Supporting Distributed Shared Memory on Multi-core Network-on-Chips Using a Dual Microcoded Controller

Xiaowen Chen[†,‡], Zhonghai Lu[‡], Axel Jantsch[‡] and Shuming Chen[†]
[†]National University of Defense Technology, 410073, Changsha, China
[‡]KTH-Royal Institute of Technology, 16440 Kista, Stockholm, Sweden
[†]{xwchen,smchen}@nudt.edu.cn [‡]{xiaowenc,zhonghai,axel}@kth.se

***Abstract*—Supporting Distributed Shared Memory (DSM) is essential for multi-core Network-on-Chips for the sake of reusing huge amount of legacy code and easy programmability. We propose a microcoded controller as a hardware module in each node to connect the core, the local memory and the network. The controller is programmable where the DSM functions such as virtual-to-physical address translation, memory access and synchronization etc. are realized using microcode. To enable concurrent processing of memory requests from the local and remote cores, our controller features two mini-processors, one dealing with requests from the local core and the other from remote cores. Synthesis results suggest that the controller consumes 51k gates for the logic and can run up to 455 MHz in 130 nm technology. To evaluate its performance, we use synthetic and application workloads. Results show that, when the system size is scaled up, the delay overhead incurred by the controller may become less significant when compared with the network delay. In this way, the delay efficiency of our DSM solution is close to hardware solutions on average but still have all the flexibility of software solutions.**

I. Introduction

It's a trend that high-performance single-chip computing architectures evolves from single-core to multi- and even many cores [1][2]. Network-on-Chip (NoC) [3][4][5] is recognized as the scalable solution to interconnect and organize so many cores and hence has attracted significant attentions over the last ten years since various buses do not scale well with the system size. For instance, in 2007, Intel researchers announced their research about a prototype multi-core NoC architecture containing 80 tiles arranged as a 10x8 2D mesh network [6]. Another trend is that the rapid development of integrated circuit technology enables more and more computing resources and storage elements to be integrated on a single chip [7]. The embedded memory content in System-on-Chips (SoCs) increases from 20% ten years ago to 85% of the chip area today and will continue to increase in the future [8]. Memories are preferably to be distributed for medium and large scale system sizes because centralized memory has already become the bottleneck of performance, power and cost.

Following the two trends, a key question for such multi-core, distributed memory architectures is what kind of communication paradigm, *shared variable* or *message passing*, to support? In our view, we envision that there is an urgent need to support Distributed but Shared Memory (DSM) because of the huge amount of legacy code and easy programming. To increase productivity and reliability and to reduce risk, reusing proven legacy code is a must. From the programmers' point of view, the shared memory programming paradigm provides a single shared address space and transparent communication, since there is no need to worry about when to communicate, where data exist and who receives or sends data, as required by explicit message passing API.

A multi-core NoC chip integrates a number of resources and may be used to support many use cases. Its design complexity results in long time-to-market and high cost. This motivates us to look for a flexible way to address DSM issues on multi-core NoCs. As we know, performance and flexibility are paradoxical. Dedicated hardware and software-only solutions are two extremes. Dedicated hardware solutions can achieve high performance, but any small change in functionality leads to redesign of the entire hardware module and hence the solutions suffice only for limited, static cases. Software-only solutions require little hardware support and main functions are implemented in software. They are flexible but may consume significant cycles, thus potentially limiting the system performance. Microcode approach is a good alternative to overcome the performance-flexibility dilemma. Its concept can be traced back to 1951 when it was first introduced by Wilkes [9]. Its crucial feature offers a programmable and flexible solution to accelerate a wide range of applications [10].

Along the aforementioned consideration, we adopt the microcode approach to address DSM issues on multi-core NoCs, aiming for hardware performance but maintaining the flexibility of programs. We present a programmable hardware module, named *Dual Microcoded Controller (DMC)*, to allow users to implement various functions. Each node hosts a DMC connecting the core, the local memory and the network. Each DMC features two minprocessors to be able to concurrently deal with requests from the local core and remote cores via the network. The execution of the mini-processors is triggered by memory requests in form of command. Basic DSM functions are realized and experimental results shows that, when the system size is scaled up, the overhead incurred by the DMC may become less significant when compared with the network delay. The DMC solution is demonstrated to be a viable way and its delay efficiency is close to hardware solutions on average but still have all the flexibility of software solutions.

The rest of the paper is organized as follows. Section II discusses the related work. Section III describes our target architecture: DSM based multi-core NoCs. In section IV, we present the architecture, the operation mechanism and the hardware cost of the DMC. Section V realizes the basic DSM functions using microcode and analyzes the performance. Experimental results with synthetic and application workloads are reported in section VI. Finally we conclude in section VII.

II. Related Work

The Alewife [11] machine from MIT addresses the problem of providing a single addressing space with integrated message passing mechanism. This is a dedicated hardware solution, and does not support virtual memory.

Few previous works used the microcode approach to address the DSM issues in multiprocessor systems. Similar to our programmable controller, both the Stanford FLASH [12] and the Wisconsin Typhoon [13] use a programmable co-processor (the MAGIC in the FLASH, the NP in the Typhoon) to support flexible cache coherence policy and communication protocol. However, both machines were developed not for on-chip network based

978-3-9810801-6-2/DATE10 © 2010 EDAA

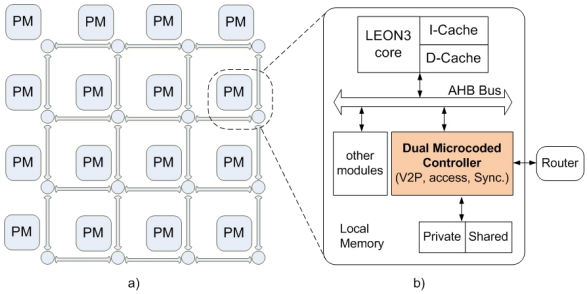

Fig. 1. a) A 16-node mesh multi-core NoC, b) Processor-Memory node

multi-core systems. The MAGIC only hosts one programmable coprocessor handling requests from the processor, the network and the I/O. The NP also uses one programmable coprocessor to deal with requests from the network and the CPU. If two or more requests come concurrently, only one can compete to be handled while the others have to be delayed, resulting in contention delay. Our DMC hosts two mini-processors to enable two concurrent processing of requests from the processor and the network, eliminating this overhead. Introducing another processor is non-trivial because we also need to address the synchronization due to possible simultaneous access requests to the same region in the local memory. Furthermore, the MAGIC and the NP organize memory banks to form a cache-coherent shared memory. Memory accesses are handled by the programmable coprocessor to hit the right memory banks in local or remote nodes. However, this causes larger processing time, compared with dedicated hardware solution. It also forces the local processor to spend more time even on the data only used by itself. In our memory organization, the memory is partitioned into a private part and a shared part. The private memory accesses are fast since they bypass the mini-processors so as to improve the performance. The SMTp [14] exploits SMT in conjunction with a standard integrated memory controller to enable a coherence protocol thread used to support DSM multiprocessors. The protocol programmability is offered by a system thread context rather than an extra programmable coprocessor. It utilizes the main processor's resources, while our DMC is a synergistic processing module to alleviate the burden of the main processor.

III. Target Architecture: Distributed Shared Memory Based Multi-core Network-on-Chips

Fig. 1 a) shows an example of our DSM based multi-core NoC architecture. The system is composed of 16 Processor-Memory (PM) nodes interconnected via a packet-switched network. The network topology is a mesh, which is a most popular NoC topology proposed today [15]. As shown in Fig. 1 b), each PM node contains a processor, for example, a LEON3 [16] shown in the figure, hardware modules connected to the local bus and a local memory. Our proposal is the hardware module, named *Dual Microcoded Controller (DMC)*, connecting the processor, the local memory and the network, and serving requests from the local processor and the remote processors via the network concurrently.

As can be observed, memories are distributed in each node and tightly integrated with processors. All local memories can logically form a single global memory address space. However, we do not treat all memories as shared. As illustrated in Fig. 1 b), the local memory is partitioned into two parts: *private* and *shared*. And two addressing schemes are introduced: *physical addressing* and *logic (virtual) addressing*. The private memory can only be accessed by the local processor and it's physical. All of shared

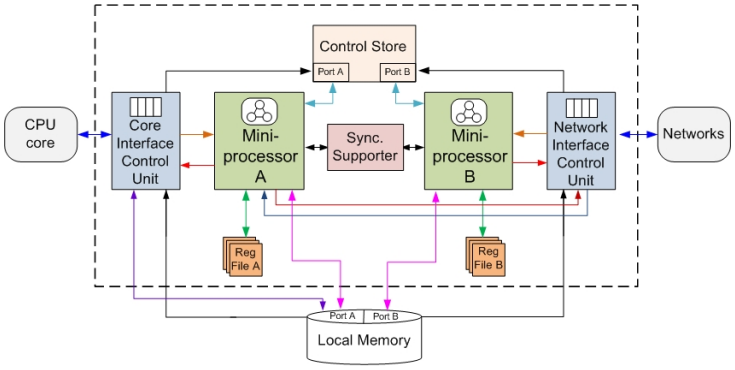

Fig. 2. Architecture of the Dual Microcoded Controller

memories are visible to all nodes and organized as a Distributed Shared Memory (DSM) and they are virtual. The philosophy of this design is to speed up frequent private access as well as to maintain a single virtual space. For shared memory access, there requires a virtual-to-physical (V2P) address translation. Such translation incurs overhead but makes DSM organization transparent to application, thus facilitating programming.

IV. Dual Microcoded Controller

In this section, we detail the architecture of the Dual Microcoded Controller, how it operates and its hardware cost.

A. Architectural Design

As shown in Fig. 2, the DMC, which connects to the CPU core, the Local Memory, and the network, mainly contains six parts, namely, *Core Interface Control Unit (CICU)*, *Network Interface Control Unit (NICU)*, *Control Store*, *Mini-processor A*, *Mini-processor B*, and *Synchronization Supporter*. As their names suggest, the CICU provides a hardware interface to the local core, and the NICU a hardware interface to the network. The two mini-processors are the central processing engine. Microprogram is initially stored in the Local Memory, and will be dynamically uploaded into the Control Store on demand during the program execution. The Synchronization Supporter coordinates the two mini-processors to avoid simultaneous accesses to the same memory address and guarantees atomic read-and-modify operations. Both the Local Memory and the control store are dual ported: port A and B, which connect to the mini-processor A and B, respectively. The functions of each module are detailed as follows:

Core Interface Control Unit

The CICU connects with the core, the mini-processor A, the NICU, the Control Store and the Local Memory. Its main functions are: (I) it receives local requests in form of command from the local core and triggers the operation of the mini-processor A accordingly; (II) it uploads the microcode from the Local Memory to the Control Store through port A; (III) it receives results from the mini-processor A; (IV) it accesses the private memory directly using physical addressing if the memory access is private; (V) it sends results back to the local core.

Network Interface Control Unit

The NICU connects the network, the mini-processor B, the CICU, the Control Store and the Local Memory. Its main functions are: (I) it receives remote requests in form of command from the network and triggers the operation of the mini-processor B accordingly; (II) it also can upload the microcode from the Local Memory to the Control Store through port B; (III) it sends remote requests from the mini-processor A or B to remote destination nodes in format of message via the network; (IV) it receives the

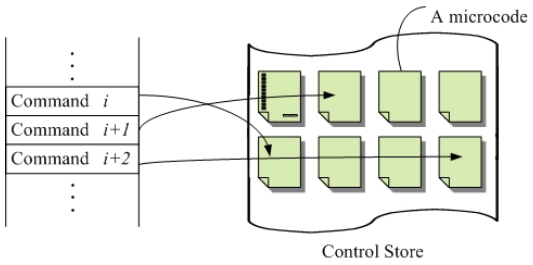

Fig. 3. One command triggers a microcode

remote results in format of message from remote destination nodes via the network and forwards them to the mini-processor A or B.

Mini-processor A

The mini-processor A connects with the CICU, the register file A, the Synchronization Supporter, the Control Store, and the Local Memory. Its operation is triggered by a command from the local core. It executes microcode from the Control Store through port A, uses register file A for temporary data storage, and accesses the Local Memory through port A.

Mini-processor B

The mini-processor B connects with the NICU, the register file B, the Synchronization Supporter, the Control Store, and the Local Memory. Its operation is triggered by a command from remote cores via the network. It executes microcode from the Control Store through port B, uses register file B for temporary data storage and accesses the Local Memory through port B.

The two min-processors features a five-stage pipeline and four function units: *Load/Store Unit (LSU)*, *Adder Unit (AU)*, *Condition Unit* and *Message Passing Unit (MPU)*, to provide operations of memory access, addition, conditional branch and message-passing. The microinstructions are designed to exploit the hardware architecture of the mini-processors. the microinstructions are organized *horizontally* [17]. For concise presentation, we do not explain the microinstructions in detail.

Synchronization Supporter

The Synchronization Supporter, which connects with the mini-processor A and B is a hardware module to support atomic read-and-modify operation. This is necessary when two synchronization requests try to access the same lock at the same time.

Control Store

The Control Store, which connects with the CICU, the NICU, the mini-processor A and B and the Local Memory, is a local storage for microcode, like an instruction cache. It dynamically uploads microcode from the Local Memory. It feeds microcode to the mini-processor A through port A, and the mini-processor B through port B. This uploading and feeding are controlled by the CICU for commands from the local core and the NICU for commands from remote cores via the network.

In summary, the DMC features (i) dual interfaces and dual processors, (ii) cooperation of the interface units and the mini-processors, (iii) dual-port shared Control Store and Local Memory, (iv) hardware support for mutex synchronization and (v) dynamic uploading microcode into the Control Store.

B. Operation Mechanism

For the DMC, the execution of the mini-processors is triggered by requests (in form of command) from the local and remote cores. This is called *command-triggered microcode execution*. As shown in Fig. 3, a command is related to a certain function, which is implemented by a microcode. A microcode is a sequence of microinstructions with an **end** microoperation at the end. A

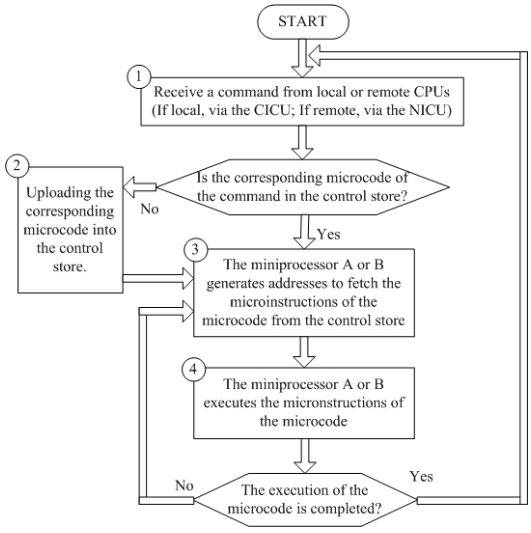

Fig. 4. Work flow of the DMC

TABLE I
SYNTHESIS RESULTS

	Optimized for area	Optimized for speed
Frequency	444 MHz (2.25 ns)	455 MHz (2.2 ns)
Area (Logic)	44k NAND gates	51k NAND gates
Area (Control Store)	300k NAND gates	

microprogram is a set of microcodes. Fig. 4 shows how the DMC works (Microprogram is initially stored in the Local Memory). This procedure is iterated over the entire execution of the system.

C. Hardware Implementation

The DMC design is synthesized by Synopsys® Design Compiler in Chartered® 0.13μm technology and the Control Store is generated by Artisan® Memory Compiler. The Control Store is composed of four 1024*32b Dual Port SRAMs. The synthesis results are listed in TABLE I. As we can see, the DMC can run up to 455 MHz consuming 51K gates if optimized for speed.

V. REALIZING DSM FUNCTIONS

Using microcode, we implement basic DSM functions: V2P address translation, shared memory access and synchronization.

A. Virtual-to-Physical (V2P) Address Translation

To maintain a Distributed Shared Memory environment, each time the request (in form of command) from the local core or a remote core comes, the Virtual-to-Physical address translation is always performed at first to obtain the physical address. And then, the target microcode related to this command will be executed. Fig. 5 a) shows this procedure. In the figure, the microinstructions above the red dash line is used to translate the logic address into the physical address. Conventional page lookup table [18] is used to implement the V2P address translation. The translation takes 11 cycles. The remainder microinstructions distinguish whether the target microcode is local or remote. If local, the execution jumps where the target microcode is; if remote, a message-passing is started up to request the execution in the remote destination node.

B. Shared Memory Access

Shared memory access is implemented by microcode. We categorize it into two types: (1) Local shared access; (2) Remote shared access. Because shared memory access uses logical addressing, it implies a V2P translation overhead. Here, we use bursty read and write as an example. Fig. 5 b) shows the microcode for memory read and write of a burstiness of ***n*** words.

```
01)   sub A0, L_ADDR, BADDR   ;Calculate L_ADDR (logical addr.) – BADDR (boundary addr.)
02)   nop
03)   pfe A0, A1, A0          ;Extract page No./page offset into A0 /A1.
04)   nop
05)   add A3, A0, V2P_HADDR   ;Compute the index of page frame No. in the V2P table.
06)   nop
07)   add A7, A3, 3           ;Compute the index of destination node No. in the V2P table.
08)   lfw *A3, A2             ;Load the page frame No. from the V2P table into A2.
09)   lfw *A7, A4             ;Load the destination node No. from the V2P table into A4.
10)   set A5, 1               ; A5<->QoS, 1-Best Effort
11)   pfm A2, A1, A6          ;Merge frame No.(A2) and offset(A1), obtaining physical addr.
12)   beq A4, SNODE, LOCAL    ;Branch to the LOCAL line if the access is local.
13)   nop
14)   mp A4, A5, A6, DATA     ;For the remote shared memory, start transfering the data.
15)   nop
16)   end 3                   ;Return '3', the data will be back from the remote node.
17)LOCAL: jmp START_ADDR      ;Jump to where the target microcode is.
18)   nop
```

V2P address translation

a)

```
;BURST_LOAD_WORD                    ;BURST_STORE_WORD
      set A0, n                           set A0, n                  ;set the burst number
BLW: sub A0, A0, 1 || bneqz A0, BLW BSW: sub A0, A0, 1 || bneqz A0, BSW
      lw *A6, cpu_core || add A6, A6, 4     sw *A6, DATA || add A6, A6, 4   ;use the branch slot
      end 1                               end 1
```

b)

```
;TEST and SET lock
01)          ll *A6,A0 ;load the lock's value
02)          nop
03)          nop
04)          bneqz A0, FAILED ;distinguish the lock is locked or not
05)          nop
06) SUCCESS: sc *A6, 1
07)          nop
08)          end 1
09) FAILED:  sc *A6, A0
10)          nop
11)          end 2: local polling, enter the tail of the queue.
```

c)

Fig. 5. a) Microcode including V2P address translation, b) Microcode for memory access, and c) Microcode for synchronization

TABLE II summarizes the shared memory access performance. We use read transaction to illustrate the performance of the two types of memory access. If the address is local, the DMC performs local access. Otherwise, the DMC starts remote access. For a remote read transaction (T_{rss} and T_{rsb}, α=1), its delay consists of seven parts: (1) V2P translation latency: T_{v2p}= 13 cycles (T_f+11), (2) latency of distinguishing whether the read is local or remote: T_d=2 cycles, (3) latency of launching a remote request message to the remote destination node: T_m=2 cycles, (4) communication latency: T_{com} = T_{csd} (from source to destination) + T_{cds} (from destination to source), including network delivery latency for the request and waiting time for being processed by the mini-processor B of the destination DMC, (5) latency of filling the pipeline at the beginning of microcode execution: T_f=2 cycles, (6) latency of branching where the memory read microcode is: T_b=2 cycles, and (7) latency of executing the memory read microcode: 3 cycles for single read and 1+2*(n_b+1)+1 cycles for burst read of n_b words. (1), (2) and (3) are in the mini-processor A of the source DMC, while (5), (6) and (7) are in the mini-processor B of the destination DMC. To facilitate discussions in section VI, we merge (2), (3), (5), (6) and (7) into one part, calling it $T_{MemAcc\_without\_v2p}$, which is the time for executing the memory access microcode excluding the V2P translation time.

C. Synchronization

The Synchronization Supporter provides underlying hardware support for synchronization. It works with a pair of special

TABLE II
TIME CALCULATION OF MEMORY ACCESS AND SYNCHRONIZATION

V2P Translation: T_{v2p} = T_f + 11 [ ] in the mini-processor A [ ] in the mini-processor B α = 1, memory read; α = 0, memory write

	Local Shared	Remote Shared
Single MemAcc	T_{lss}=T_{v2p}+T_d+T_b+3	T_{rss}=T_{v2p}+T_d+T_m+T_f+T_b+3+T_{csd}+α*T_{cds}
Burst MemAcc	T_{lsb}=T_{v2p}+T_d+T_b+1+2*(n_b+1)+1	T_{rsb}=T_{v2p}+T_d+T_m+T_f+T_b+1+2*(n_b+1)+1+T_{csd}+α*T_{cds}
Sync.	T_{sync_l}=T_{v2p}+T_d+T_b+8*n_l	T_{sync_r}=T_{v2p}+T_d+T_m+(T_f+T_b+8)*n_l+T_{csd}+T_{cds}

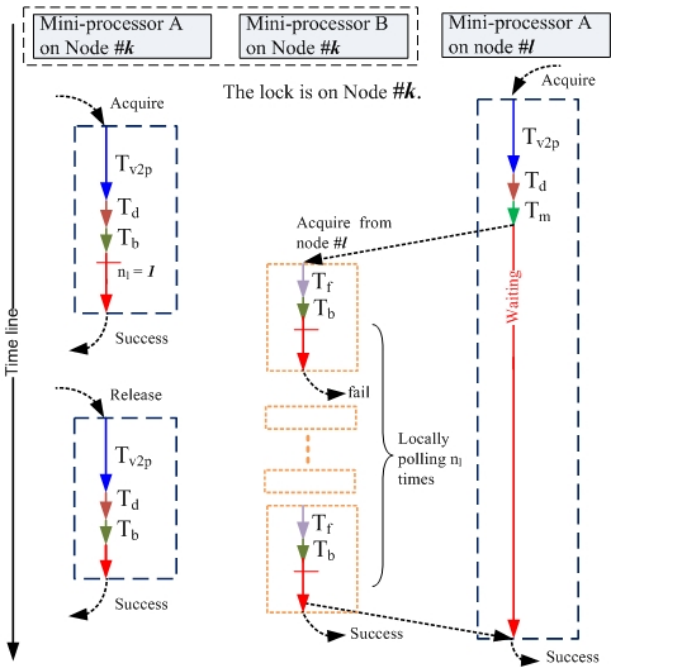

Fig. 6. Examples of synchronization transactions

microoperations (**ll** and **sc**) to guarantee atomic operation. Based on them, various synchronization primitives can be built. We implement a synchronization primitive: *test-and-set()*, as shown in Fig. 5 c). If an acquire of lock fails, the related command will be placed to the tail of the command queue in the CICU/NICU to wait for the next execution. This avoids incurring additional network traffic and won't block other commands for a long time.

TABLE II lists the synchronization performance. Synchronization is categorized into two types: (1) Local shared; (2) Remote shared. For acquiring a remote lock, its delay (T_{sync_r}) consists of seven parts (similar with shared memory access): (1) T_{v2p}, (2) T_d, (3) T_m, (4) T_{com} = T_{csd} + T_{cds}, (5) T_f, (6) T_b, (7) latency of executing *test-and-set()*, 8 cycles. The (5), (6) and (7) are multiplied by the acquire times, n_l. We also merge (2), (3), (5), (6) and (7) into one part, calling it $T_{Sync\_without\_v2p}$, which is the time for executing *test-and-set()* excluding the V2P translation time.

To further analyze the DMC performance, we choose synchronization as an example to illustrate its execution procedure in Fig. 6. Assume that the lock is on node ***#k***. As we can see, the mini-processor A and B in node ***#k*** concurrently deal with lock acquire commands from the local node and the remote node, respectively. The mini-processor A acquires the lock previously, so the mini-processor B fails. The command re-enters into the command queue in the NICU in node ***#k***. Since there are no other commands in the queue, the mini-processor B is activated again by this command to acquire the lock again. This procedure continues until the mini-processor A in node ***#k*** accepts the release command to release the lock. Then, the acquire of the lock by node ***#l*** succeeds and the success message is returned to node ***#l***.

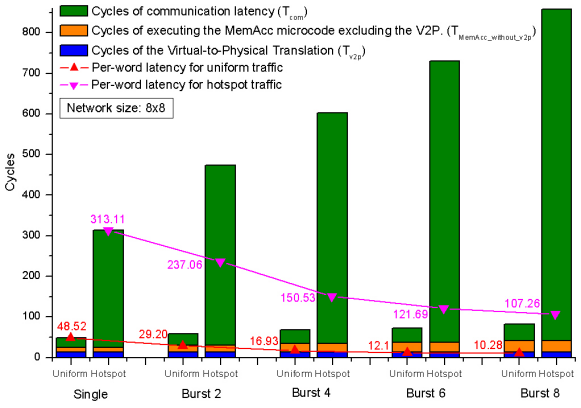

Fig. 7. Average read transaction latency for uniform and hotspot traffic

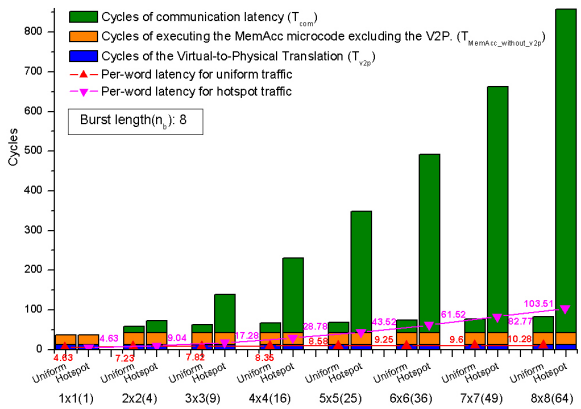

Fig. 8. Burst read latency under uniform and hotspot traffic

VI. Experiments and Results

We performed experiments to evaluate the DMC in terms of execution overhead in a multi-core NoC platform, applying both synthetic and application workloads.

A. Experimental Platform

We constructed a DSM based multi-core NoC experimental platform as shown in Fig. 1. The multi-core NoC has a mesh topology and its size is configurable. The network performs dimension-order XY routing, provides best-effort service and also guarantees in-order packet delivery. Moving one hop in the network takes one cycle. In all experiments, commands' corresponding microcodes have already uploaded into the Control Store.

B. Simulation Results with Synthetic Workloads

We first apply two synthetic workloads: *uniform* and *hotspot*.

Shared Memory Access

Since reads are usually more critical than writes, we use read transactions for all traffic. For a read with n_b words, one request is sent from the source to read n_b words from the destination. For uniform traffic, a node sends read requests to all other nodes one by one. Initially all nodes send requests at the same time. A new request will not be launched until the previous transaction is completed. For hotspot traffic, a corner node (0, 0) is selected as the hot spot node. All other nodes send read requests to the hotspot node. Simulation stops after all reads are completed.

Fig. 7 illustrates the effect of transaction size. It plots the average read transaction latency for uniform and hotspot traffic versus burst length in a 8x8 mesh multi-core NoC. The burst length varies from 1, 2, 4, 6 to 8 words. For the same transaction size, the overhead of $T_{MemAcc\_without\_v2p}$ is the same. For the single reads, the DMC overhead T_{DMC} (T_{DMC} = T_{v2p} + $T_{MemAcc\_without\_v2p}$) equals to 24 cycles (13 + 11). Under uniform traffic, the communication latency T_{com} is 24.52 cycles. So the total time T_{total} (= T_{com} + T_{DMC}) is 48.52 cycles (24 + 24.52). In this case, the DMC overhead is significant. However, under hotspot traffic, the network delivery time significantly increases because of increased contention in the network and waiting to be processed by the mini-processor B in the destination DMC. In this case, the DMC overhead is little. When increasing the transaction size, $T_{MemAcc\_without\_v2p}$ and T_{com} are increased, resulting in the increase of T_{total}. For all hotspot traffic, T_{com} dominates T_{total}. To compare the per-word latency (T_{total}/n_b), we draw two lines, one for uniform and the other for hotspot traffic. We can observe that, while increasing transaction size increases T_{total}, the per-word latency is decreasing for both uniform and hotspot traffic.

Fig. 8 illustrates the effect of network size. It plots burst read latency under uniform and hotspot traffic. With respect to the same transaction size (n_b = 8), the DMC overhead (T_{DMC}) of a remote read is a constant, 41 cycles (T_{v2p}=13, $T_{MemAcc\_without\_v2p}$=28) for different system sizes, while T_{DMC} of a local read for the single core is 37 cycles (T_{v2p}=13, $T_{MemAcc\_without\_v2p}$=24) since there is no microcode execution in the destination node. As the network size increases, T_{total} increases because the average communication distance increases. For uniform traffic, the increase in T_{total} is rather linear, and for hotspot traffic, the increase goes nearly exponentially. This is due to balanced workload in uniform traffic in contrast to centralized contention in hotspot traffic. We also plot the average per-word latency (T_{total}/n_b) for the two traffic types. The per-word latency for both traffics increases with the network size but much smoother. This suggests it is still advantageous to use larger transaction size, especially for larger size networks.

Synchronization

To experiment on synchronization latency, we use our microcoded *test-and-set()* primitive, which performs polling at the destination. For uniform traffic, all nodes start to acquire locks at the same time. After the acknowledgement (successful acquire) returns, each node sequentially acquires a lock in the next node following a predefined order. For hotspot traffic, all nodes try to acquire locks in the same node (0, 0). Simulation stops after all locks are acquired. Since locks in the same node acquired by different nodes can be the same or different, we distinguish the same lock and different locks for both uniform and hotspot traffic, resulting in 4 scenarios: (1) uniform, different locks, (2) hotspot, different locks, (3) uniform, same lock, and (4) hotspot, same lock.

Fig. 9 illustrates the effect of network size. It plots the synchronization latency for different network sizes under the four scenarios classified into Type A for different locks and Type B for the same lock. Note that, due to the huge latency for the hotspot cases, we use the Log10 scale for the Y-axis. The DMC overhead (T_{DMC}) is a constant, 29 cycles (T_{v2p}=13, $T_{Sync\_without\_v2p}$=16) for different system sizes, while T_{DMC} for the single core is 25 cycles (T_{v2p}=13, $T_{Sync\_without\_v2p}$=12) since there is no microcode execution in the destination node. We can observe that: (1) As the network size is increased, the DMC overhead is gradually diluted; (2) As expected, the synchronization latency acquiring the same lock (Type B) creates more contention and thus more blocking time for all cases than acquiring the different locks (Type A).

C. Simulation Results with Application Workloads

Besides using synthetic workloads, two applications, matrix multiplication and 2D radix-2 DIT FFT, are mapped manually

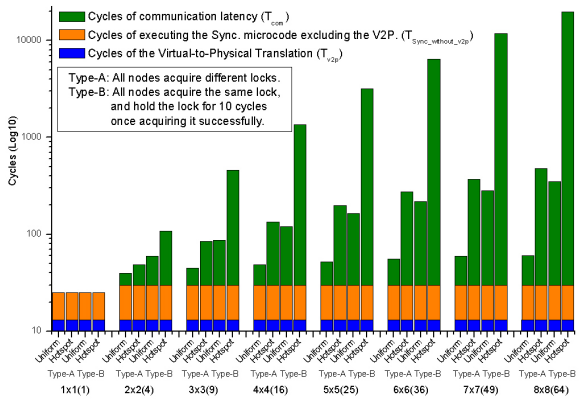

Fig. 9. Synchronization latency under uniform and hotspot traffic

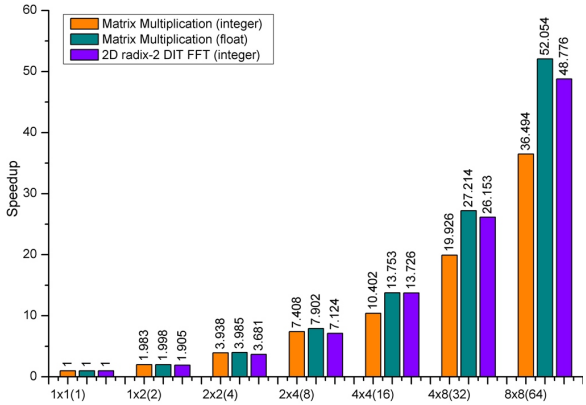

Fig. 10. Speedup of matrix multiplication and 2D radix-2 DIT FFT

over the LEON3 processors to evaluate the DMC performance. The matrix multiplication calculates the product of two matrices, $A[64,1]$ and $B[1,64]$, resulting in a $C[64,64]$ matrice and doesn't involve synchronization. We consider both integer and floating point matrix multiplication. The data of the 2D radix-2 DIT FFT are equally partitioned into n rows storing on n nodes respectively. The 2D FFT application performs 1D FFT of all rows firstly and then does 1D FFT of all columns. There is a synchronization point between the FFT-on-rows and the following FFT-on-columns.

Fig. 10 shows the performance speedup of the matrix multiplication and the 2D radix-2 DIT FFT. From this figure, we can see that the multi-core NoC achieves fairly good speedup. When the system size increases, the speedup ($\Omega_m = T_{1core}/T_{mcore}$, where T_{1core} is the single core execution time as the baseline, T_{mcore} the execution time of m core(s).) goes up from 1 to 36.494 for the integer matrix multiplication, from 1 to 52.054 for the floating point matrix multiplication, and from 1 to 48.776 for the 2D radix-2 DIT FFT. To make the comparison fair, we calculate the per-node speedup by Ω_m/m. As the system size increases, the per-node speedup decreases from 1 to 0.57 for the integer matrix multiplication, from 1 to 0.813 for the floating point matrix multiplication, and from 1 to 0.76 for the 2D radix-2 DIT FFT. This means that, as the system size increases, the speedup acceleration is slowing down. This is due to that the communication latency goes up nonlinearly with the system size, limiting the performance. We can also see that the speedup for the floating point matrix multiplication is higher than that for the integer matrix multiplication. This is as expected, because, when increasing the computation time, the portion of communication delay becomes less significant, thus achieving higher speedup.

VII. Conclusion

In this paper, we propose a microcoded controller as a central hardware engine for managing DSMs in multi-core NoCs. The controller is programmable and the support for DSM functions is implemented in microcode. The operation of the controller is triggered by requests from local and remote cores. In particular, it features two mini-processors in order to be able to serve requests from local and remote requests at the same time. Besides its architecture, we detail its operation mechanism and give examples. Our experiments on uniform and hotspot workloads suggest that the overhead of the controller may become insignificant as the system size increases and the communication delay becomes dominating performance. Our experiments on application workloads show that our multi-core NoCs (with the controller in each node) achieves good performance speedup with increasing system size. Moreover, our synthesis results show the controller runs up to 455 MHz and consumes 51k gates in a 130 nm technology. Therefore, we can conclude that the DMC is a viable approach providing an integrated, modular and flexible solution for addressing the DSM issues in multi-core NoCs.

Acknowledgment

The research is partially supported by the FP7 EU project MOSART (No. IST-215244), the National 863 Program of China (No. 2007AA01Z108), the Innovative Team of High-performance Microprocessor Technology, and the National Natural Science Foundation of China (No. 60676010).

References

[1] M. Horowitz and W. Dally, "How scaling will change processor architecture," in *Int'l Solid-State Circuits Conf. (ISSCC'04), Digest of Technical Papers*, Feb. 2004, pp. 132–133.

[2] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. of the 44th Design Automation Conf. (DAC'07)*, Jun. 2007, pp. 746–749.

[3] A. Jantsch and H. Tenhunen, *Networks on chip*. Kluwer Academic Publishers, 2003.

[4] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comp. Surveys*, vol. 38, no. 1, pp. 1–51, Mar. 2006.

[5] J. D. Owens, W. J. Dally *et al.*, "Research challenges for on-chip interconnection networks," *IEEE MICRO*, vol. 27, no. 5, pp. 96–108, Oct. 2007.

[6] S. Vangal, J. Howard, G. Ruhl *et al.*, "An 80-tile 1.28tflops network-on-chip in 65nm cmos," in *Int'l Solid-State Circuits Conf. (ISSCC'07), Digest of Technical Papers*, Feb. 2007, pp. 98–100.

[7] "Itrs 2007 document," in *http://www.itrs.net/Links/2007ITRS/Home2007.htm*.

[8] E. Marinissen, B. Prince, D. Keltel-Schulz, and Y. Zorian, "Challenges in embedded memory design and test," in *Proc. of Design, Automation and Test in Europe Conf. (DATE'05)*, Mar. 2005, pp. 722–727.

[9] M. V. Wilkes, "The best way to design an automatic calculating machine," in *Proc. of Manchester Univ. Computer Inaugural Conf.*, Jul. 1951, pp. 16–18.

[10] S. Vassiliadis, S. Wong, and S.Cotofana, "Microcode processing: Positioning and directions," *IEEE MICRO*, vol. 23, no. 4, pp. 21–30, Apr. 2003.

[11] A. Agarwal, R. Bianchini *et al.*, "The mit alewife machine: architecture and performance," in *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA 1995)*, Jun. 1995, pp. 2–13.

[12] J. Kuskin, D. Ofelt *et al.*, "The stanford flash multiprocessor," in *Proc. of the 21st Annual Int'l Symp. on Computer Architecture (ISCA 1994)*, Apr. 1994, pp. 302–313.

[13] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and typhoon: user-level shared memory," in *Proc. of the 21st Annual Int'l Symp. on Computer Architecture (ISCA 1994)*, Apr. 1994, pp. 325–336.

[14] M. Chaudhuri and M. Heinrich, "Smtp: an architecture for next-generation scalable multi-threading," in *Proc. of the 31st Annual Int'l Symp. on Computer Architecture (ISCA 2004)*, Jun. 2004, pp. 124–135.

[15] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. on Computer*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[16] "Leon3 processor," in *http://www.gaisler.com*.

[17] T. G. Rauscher and P. M. Adams, "Microprogramming: A tutorial and survey of recent developments," *IEEE Trans. on Computer*, vol. C-29, no. 1, pp. 2–20, Jan. 1980.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (4th Edition)*. Elsevier, Inc., 2007.