al. [2011] considered the NoC designs for low-overhead broadcast and reduced transmission but did not consider the communication protocol.

The communication protocol has been investigated, and applications were found to have the tendency to consume time on traversing message queues, thus resulting in an increase in performance gap [Goudy 2005]. Thus, a unique hardware structure is extended to accelerate list traversal and matching [Underwood et al. 2005]. An active research area is the use of reconfiguration to improve application performance such as adaptive MPI [Huang et al. 2006] and Maghraoui et al. [2005]. Venkata et al. showed how reconfiguration can be used to improve bandwidth availability. They also used profile data for fine-grained runtime reconfiguration and provided a framework that can be used to implement other similar reconfigurations [Venkata and Bridges 2006; Venkata et al. 2009]. Other systems such as STAR-MPI [Faraj et al. 2006] and HP-MPI [David 2007] have shown that profile data can be used for optimizing MPI performance at link time or launch.

Unlike the aforementioned optimization techniques, this article focuses on multicore architectures and utilizes the advantages of on-chip hardware resources. Given that power, area, and latency constraints for off- versus on-chip communication architectures differ substantially, prior off-chip communication architectures are not directly suitable for on-chip usage. Thus, this article proposes a new communication mechanism for accelerating MPI functions using an adaptive implementation technique.

6. CONCLUSIONS

In this article, ADCM, an adaptive communication mechanism for accelerating MPI functions on NoC-based multicore architectures, has been proposed. ADCM integrates two conventional communication protocols, namely, the buffered and synchronous protocols, and behaves adaptively according to the application and NoC configurations. ADCM exhibits behavior similar to buffered communication when a sufficient number of buffers are available in the receiver but exhibits behavior similar to the synchronous protocol when the receiver has limited buffers. ADCM can combine the advantages of the buffered and synchronous communication protocols to achieve better throughput and performance. The promising results confirm that the proposed ADCM communication mechanism can be effectively used in future NoC designs to accelerate MPI functions.

REFERENCES

- AGBARIA, A., KANG, D.-I., AND SINGH, K. 2006. Lmpi: Mpi for heterogeneous embedded distributed systems. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*. 79–86.
- ARTERIS. 2011. Network on chip (noc) interconnect ips. <http://www.artemis.com/>.
- BINKERT, N. L., DRESLINSKI, R. G., HSU, L. R., LIM, K. T., SAIDI, A. G., AND REINHARDT, S. K. 2006. The MS simulator: Modeling networked systems. *IEEE Micro* 26, 4, 52–60.
- FARAJ, A. AND YUAN, X. 2005. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM Press, New York, 393–402.
- FARAJ, A., YUAN, X., AND LOWENTHAL, D. 2006. Star-mpi: Self tuned adaptive routines for mpi collective operations. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06)*. ACM Press, New York, 199–208.
- FEIND, K. AND MCMAHON, K. 2006. An ultrahigh performance mpi implementation on sgi ccnuma altix systems. *Comput. Methods Sci. Technol.* 2006 (Special Issue), 67–70. <http://www.man.poznan.pl/cmst/2005/las/10-Feind-Gra.pdf>.
- GALLES, M. 1996. Scalable pipelined interconnect for distributed endpoint routing: The sgi spider chip. In *Proceedings of the 19th International Conference on Parallel and Distributed Computing Systems (ISCA'96)*. 141–146.
- GOOSSENS, K., DIELISSEN, J., MEERBERGEN, J. V., PAPLAVCO, P., RADULESCU, A., ET AL. 2003. Guaranteeing the quality of services in networks on chip. In *Networks on Chip*. Kluwer Academic Publishers, 61–82.
- GOUDY, S. 2005. A preliminary analysis of the mpi queue characteristics of several applications. In *Proceedings of the International Conference on Parallel Processing (ICPP'05)*. IEEE Computer Society, Washington, DC, 175–183.
- GROPP, W., LUSK, E., AND SKJELLUM, A. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA.
- HOEFLE, T., SIEBERT, C., AND REHM, W. 2007. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*. 1–8.
- HOWARD, J. E. A. 2010. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proceedings of the IEEE International Solid State Circuit Conference Digest of Technical Papers (ISSCC'10)*. 108–109.
- HUANG, C., ZHENG, G., KALÉ, L., AND KUMAR, S. 2006. Performance evaluation of adaptive mpi. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM Press, New York, 12–21.
- JAUME, J. 2009. A lightweight mpi-based programming model and its hw support for noc-based mpsoes. In *IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE'09)*.
- KARWANDE, A., YUAN, X., AND LOWENTHAL, D. K. 2005. An mpi prototype for compiled communication on ethernet switched clusters. *J. Parallel Distrib. Comput.* 65, 10, 1123–1133.
- KUMAR, R., MATTSO, T. G., POKAM, G., AND VAN DER WIJNGAART, R. F. 2011. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*, M. Hbner and J. Becker, Eds., Springer, 115–123.
- JAIN, L. 2007. Nirgam: A simulator for noc interconnect routing and applications modeling. In *Proceedings of the ACM/IEEE Design, Automation, and Test in Europe Conference*.

- LAM/MPI. 2013. <http://www.lam-mpi.org/>.
- LIU, J., MAMIDALA, A. R., AND PANDA, D. K. 2004. Fast and scalable mpi-level broadcast using infiniband's hardware multicast support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*.
- MAGHRAOUI, K. E., SZYMANSKI, B. K., AND VARELA, C. A. 2005. An architecture for reconfigurable iterative mpi applications in dynamic environments. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM'05)*. 258–271.
- MAHR, P., LORCHNER, C., ISHEBABI, H., AND BOBDA, C. 2008. Soc-mpi: A flexible message passing library for multi-processor systems-on-chips. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAS' (ReConFig'08)*. 187–192.
- MATTSON, T. G., RIEPEN, M., LEHNIG, T., BRETT, P., HAAS, W., KENNEDY, P., HOWARD, J., VANGAL, S., BORKAR, N., RUHL, G., AND DIGHE, S. 2010. The 48-core scc processor: The programmer's view. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Computer Society, 1–11.
- MATTSON, T. G., VAN DER WIJNGAART, R., AND FRUMKIN, M. 2008. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'08)*. 38:1–38:11.
- MPICH. 2012. <http://www.mcs.anl.gov/mpi/mpich>.
- MURILLO, J. J. 2009. HW-sw components for parallel embedded computing on noc-based mpsoes. Ph. D. thesis. <http://www.tdx.cat/bitstream/handle/10803/5779/jjm1de1.pdf;jsessionid=6B2C471E958912BEE5D756A02B03D3D3.tdx2?sequence=1>.
- OGAWA, H. AND MATSUOKA, S. 1996. Ompi: Optimizing mpi programs using partial evaluation. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 37.
- OHARA, M., INOUE, H., SOHDA, Y., KOMATSU, H., AND NAKATANI, T. 2006. Mpi microtask for programming the cell broadband engine processor. *IBM Syst. J.* 45, 85–102.
- PEH, L.-S. AND DALLY, W. J. 2001. A delay model and speculative architecture for pipelined routers. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA'01)*. 255–266.
- PENG, Y., SALDANA, M., AND CHOW, P. 2011. Hardware support for broadcast and reduce in mpsoe. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'11)*. IEEE Computer Society, Washington, DC, 144–150.
- PETROVIC, D., SHAHMIRZADI, O., ROPARS, T., AND SCHIPER, A. 2012. High-performance rma-based broadcast on the intel scc. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM Press, New York, 121–130.
- PSOTA, J. AND AGARWAL, A. 2008. Rmpi: Message passing on multicore processors with on-chip interconnect. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'08)*. 22–37.
- SALDANA, M. AND CHOW, P. 2006. Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL'06)*. IEEE Computer Society, Washington, DC, 1–6.
- SOLT, D. 2007. A profile-based approach for topology aware mpi rank placement. In *Proceedings of the High Performance Computation Conference (HPCC'07)*.
- UNDERWOOD, K. D., HEMMERT, K. S., RODRIGUES, A., MURPHY, R., AND BRIGHTWELL, R. 2005. A hardware acceleration unit for mpi queue processing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*. IEEE Computer Society, Washington, DC, 96.2.
- VENKATA, M. G. AND BRIDGES, P. G. 2006. Mpi/ctp: A reconfigurable mpi for hpc applications. In *Proceedings of the 13th European PVM/MPI Users Group Meeting*. Lecture Notes in Computer Science, vol. 4192. Springer, 96–104.
- VENKATA, M. G., BRIDGES, P. G., AND WIDENER, P. M. 2009. Using application communication characteristics to drive dynamic mpi reconfiguration. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE Computer Society, 1–6.
- VERARI SYSTEMS, I. 2012. <http://www.mpi-softtech.com/>.
- WENTZLAFF, D. E. A. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 15–31.
- WILLIAMS, J. A., SYED, I., WU, J., AND BERGMANN, N. W. 2006. A reconfigurable cluster-on-chip architecture with mpi communication layer. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. IEEE Computer Society, Washington, DC, 350–352.

Received August 2012; revised December 2012; accepted February 2013