Command-Triggered Microcode Execution for Distributed Shared Memory Based Multi-Core Network-on-Chips

Xiaowen Chen[1, 2,*], Zhonghai Lu[2], Axel Jantsch[3], Shuming Chen[1], Yang Guo[1], Shenggang Chen[1], Hu Chen[1], Man Liao[1]

[1] College of Computer, National University of Defense Technology, 410073, Changsha, Hunan, China.
[2] Department of Electronic Systems, KTH-Royal Institute of Technology, 16440 Kista, Stockholm, Sweden.
[3] Institute of Computer Technology, Vienna University of Technology, 1040 Vienna, Austria.

* Corresponding author. Tel.: +86 135 0848 1483; email: xwchen@nudt.edu.cn
Manuscript submitted January 10, 2014; accepted March 28, 2014.

Abstract: Technology advance enables integration of a lot of resources on multi-core Network-on-Chips (NoCs). In such complex system, memories are preferably distributed and supporting Distributed Shared Memory (DSM) is essential for the sake of re-using huge amount of legacy code and easy programming. Besides, the design complexity of multi-core NoCs results in long time-to-market and high cost. Motivated by these two considerations, we propose a hardware/software co-design, called "command-triggered microcode execution". It guides a Dual Microcode Controller (DMC) to look for a flexible DSM support under multi-core NoCs. This paper describes hardware/software co-operation, command type, work model, work flow and microprogramming development flow in our proposed hardware/software co-design. Microcodes of basic DSM functions are implemented and their performance evaluations are discussed. Experimental results show that, when the system size is scaled up, the delay overhead incurred by the DMC may become less significant in comparison to the network delay. In this way, the delay efficiency of our hardware/software co-design is close to that of hardware solutions on average but our co-design still has all the flexibility of software solutions.

Key words: Command, microcode, microprogramming, hardware/software co-design, distributed shared memory, multi-core network-on-chips.

1. Introduction and Motivation

The rapid development of integrated circuit and computer architecture technology enables more and more computing resources and storage elements to be integrated on a single chip [1]. It is a trend that the high-performance single-chip computing architecture is evolving from single-core to multi- and many cores [2], [3]. Network-on-Chip (NoC) [4]-[6] 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. Another trend is that, due to technology advance, 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 grow in the future [7]. Technologies such as high density embedded memory (e.g. Z-RAM from Innovative Silicon [8]) and 3D integration, high bandwidth and parallel access to memory are becoming feasible and preferable. For instance, G. Loh proposes a 3D-stacked memory architecture where each core has its own memory bank with significant performance gains [9]. It's convinced that such many on-chip 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. 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. 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 re-design 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 [10]. Its crucial feature offers a programmable and flexible solution to accelerate a wide range of applications [11].

Along the aforementioned consideration, we propose a hardware/software co-design, called "command-triggered microcode execution". It guides a Dual Microcode Controller (DMC) [12] to look for a flexible DSM support under multi-core NoCs, aiming for the performance of hardware solutions but maintaining the flexibility of software solutions. The DMC is a programmable coprocessor, flexibly supporting various functions implemented in microcode. Functions are triggered by requests from main processors in form of command. In this paper, our proposal shows an entire hardware/software co-design flow targeting DSM based multi-core NoCs with such DMC modules. It describes in detail hardware/software co-operation, command type, work model, work flow as well as microprogramming development flow. As microcode examples, basic DSM functions (Virtual-to-Physical address translation, shared memory access and synchronization) are implemented by following the proposed hardware/software co-design flow. Performance analysis and experimental 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 hardware/software co-design is close to that of hardware solutions on average but the co-design still has all the flexibility of software solutions.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes our DSM based multi-core NoC architecture with DMC modules. In Section 4, we present the hardware/software co-design: Command-triggered Microcode Execution in detail. Section 5 evaluates the performance of the proposed hardware/software co-design by realizing basic DSM functions in microcode. Section 6 shows the application experiment results. Finally we conclude in Section 7.

2. Background and Related Work

The crucial feature of microcode approach offers a configurable and programmable solution to accelerate a wide range of applications [11]. It's a good alternative to overcome the performance-flexibility dilemma. We note that up to today there are few researches on using the microcode approach as the basis of hardware/software co-design method to support DSM, especially on multicore interconnected computing chips and systems. The Alewife [13] 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. Both the Stanford FLASH [14] and the Wisconsin Typhoon [15] 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 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. 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. The SMTp [16] 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 and hence deepens the burden of the main processor.

The microcode execution mechanism of the three researches above [14]-[16] are different with ours. In our hardware/software co-design, functions implemented in microcode are corresponded to a certain command. Command-triggered microcode execution enhances coupling degree of the hardware/software co-operation. Similarly, in [17], [18], Schmidt also uses the concept of command to define his message frames. The message frames make his programmable controller to respond requests from external devices. However, his inventions have nothing to do with supporting DSM and his commands are only used in the networks to enable dialogue with external devices. In our design, commands are coupled with their corresponding microcodes. This coupling is part of our proposed hardware/software co-design and supported by necessary hardware.

3. The Multi-Core NoC Architecture

Following [12], this section presents our Distributed Shared Memory (DSM) based multi-core Network-on-Chip (NoC) architecture with Dual Microcoded Controllers (DMCs).

3.1. Distributed Shared Memory

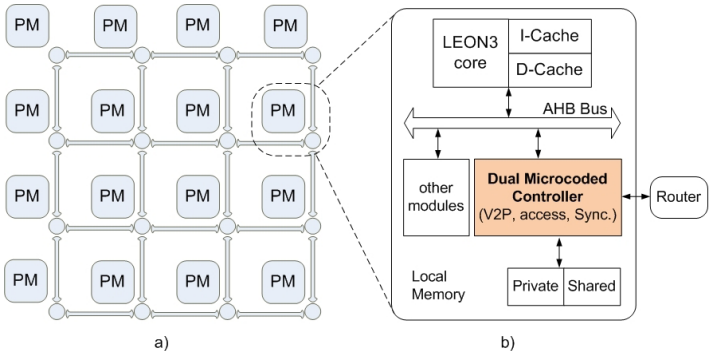

Fig. 1. a) A 16-node mesh multi-core NoC, b) processor-memory node.

In our Multi-core NoC, memories are distributed at network nodes but shared. Fig. 1. a) shows an example of our McNoC architecture with DSM. The system is composed of 16 Processor-Memory (PM) nodes interconnected via a packet-switched mesh network, which is a most popular NoC topology proposed today [19]. Note that a node can also be a memory node without a processor. As shown in Fig. 1. b), each PM node contains a processor, for example, a LEON3 [20], hardware modules connected to the local bus, and a local memory. The module, which connects the processor, the local memory and the network, is the Dual

Microcoded Controller (DMC), which features two microcoded controllers that can simultaneously serve requests from the local core and the remote cores via the network.

In our memory organization, we do not treat all memories as shared, though all local memories can logically form a single global memory address space. As illustrated in Fig. 1. b), the local memory is partitioned into two parts: private and shared. Two addressing schemes are introduced: physical addressing and logic (virtual) addressing. The private memory can only be accessed by the local core using physical addressing. All of shared memories are visible to all nodes and organized as a DSM addressing space and they are virtual. The philosophy of this organization is to speed up frequent private accesses 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 the DSM organization transparent to application, thus facilitating software programming.

3.2. Dual Microcoded Controller

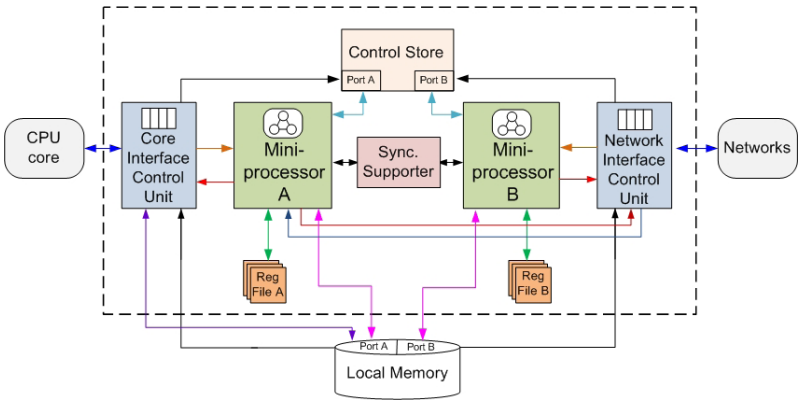

Fig. 2. Architecture of the dual microcoded controller.

As shown in Fig. 2, a DMC mainly consists of six components, 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 offers a hardware interface to the network. The two interface units receive commands from and sent replies to the local and remote cores. The two mini-processors, which are the core of the DMC, are the central processing engine. 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. 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 DMC is microcoded because of the two programmable mini-processors. The execution of the miniprocessors is triggered by requests (in the form of command) sent from the local and remote cores through the two interface units, specifically, local requests through the CICU and remote requests through the NICU. The existence of two mini-processors increases cost but enhances performance since it enables to concurrently process requests from the local and remote cores. If there is only one mini-processor, the essence of the DMC does not change but the processing of simultaneous requests has to be serialized. 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 microprogram is a set of microcodes.

Microprogram is initially stored in the Local Memory and will be dynamically uploaded into the Control Store and then be executed in the miniprocessors on the demand of commands during the program execution. Synthesis results show that the implementation of the DMC design can run up to 455 MHz consuming 51K gates (without counting the area of the Control Store) in 0.13 m technology.

4. Hardware/Software Co-design: Command-Triggered Microcode Execution

In this section, we present the detail of our proposed hardware/software co-design.

4.1. Coupled Hardware/Software Co-operation

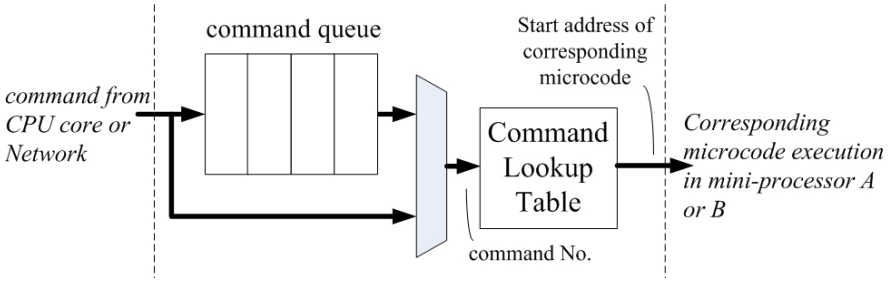

Fig. 3. Coupled hardware/software co-operation.

As shown in Fig. 2, the two interface units are coupled with their corresponding mini-processors (the CICU ⇔ the mini-processor A; the NICU ⇔ the mini-processor B) to support the command-triggered microcode execution. The two interface units are pure hardware modules responsible for receiving commands from the local CPU core and remote cores via the on-chip network, respectively, and then triggering the execution of the two mini-processors. The two mini-processor are microprogrammable. Fig. 3 illustrates the coupled hardware/ software co-operation. As shown in the figure, there is a command queue as well as a Command Lookup Table (CLT) in each interface unit. The command queue buffers commands from the CPU core or remote cores via the on-chip network, if the command queue isn't empty or the mini-processor is working. If both the command queue is empty and the mini-processor is idle, the command bypasses the command queue to reach the CLT directly.

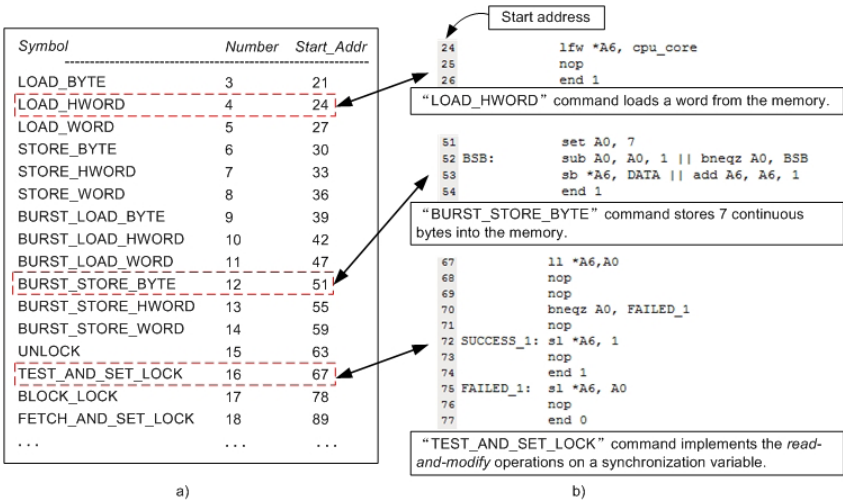

Fig. 4. a) Command Lookup Table (CLT) and b) Microcode segments.

The Command Lookup Table (CLT) reflects the correspondence of a command and a microcode. The CLT is indexed by the command to output the start address of the command's corresponding microcode. The start

address is forwarded to the mini-processor, so the miniprocessor is able to know where the current microcode execution starts. Fig. 4. a) shows how the CLT looks like. The "Symbol" is mnemonic. The command "Number" has a one-to-one correspondence with the "Start_Addr" of the related microcode. Fig. 4. a) lists several commands we have implemented and Fig. 4. b) illustrates the snapshots of three microcodes. As we can see, "LOAD HWORD" command with its command No. of 4 is responsible for loading a half word from the local memory. The start address of its related microcode is 24, so in the CLT we have an item recording the relationship between "LOAD HWORD" command and its microcode. It's the same for "BURST STORE BYTE" command which stores 7 continuous byte data into the local memory. Its command No. and start address are 12 and 51, respectively. "TEST AND SET LOCK" command implements the read-and-modify operations to guarantee mutex access on a synchronization variable. The CLT stores its command No. (16) and start address (67).

From Fig. 4. a), we can see the command No. starts from 3 not 1. This is because Command No. 1 and 2 are two special commands (see below).

4.2. Command Type

We define two types of commands:

i) Special Command

There are two kinds of special commands. The one, named "CLT command" with command No. of 1, is used to update the Command Lookup Table. Users can use this command to add, modify or delete a command-microcode correspondence in the CLT. The other, named "MDL command" with command No. of 2, is used to upload a microcode from the Local Memory to the Control Store. These two special commands are not stored in the CLT and are usually used together.

ii) Generic Command

A generic command is user-defined to implement a function. Users use "CLT command" to add a new generic command into the CLT and use "MDL command" to upload the corresponding microcode from the Local Memory to the Control Store. The correspondence between a generic command and its microcode is stored in the CLT. The command No. of a generic command is equal to or greater than 3.

4.3. Work Model

The behavior of command-triggered microcode execution is categorized into two kinds of work model.

1): If the corresponding microcode is in the Control Store.

During the run-time, the two interface units respond to commands from CPU core or remote cores via the on-chip network and then trigger mini-processors to execute the corresponding microcodes.

2): If the corresponding microcode isn't in the Control Store.

Due to the limited size of the Control Store, a few microcodes can be stored in it. To some extent, this limitation causes inconvenience of software development. Therefore, our hardware/software co-design provide users with the support of adding a new microcode into the Control Store and deleting an old microcode out of the Control Store during the system is running. During the situation that the command responded to by two interface units is not in the CLT (that also means the corresponding microcode isn't in the Control Store), (1) if the Control Store have enough space to store the corresponding microcode, "CLT command" is used to add this command into the CLT and "MDL command" is used to upload the corresponding microcode from the Local Memory to the unused space in the Control Store; (2) if the Control Store doesn't have enough space to store the corresponding microcode, one or more "CLT command" are used to delete one or more old commands out of the CLT in order to release enough space for the corresponding microcode, and then "CLT command" is used to add this command into the CLT and "MDL command" is used to upload the corresponding microcode from the Local Memory to the space in the

Control Store of those retired microcode related to the deleted commands.

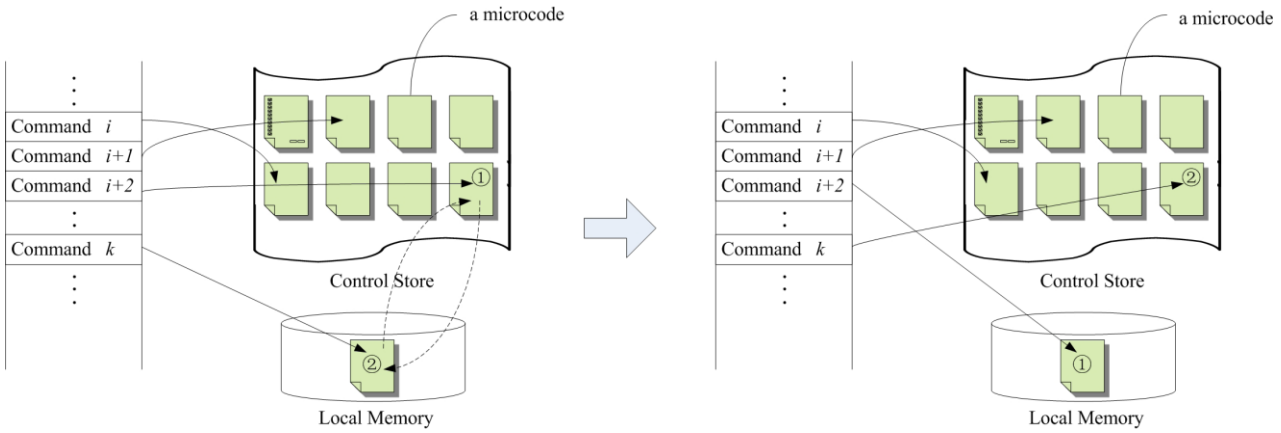

Fig. 5. Examples of command-triggered microcode execution.

Fig. 5 shows an example of the behavior of command triggered microcode execution. As we can see, each microcode is related to a function, which is implemented by a microcode. A microcode is a sequence of microinstructions with an "end" microoperation at the end. A microprogram is a set of microcodes. There are a number of commands. Each command is related to and triggers a microcode. As shown in the left part of Fig. 5, Command i is related to the leftmost microcode in the second line, Command i+1 is related to the second microcode in the first line, and Command i + 2 is related to the rightmost microcode in the second line. Users define their command and write the microcode of what function they want to implement. Then, they use "CLT command" and "MDL command" to update the CLT and transfer the microcode from the Local Memory to the Control Store.

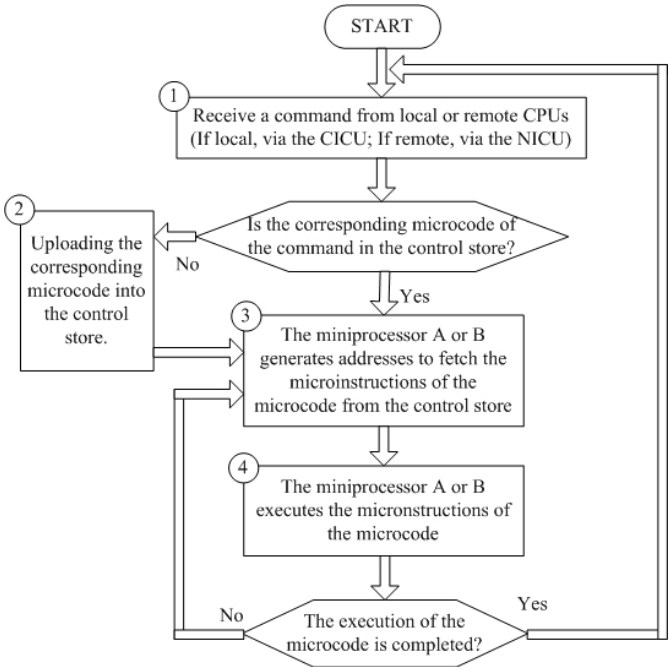

Fig. 6. Work flow.

Regarding the situation that the corresponding microcode isn't in the Control Store, as shown in the left part of Fig. 5, the microcode which is marked with 1) and related to Command i + 2 is now located in the

Control Store, while the microcode which is marked with 2) and related to Command k is located in the Local Memory. At this time, Command k will be used soon while Command i+2 won't be used in a near future. Thus, as shown in the bottom picture of Fig. 5, the microcode marked with 1) is replaced out of the Control Store while the microcode marked with 2) has a copy in the Control Store.

4.4. Work Flow

As illustrated in Fig. 6, the DMC works as follows (Microprogram is initially stored in the Local Memory):

1) The CICU/NICU receives a command from the local or a remote core.
2) A command will trigger the uploading of its microcode from the Local Memory to the Control Store. The Control Store has limited storage. If there is no space available when uploading the microcode to the Control Store, a replacement policy will be activated to replace a microcode with the currently activated one.
3) Then the mini-processor A (for commands from the local CPU core via the CICU) or B (for commands from other CPU cores via the NICU) will generate addresses to load the microinstruction from the Control Store to the datapath of the mini-processor.
4) The mini-processor A or B executes the microinstructions of the microcode.

This procedure is iterated over the entire execution period of the system.

4.5. DMC Library

```
/*==================*/
/* System Function */
/*==================*/
//Initialize the DMC when the entire program begins.
void DMC_initialization(void);
//Add or update a generic command.
void Write_CLT(unsigned int Command_No, unsigned int start_addr_in_Control_Store);
//Upload the microcodes from the Local Memory to the Control Store.
void Microcode_Uploading(void);
//Calculate the execution time of programs.
void time_cal_starts(void);
void time_cal_ends(void);

/*=========================*/
/* User-defined Functions */
/*=========================*/
//Correspond to the "TEST AND SET LOCK" command.
unsigned int test_and_set(volatile unsigned int * sync_addr);
//Correspond to the "UNLOCK" command.
void unlock(volatile unsigned int * sync_addr);
......
```

Fig. 7. DMC library.

As shown in Fig. 1, we adopt LEON3 as the processor core. To support programming under LEON3 software development environment, we provide a set of DMC library functions, as shown in Fig. 7. The DMC library bridges the DMC's microprogramming and the LEON3's programming. It contains two parts: System Functions and User-defined Functions. In the first part, **DMC initialization()** initializes the DMC when the entire program begins. **Write CLT()** is offered to users to add or update a generic command. **Microcode Uploading()** uploads the microcodes from the Local Memory to the Control Store. The last two system functions, **time_cal_starts()** and **time_cal_ends()**, allows users calculates the execution time of their programs. Users can define their own functions in the second part. Fig. 1 shows two examples: **test_and_set()** and **unlock()**. They correspond to the "TEST_AND_SET_LOCK" command and "UNLOCK" command, respectively.

4.6. Microprogramming Development Flow

Fig. 8 depicts the entire flow of microprogramming the DMC. The flow consists of 6 steps.

1) The user specifies the function to be implemented in the DMC.

2) A new generic command is defined according to the function specification.
3) A newly defined command is added into the CLT or an existing command is updated or removed by the "CLT command".

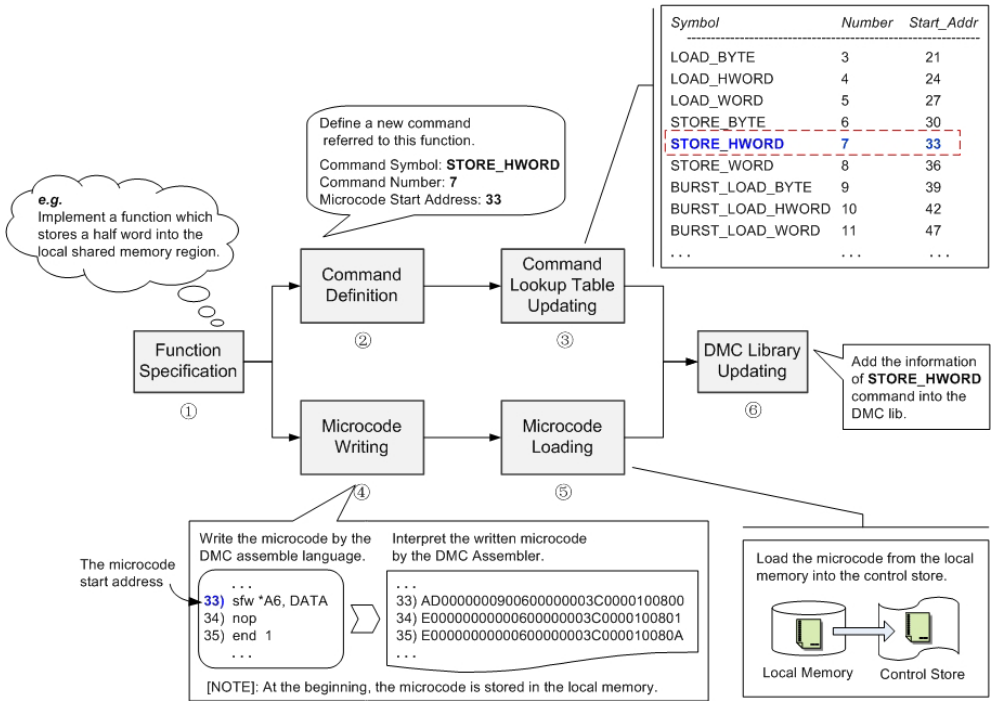

Fig. 8. Microprogramming development flow.

4) The user writes a microcode in the DMC assemble language for the specified function, then interpret it into executable binary code by the DMC assembler.
5) The binary microcode is uploaded from the Local Memory into the Control Store.
6) The DMC Library has to be updated in order for the newly defined command to be supported by the compiler and used by application programs.

Users can follow this flow iteratively to add, update and delete commands of different functions. For instance, as shown in Fig. 8, our function specification is implementing a function to store a half word into the Local Memory. In Step 2, we define a new generic command with Command Symbol of **STORE_HWORD**, Command No. of 7 and the start address of 55. The start address is obtained in Step 4 after the corresponding microcode is written and interpreted. In Step 3, the newly defined command is added into the Command Lookup Table (see the red dashed box) using "CLT command". The binary code of the newly written microcode is uploaded from the Local Memory to the Control Store using "MDL command". Finally, write a user-defined function to add the information of **STORE_HWORD** into the DMC Library. Afterwards, the command **STORE_HWORD** can be used successfully.

5. Microcode Examples and Performance Evaluations

To evaluate the hardware/software co-design, we used microcode to implement basic DSM functions: V2P address translation, shared memory access and synchronization. Performance analysis is performed. We also performed experiments to evaluate the proposal in terms of execution overhead in a multi-core NoC 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. Besides, moving one hop in

the network takes one cycle. Finally, two applications are performed. In all experiments, commands' corresponding microcodes have already uploaded into the Control Store.

```
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
      set A0, n
BLW: sub A0, A0, 1 || bneqz A0, BLW
      lw *A6, cpu_core || add A6, A6, 4
      end 1
```

```
;BURST_STORE_WORD
      set A0, n                  ;set the burst number
BSW: sub A0, A0, 1 || bneqz A0, BSW
      sw *A6, DATA || add A6, A6, 4   ;use the branch slot
      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. 9. a) Microcode including V2P address translation, b) Microcode for memory access, and c) Microcode for synchronization.

5.1. Microcode Example 1: V2P Address Translation

5.1.1. Microcode implementation

To maintain a Distributed Shared Memory environment, each time the 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. 9. 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 [21] 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. Since the V2P translation is always executed before the Shared Memory Access (see Example 2 in Subsection V-B) and Synchronization (see Example 3 in Subsection V-C) microcodes, its performance analysis is combined with that of the Shared Memory Access and Synchronization microcodes.

Table 1. Time Calculation of Memory Access and Synchronization

V2P Translation: $T_{v2p} = T_f + 11$ in the mini-processor A in the mini-processor B $\alpha = 1$, memory read $\alpha = 0$, memory write

	Local Shared	Remote Shared
Single MemAcc	$T_{lss}=T_{v2p}+T_d+T_b+3$	$T_{rss}=T_{v2p}+T_d+T_m+T_f+T_b+3+T_{csd}+\alpha*T_{cds}$
Burst MemAcc	$T_{lsb}=T_{v2p}+T_d+T_b+1+2*(n_b+1)+1$	$T_{rsb}=T_{v2p}+T_d+T_m+T_f+T_b+1+2*(n_b+1)+1+T_{csd}+\alpha*T_{cds}$
Sync.	$T_{sync\_l}=T_{v2p}+T_d+T_b+8*n_l$	$T_{sync\_r}=T_{v2p}+T_d+T_m+(T_f+T_b+8)*n_l+T_{csd}+T_{cds}$

5.2. Microcode Example 2: Shared Memory Access

5.2.1. Microcode implementation

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, burst reads and writes are used as an example. Fig. 9. b) shows the microcode for memory read and write of a burstiness of n words.

5.2.2. Performance analysis

Table 1 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 (α=1), its delay (T_{rss} and T_{rsb}) 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) + Tcds (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 synthetic experiment results, 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.

Fig. 10 illustrates the execution and time of shared single and bursty read memory accesses. In this example, the three nodes, #k, #l, and #m execute concurrently. Node #k first performs a local bursty read followed by a local single read. Meanwhile, node #l performs a remote single read from node #k, and node #m a remote bursty read from node #k. As shown in the figure, the two miniprocessors A and B in node #k deal with local and remote read requests concurrently. The mini-processor B in node #k first handles the single read request from node #l, and then the burst read from node #m.

5.2.3. Synthetic experiment results

Since reads are usually more critical than writes, we use read transactions for all traffic. For a read with nb 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.

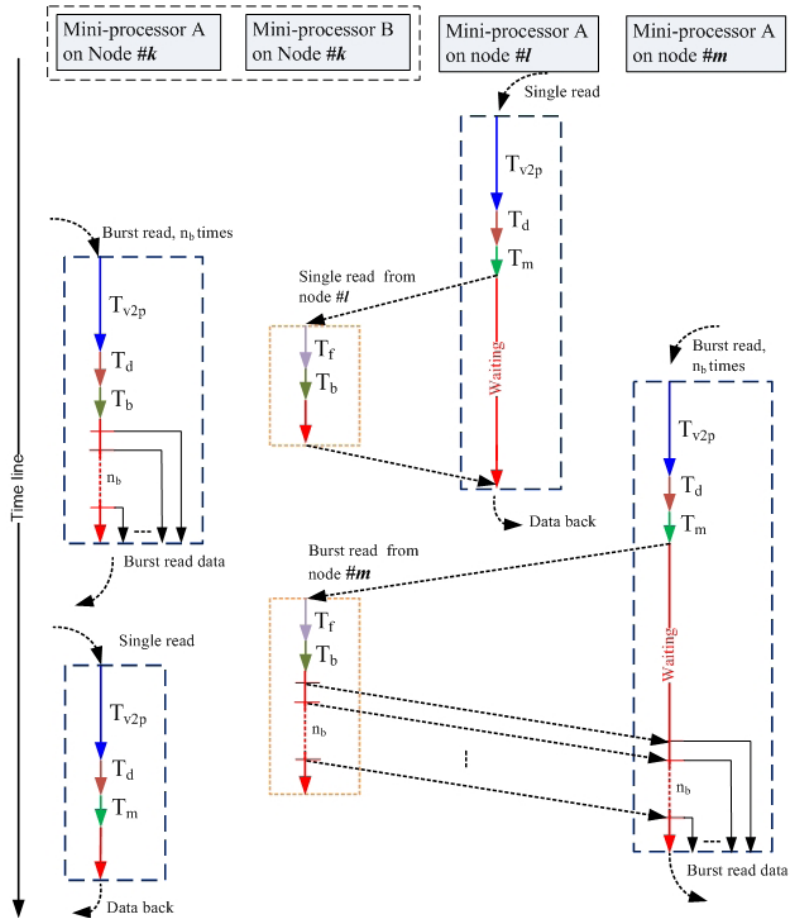

Fig. 10. Examples of single and bursty reads.

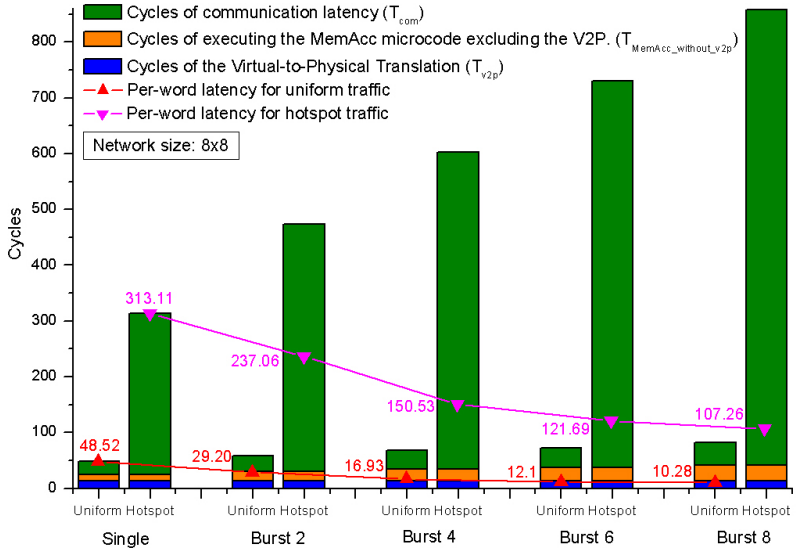

Fig. 11. Average read transaction latency for uniform and hotspot traffic.

Fig. 11 illustrates the effect of transaction size. It plots the average read transaction latency for uniform and hotspot traffic versus burst length in a 8×8 mesh multicore 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 TDMC (T_{DMC} = Tv2p + $T_{memAcc\_without\_v2p}$) equals to 24 cycles (13 + 11). Under uniform traffic, the communication latency Tcom is 24.52 cycles. So the total time Ttotal (= Tcom + 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 Tcom 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.

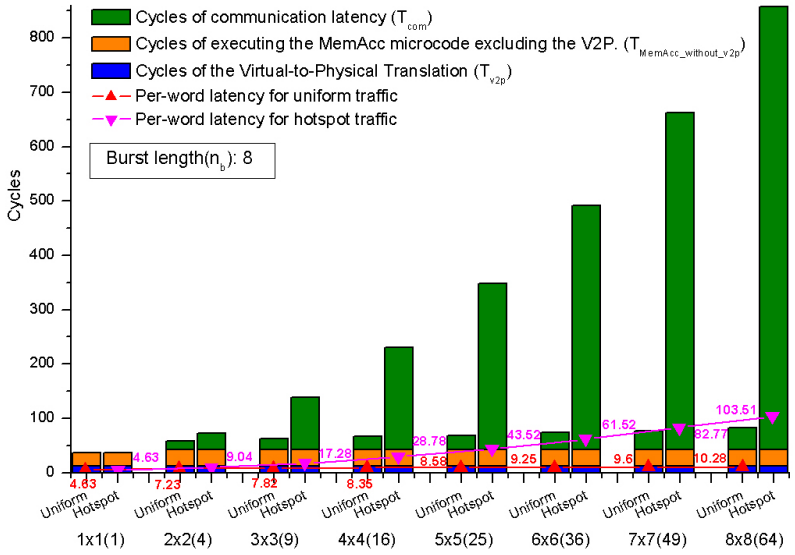

Fig. 12. Burst read latency under uniform and hotspot traffic.

Fig. 12 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 (TDMC) of a remote read is a constant, 41 cycles (T_{v2p}=13, $T_{memAcc\_without\_v2p}$=28) for different system sizes, while TDMC 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, Ttotal increases because the average communication distance increases. For uniform traffic, the increase in Ttotal 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.

5.3. Microcode Example 3: Synchronization

5.3.1. Microcode implementation

The Synchronization Supporter provides underlying hardware support for synchronization. It works with a pair of special 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. 9 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.

5.3.2. Performance analysis

Table 1 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) 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 *test-and-set()*: 8 cycles. The (5), (6) and (7) are multiplied by the acquire times: nl. 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.

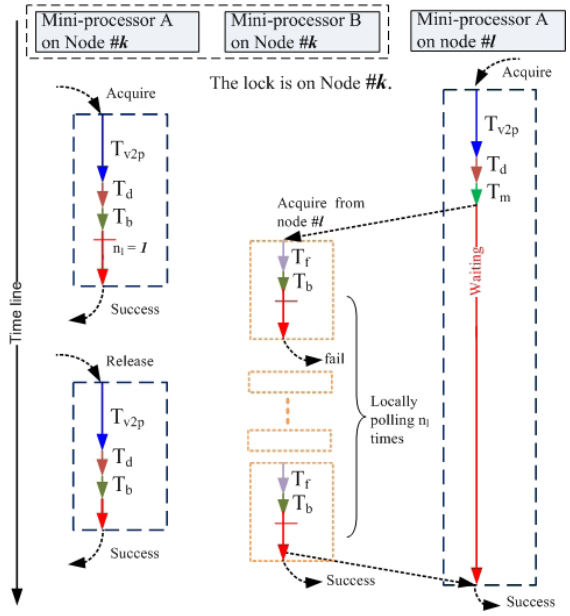

Fig. 13. Examples of synchronization transactions.

Fig. 13 illustrates synchronization execution procedure. 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.

5.3.3. Synthetic experiment results

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.

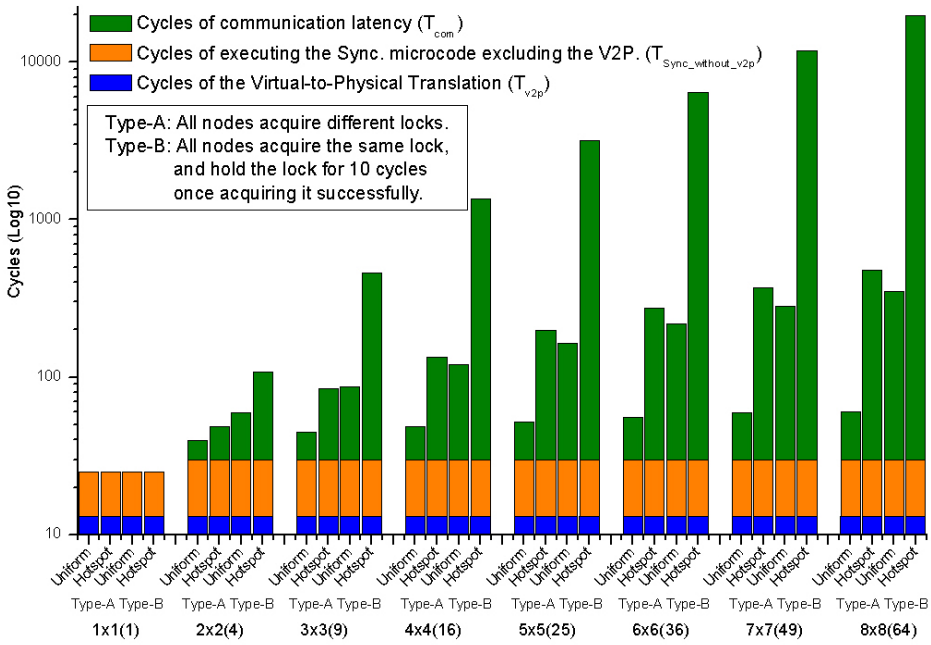

Fig. 14. Synchronization latency under uniform and hotspot traffic.

Fig. 14 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).

6. Application Experiments

In this section, we map three applications, matrix multiplication 2D radix-2 DIT FFT and Wavefront Computation, manually over the LEON3 processors, based on our proposed hardware/software co-design flow. 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 FFTon- columns. Wavefront Computations are common in scientific applications. In Wavefront Computation, the computation of each matrix element depends on its neighbors to the left, above, and above-left. If the solution is computed in parallel, the computation at any instant form a wavefront propagating toward in the solution space. Therefore, this form of computation get its name as wavefront.

Taking the code segment of matrix multiplication as an example, Fig. 15 illustrates how to write a C code based on our proposed hardware/software co-design flow. From the figure, we can see that, the program firstly initializes the DMC by invoking **DMC_initialization()** followed by **Write_CLT()** functions. After using **Microcode_Uploading()** function to upload the microcode from the Local Memory to the Control Store, the computation of matrix multiplication begins.

```
#include "DMC.h"
......
main()
{
......
    //==================================
    // DMC initialization
    //==================================
    DMC_initialization();
    //==================================
    // write Command Lookup Table
    //==================================
    Write_CLT( 3,21); //LOAD_BYTE
    Write_CLT( 4,24); //LOAD_HWORD
    Write_CLT( 5,27); //LOAD_WORD
    Write_CLT( 6,30); //STORE_BYTE
    Write_CLT( 7,33); //STORE_HWORD
    Write_CLT( 8,36); //STORE_WORD
......
    //==================================
    // Microcode dynamic loading
    //==================================
    Microcode_Uploading();
......
    time_cal_starts();

    for(i=M*DATA_LOCATION/NODE_NUM;i<M*(DATA_LOCATION+1)/NODE_NUM;i++) {
      flag = N/NODE_NUM;
      b = b_ini_2;
      for(k=0;k<N;k++) {
        if((flag-k)==0) {b = b + (0x20000 >> 2); flag = flag + N/NODE_NUM;}
        for(j=0;j<P;j++)
          *(c+i*N+k) = *(c+i*N+k) + (*(a+i*P+j))*(*(b+j*N+k));
      }  // c[i][k] = c[i][k] + a[i][j]*b[j][k];
    }

    time_cal_ends();

}
```

Fig. 15. Code segment of matrix multiplication.

Fig. 16 shows the performance speedup of the three applications. From this figure, we can see that the multicore 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 1.983, 3.938, 7.408, 10.402, 19.926 and 36.494 for the integer matrix multiplication, from 1 to 1.998, 3.985, 7.902, 13.753, 27.214 and 52.054 for the floating point matrix multiplication, from 1 to 1.905, 3.681, 7.124, 13.726, 26.153 and 48.776 for the 2D radix-2 DIT FFT, and from 1 to 1.815, 3.542, 6.735, 12.663, 19.735 and 21.539 for the Wavefront Computation. 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 0.992, 0.985, 0.926, 0.650, 0.623 and 0.570 for the integer matrix multiplication, from 1 to 0.999, 0.996, 0.988, 0.860, 0.850 and 0.813 for the floating point matrix multiplication, from 1 to 0.953, 0.920, 0.891, 0.858, 0.817 and 0.762 for the 2D radix-2 DIT FFT, and from 1 to 0.908, 0.886, 0.842, 0.791, 0.617 and 0.337 for the Wavefront Computation. 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. Wavefront Computation is synchronization-intensive. Its speedup increases more slowly than Matrix Multiplication and 2D DIT FFT, because synchronization overhead and communication delay become dominating as the network size is scaled up.

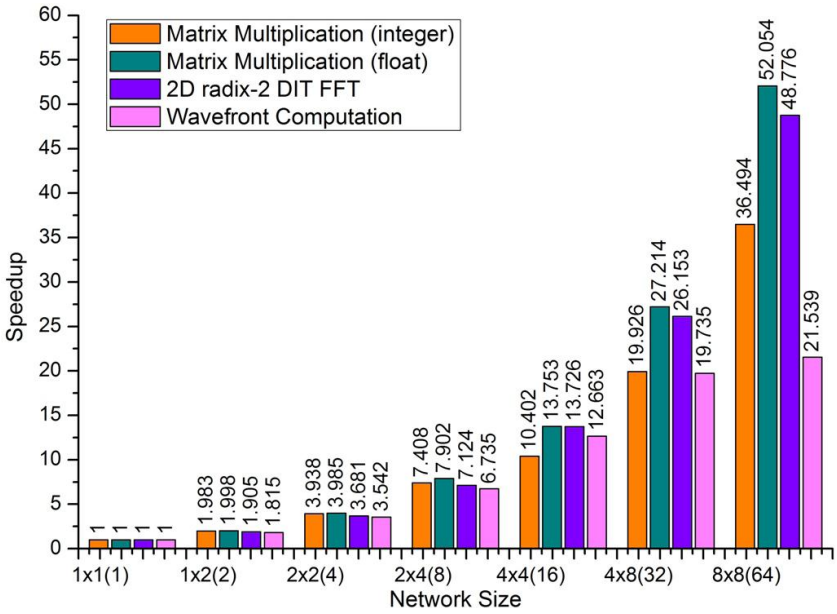

Fig. 16. Speedup of matrix multiplication and 2D radix-2 DIT FFT.

7. Concluding Remark

In multi-core Network-on-Chips (NoCs), memories are preferably distributed and it's essential to support Distributed Shared Memory (DSM) for the sake of re-using huge amount of legacy code and ease of programming. The design complexity of multi-core NoCs results in long time-to-market and high cost. Therefore, in this paper, we propose a hardware/software co-design, called "command-triggered microcode execution", in order to explore a flexible microcoded method to support DSM under multi-core NoCs. The "command-triggered microcode execution" guides a microcoded co-processor, named Dual Microcode Controller (DMC), in our multi-core NoC platform, purchasing the performance of hardware solutions but maintaining the flexibility of software solutions. The hardware/software co-design is fully described with its hardware/software co-operation, command type, work model, work flow as well as microprogramming development flow. We implemented basic DSM functions (Virtual-to-Physical address translation, shared memory access and synchronization) as microcode examples. Performance analysis and experimental results shows that, when the system size is scaled up, the delay overhead incurred by the controller may become less significant in comparison with the network delay. Application experiments show that our multi-core NoCs (with the controller in each node) achieves good performance speedup with increasing system size. Therefore, we can conclude that our hardware/software co-design is a viable way and its delay efficiency is close to hardware solutions on average but still have all the flexibility of software solutions.

Acknowledgment

The research is partially supported by the Hunan Natural Science Foundation of China (No. 2015JJ3017), and the Doctoral Program of the Ministry of Education in China (No. 20134307120034).

References

[1] International Technology Roadmap for Semiconductors. (2013). *ITRS 2013 Document*. Retrieved 2014, from http://www.itrs.net/Links/2013ITRS/Home2013.htm.

[2] Horowitz, M., & Dally, W. (2004). How scaling will change processor architecture. *Proceedings of the*

International Solid-State Circuits Conference (pp. 132-133).

[3] Borkar, S. (2007). Thousand core chips: A technology perspective. *Proceedings of the 44th Design Automation Conference* (pp. 746-749).

[4] Jantsch, A., & Tenhunen, H. (2003). *Networks on Chip*. Kluwer Academic Publishers.

[5] Bjerregaard, T., & Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computer Surveys, 38(1),* 1-51.

[6] Owens, J. D., & Dally, W. J. (2007). Research challenges for on-chip interconnection networks. *IEEE MICRO, 27(5),* 96-108.

[7] Marinissen, E., Prince, B., Keltel-Schulz, D., & Zorian, Y. (2005). Challenges in embedded memory design and test. *Proceedings of Design, Automation and Test in Europe Conference* (pp. 722-727).

[8] Innovative Silicon. (2010). *Z-RAM*. Retrieved from http://en.wikipedia.org/wiki/Z-RAM.

[9] Loh, G. (2008). 3D-stacked memory architectures for multi-core processors. *Proceedings of the 35th Annual International Symposium on Computer Architecture* (pp. 453-464).

[10] Wilkes, M. V. (1951). The best way to design an automatic calculating machine. *Proceedings of Manchester University Computer Conference* (pp. 16-18).

[11] Vassiliadis, S., Wong, S., & Cotofana, S. (2003). Microcode processing: Positioning and directions. *IEEE MICRO, 23(4),* 21-30.

[12] Chen, X., Lu, Z., Jantsch, A., & Chen, S. (2010). Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. *Proceedings of Design, Automation and Test in Europe Conference* (pp. 39-44).

[13] Agarwal, A. & Bianchini., R. (1995). The MIT alewife machine: Architecture and performance. *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (pp. 2-13).

[14] Kuskin, J. & Ofelt, D. (1994). The Stanford flash multiprocessor. *Proceedings of the 21st Annual International Symposium on Computer Architecture* (pp. 302-313).

[15] Reinhardt, S. K., Larus, J. R., & Wood, D. A. (1994). Tempest and typhoon: User-level shared memory. *Proceedings of the 21st Annual International Symposium on Computer Architecture* (pp. 325-336).

[16] Chaudhuri, M., & Heinrich, M, (2004). SMTP: An architecture for next-generation scalable multi-threading. *Proceedings of the 31st Annual International Symposium on Computer Architecture* (pp. 124-135).

[17] Schmidt, O. S. Programmable controller processor module having multiple program instruction. United States Patent (No. 5212631).

[18] Schmidt, O. S. Processor for a programmable controller. United States Patent (No. 5265005).

[19] Pande, P., Grecu, C., Jones, M., Ivanov, A., & Saleh, R. (2005). Performance evaluation and design tradeoffs for network-on-chip interconnect architectures. *IEEE Transactions on Computer, 54(8)*, 1025-1040.

[20] Aeroflex Gaisler (2013). *LEON3 Processor*. Retrieved from http://www.gaisler.htm.

[21] Hennessy, J. L., & Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach* (4th ed.). Elsevier Incorporation.

Xiaowen Chen received two B.S. degrees in microelectronics and computer science, respectively, from the University of Electronic Science and Technology of China (UESTC) in 2005, and received his PhD in microelectronics from National University of Defense Technology (NUDT).

Currently, he is a research assistant in microprocessor design. His research interests

include computer architecture, microarchitecture, VLSI design, system-on-chips, network-on-chips, distributed shared memory.

Zhonghai Lu received BSc. from Beijing Normal University, China in 1989. Since then he had worked extensively in industry for several electronic, communication and embedded systems companies as a system engineer and project manager for eleven years. Afterwards, he entered the Royal Institute of Technology (KTH), Sweden in 2000. From KTH, he received MSc. And PhD. in 2002 and 2007, respectively.

He is currently a researcher at KTH. Dr. Lu has published over 25 peer-reviewed technical papers in journals, book chapters and international conferences in the areas of networks/systems on chips, embedded real-time systems and communication networks. His research interests include computer systems and VLSI architectures, interconnection networks, system-level design and HW/SW co-design, reconfigurable and parallel computing, system modeling, refinement and synthesis, and design automation.

Axel Jantsch received a Dipl. Ing. (1988) and a Dr. Tech. (1992) degree from the Technical University Vienna. Between 1993 and 1995, he received the Alfred Schrdinger scholarship from the Austrian Science Foundation as a guest researcher at the Royal Institute of Technology (KTH). From 1995 through 1997, he was with Siemens Austria in Vienna as a system validation engineer. Since 1997, he is with the Royal Institute of Technology, Stockholm, Sweden.

Since December 2002, he is a full professor in electronic system design. A. Jantsch has published over 140 papers in international conferences and journals in the areas of VLSI design and synthesis, system level specification, modeling and validation, HW/ SW co-design and co-synthesis, reconfigurable computing and networks on chip. At the Royal Institute of Technology, A. Jantsch is heading a number of research projects, in the areas of system level specification, design, synthesis, validation and networks on chip.

Shuming Chen received the BSc., MSc. and PhD degrees from National University of Defense Technology (NUDT), Changsha, China, in 1982, 1988 and 1993, respectively. Since then he is with School of Computer, National University of Defense Technology (NUDT).

Currently, he is a full professor in microprocessor design. His research interests include processor architecture, high performance circuits, custom design for reliability, and SOC. S. Chen has published over 110 papers in conferences and journals in the area of microarchitecture, VLSI design, and digital signal processor. As a chief architect, he designed more than ten microprocessor chips in recent years.

Yang Guo received the BSc., MSc. and PhD degrees from National University of Defense Technology (NUDT), Changsha, China, in 1992, 1996 and 2001, respectively. Since then he is with School of Computer, National University of Defense Technology (NUDT).

Currently, he is a full professor in VLSI design and test. His research interests include microarchitecture, VLSI design and VLSI test methodology. He has published over 60 papers in conferences and journals in the area of microarchitecture, VLSI design, and VLSI test methodology.

Shenggang Chen received his the BSc., MSc. and PhD degrees from National University of Defense Technology (NUDT), Changsha, China, in 2004, 2006 and 2010, respectively. He works in the School of Computer, National University of Defense Technology (NUDT) from the beginning of the year 2011. His main research interests include VLSI design, digital signal processor and video encoding.

Hu Chen received the BSc., MSc. and PhD degrees from National University of Defense Technology (NUDT), Changsha, China, in 2004, 2006 and 2011, respectively. He works in the School of Computer, National University of Defense Technology (NUDT) from the beginning of the year of 2011.

His main research interests include high performance microprocessor design and video encoding.

Man Liao received the MSc. degree from Zhongnan University of Economics and Law (ZUEL), Changsha, China, in 2009, and 2012, respectively. She works in the School of Computer, National University of Defense Technology (NUDT) from 2011 June. Her main research interest is VLSI design.

.

.