Chapter 1
Memory Architecture and Management in an NoC Platform

Axel Jantsch, Xiaowen Chen, Abdul Naeem, Yuang Zhang, Sandro Penolazzi and Zhonghai Lu

Abstract The memory organization and the management of the memory space is a critical part of every NoC based platform design. We propose a Data Management Engine (DME), that is a block of programmable hardware and part of every processing element. It off-loads the processing element (CPU, DSP, etc.) by managing the memory space, memory access and the communication over the on-chip network. The DME's main functions are virtual address translation, private and shared memory management, cache coherence protocol, support for memory consistency models, synchronization and protection mechanisms for shared memory communication. The DME is fully programmable and configurable thus allowing for customized support for high level data management functions such as dynamic memory allocation and abstract data types. This chapter describes the main concepts, design and functionality of the DME and presents case studies illustrating its usage and performance.

Key words: Network on Chip, SoC Architecture, Memory Organization

1.1 On-Chip Memory Organization

On-chip Computation is moving away from a sequential to a parallel paradigm leading to dozens, hundreds, and soon even thousands of cores and computational units on a single die. These many core chips can be highly homogeneous or irregular and heterogeneous, depending on the application area and market segment. At the same time, the communication infrastructure is developing into a similarly parallel structure, which is often called a Network-on-Chip (NoC). Shared, serial buses are replaced by pipelined communication networks that allow hundreds or thousands of communications going on concurrently at any time.

Only the third member of the trio computation, communication, storage, is lacking behind in this rapid transformation of sequential into massively parallel activity. Among the profound obstacles against making memory access as parallel as computation and communication, are the usage of shared memory for communication and temporary storage, and the large divide between on-chip and off-chip memory technology.

In 1995 Wulf and McKee described in a remarkable, short article a trend that is since then known as the approach to the memory wall [32]. Based on the observation that processor speed grows by 80% every year but memory access time decreases only by 7% per year, they predict that, if no invention breaks this

---

Royal Institute of Technology, Stockholm, Sweden

trend, program performance is solely limited by memory access within 5 to 12 years (in year 2000 and 2007, respectively). Before we discuss the situation today, it is worthwhile to recall Wulf and McKee's original argument. Consider the equation

$$t_{\mathrm{avg}} = p \times t_c + (1-p) \times t_m$$

where t_{avg} is the average access latency to access a data word, p is the probability of a cache hit, t_c is the access time to the cache, and t_m is the access time to main memory. If a program has 1 memory instruction per 4 other instructions, the memory wall is hit when $t_{\mathrm{avg}} \geq 5$ cycles (assuming one instruction takes 1 cycle to execute). When this condition is met, the program execution time is determined by the access time to main memory and there is no benefit to improve the processor performance.

Since 1995 many architectural innovations have avoided a hard impact into the memory wall. Improved interfaces such as DDR, DDR2, and DDR3 have increased the bandwidth to DRAM memory; packaging technology has increased the pin count such that two or four DDR3 interfaces are feasible today; additional levels of caches have helped to avoid the penalty of going off-chip. The former two effectively keep t_m low while the latter increases p.

To illustrate current state of the art, consider a fictive example with a 10×10 core chip, each core executes three instructions per cycle (either due to pipelining or with multiple functional units). There is an L1 cache for each core and one L2 cache on chip. The chip has four 64bit DDR3 interfaces. For the sake of simplicity we assume the access time to L1 is 1 cycle, to L2 is 10 cycles and to external DRAM it is 20 cycles. Let the hit ratio p be 90% for both caches. With a slightly generalized version of above equation, taking into account a second level of caching, we obtain $t_{\mathrm{avg}} = 2.0$ cycles. If each core would execute 1 instruction per cycle, we would be fine since we are below the limit $t_{\mathrm{avg}} = 5$. But because each core can execute 3 instructions per cycle and every 5th instruction on average is a memory access, our limit is in fact $t_{\mathrm{avg}} = 5/3 = 1.67$. Thus, our chip operates close to the memory wall and its performance is limited by the access latency to external memory. In fact it seems to be a fairly balanced system where neither the processors nor the memory access is over- or under-dimensioned.

However, this example assumes there is no contention in the network, at the DRAM ports, at the L2 cache, we have no delays due to cache coherence, memory consistency issues, or shared variable protection. If any of these issues appear, it will increase the t_{avg}, effectively making the memory access the bottleneck. Increasing the number of cores will significantly worsen the situation because it will increase the demand for on-chip data more than the off-chip interface will allow to pass through.

Wulf and McKee speculated that the most convenient resolution to the problem would be "the discovery of a cool, dense memory technology whose speed scales with that of processors". Such a technology may in fact come to our rescue. ZRAM [2] and the memristor [31] are contenders for very dense, very fast memory technology with nicely scalable access times. Integration of logic dies with memory dies in a 3D stack is already viable today and has very strong potential for the high performance, high volume consumer markets in the next few years. Placing DRAM dies on top of the logic dies puts the memory within a few tens of μm distance from the core that needs the data. Through Silicon Vias (TSVs), the interconnect between two vertically stacked dies, occupy an area of less than $100\mu m^2$, consume less than $25\mu W$, and incur a latency of less than a few hundred picoseconds. All this together gives us performance per power and cost improvements of 1000X to 10000X over off-chip connections [19, 30]. Hence, high density, low cost memory can be placed very close to the cores of the many-core chip, effectively bringing t_{avg} to 1 and eliminating one or two cache levels.

However, there is a catch, requiring architectural innovation in addition to manufacturing technology. The geometric distance and thus the access latency varies a lot depending on where in the system the core and the accessed memory are. A core can fetch a variable within one cycle if it is stored in the memory

bank just above, but it may take 10-100 cycles or more to fetch it from across the chip. Already Wulf and McKee have suggested that it may be wise to abandon the assumption that access time is uniform to all parts of the address space, and in [19] it is shown, that the principle performance limit in a 3D stacked system and a projected 17nm technology is a factor 34 higher for a distributed memory system compared to a centralized one. Following the same logic, Kim et al. [20] have proposed a *Non-uniform Cache Architecture (NUCA)* in 2002 and many others have elaborated this idea since then (e.g. [4, 10, 11, 18, 33]). When 3D integration became a realistic option, a number of groups used it to integrate logic and memory, first by reusing traditional memory components [23, 25]. In a next step, customized memory architecture were proposed to fully exploit the potential [22, 24].

Reviewing these arguments and the technology trends, we conclude that a distributed memory organization with massively parallel access to different parts of the memory will allow for scalability en par with many core architectures and general on-chip communication networks. It will avoid memory access to become a bottleneck even for systems with thousands of cores and NoCs with raw bandwidth in the range of petabit per seconds.

In order to make distributed memory an attractive option to system designers, we need to address all the caveats and problems that typically come with distributed memory schemes. One first, critical question is, if the memory space is shared or not. Keeping memory only local and private is an elegant architecture solution but ignores rather than solves the issues of legacy code and programing convenience. A lot of legacy software assumes a shared memory model and most programmers find it easier to express themselves within a shared memory programming paradigm rather than a model that is strictly based on message passing. Consequently, we believe that the architecture should support both a private and a shared memory organization.

Distributed, shared memory (DSM) schemes need to address cache coherency, memory consistency, and synchronized and protected access to shared variables. In order to offer solutions for all these and potentially many other memory and data management related tasks, we propose a programmable controller, that is optimized for managing the memory and data. It is called *Data Management Engine (DME)* and it is described in the following sections.

The DME provides efficiency through specialization. It can be viewed as an attempt to increase the parallelism on chip and to increase efficiency by means of specialization. By increasing parallelism higher performance can be gained at the same frequency; by specialization, less energy is expended for the same task. Since memory access is common in all applications and determines to a large extent performance, cost, and power consumption, it is reasonable to expect that a specialized memory transaction handler can improve performance and efficiency at modest cost.

1.2 DME-enhanced Multi-core NoC Platform

The basic concepts of the DME have been first introduced at DATE 2010 [8]. It has since then been used for a range of different applications and in different ways, for instance for different synchronization techniques [5,7,9] and for run-time partitioning of private/shared memory [6]. In this section we review the basic architecture of the DME, its main functions and features. In the following sections we then touch upon some of the applications and usages of the DME.

Fig. 1.1 a) shows an example of our DME-enhanced multi-core NoC platform. The system is composed of 16 Processor-Memory (PM) nodes interconnected via a packet-switched network. The network topology is a mesh. As shown in Fig. 1.1 b), each PM node contains a processor core, a DME, a local memory, and other hardware modules connected to the local bus. In our experiments, we use Leon3 [1] as the processor

core, but any other core, or IP, or local computing cluster could be used as well. The DME connects the local processor core, the local memory, and the network. It not only handles all memory transactions from a local processor to both local memory and remote memory on- or off-chip, but also serves remote memory requests from remote processors via the network.
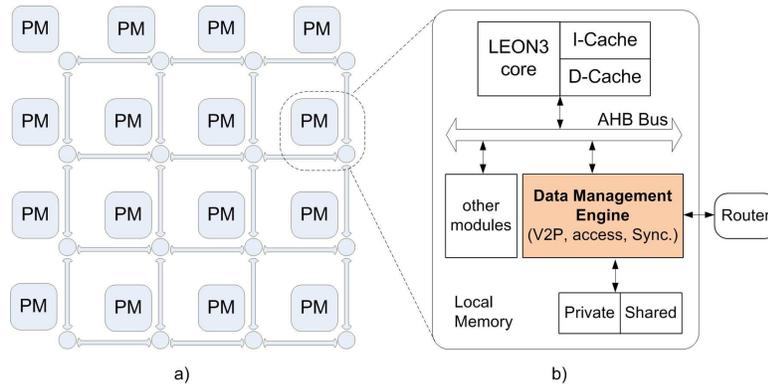
Fig. 1.1 In a multi-core network (a) each node contains its own DME (b) for handling memory transactions. The figure illustartes the experimental platform used.

The DME contains two programmable micro-controllers called *mini-processor A* and *mini-processor B* (figure 1.2). Their instruction sets are identical and include general purpose instructions such as load, store, addition, subtraction, condition testing, and branching, as well as a number of specialized instructions for support of synchronization, message generation, and address manipulations. In principle, all kinds of functions can be realized by the DME, since it is programmable, and the main local processor can use it as a local, specialized co-processor to which it can off-load memory management functions.

The two mini-processors fetch their instructions from a local control store, which they share, but operate on their own private set of registers. Mini-processor A is triggered by commands from the local processor that are received through the *Core Interface Control Unit (CICU)*. Mini-processor B is either invoked by a command from the local mini-processor A or by a request received from a remote node through the *Network Interface Control Unit (NICU)*. The *Sync Supporter* manages the access to locks by means of two specialized instructions, *load linked (ll)* and *store linked (sl)*. It allows for implementation of locks and barriers.

1.3 Data Managment Engine (DME)

Although it is optimized for handling memory transactions and for managing the memory and address space, the DME is a programmable dual-core controller with a fairly general instruction set. First we describe the overall architecture of the DME and its components (section 1.3.1), then we elaborate on the execution flow of the DME (section 1.3.3) and the methodology for developing new micro-programs (section 1.3.4).
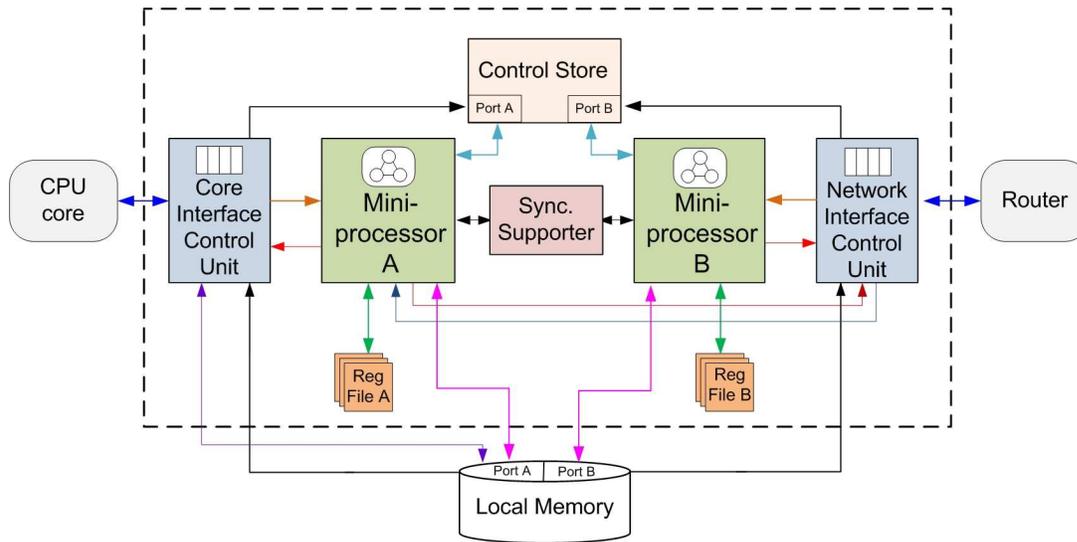
Fig. 1.2 Architecture of the Data Management Engine.

1.3.1 Architectural Design

As shown in Fig. 1.2, the DME, 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. A microprogram is initially stored in the local memory, and is 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 it guarantees atomic read-and-modify operations (see section 1.4.3). Both the local memory and the control store are dual ported: port A and B, which connect to the mini-processors 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 commands from the local core and triggers the operation of the mini-processor A accordingly; (II) if the microcode is not already in the control store, it uploads it 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 commands from the network and triggers the operation of the mini-processor B accordingly; (II) if the microcode is not already in the control store, it uploads it 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 by packaging the request in a message via the network; (IV) it receives the remote results as messages from remote destination nodes via the network, unpacks them, 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 mini-processors feature 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 branching, and message-passing. The microinstructions are designed to exploit the hardware architecture of the mini-processors. the microinstructions are organized *horizontally* [29].

Synchronization Supporter

The synchronization supporter, which connects with the mini-processor A and B, is a hardware module to support atomic read-and-modify operations. This is necessary to support locks 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. It acts as an instruction cache and 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 DME 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 of microcode into the control store.

1.3.2 DME Implementation

The DME design has been synthesized by Synopsys Design Compiler in SMIC 90nm technology and the control store is generated by Artisan Memory Compiler. The control store is $2048 \times 128b$ dual port SRAM. Table 1.1 shows the results of the synthesis in terms of maximum frequency and gate count. For power analysis the DME has been synthesizes with the Cadence RTL compiler with TSMC 90nm technology, and the gate netlist has been simulated for all possible transactions and instructions. Table 1.2 gives the power consumption of the different DME components. The power consumption is given for an idle DME at 400MHz clock frequency. Depending on the operation of the DME, the power consumption has been observed to be up to 15-20% higher compared to the idle state.

Table 1.1 Synthesis results with 90nm SMIC technology.

	Optimized for area	Optimized for speed
Frequency	448 MHz (2.23 ns)	500 MHz (2 ns)
Area (Logic)	46k NAND gates	57k NAND gates
Area (Control Store)	237k NAND gates	

Table 1.2 DME power consumption at 400MHz and implemented with 90nm TSMC.

	power consumption [mW]
Mini A	6.9
Mini B	7.0
NICU	2.3
CICU	5.2
Synchronizer	0.2
DME total	21.6

1.3.3 Command-triggered Microcode Execution

For the DME, the execution of the mini-processors is triggered by requests (in form of commands) from the local and remote cores. This is called *command-triggered microcode execution*.

As shown in Fig. 1.2, the two interface units are coupled with their corresponding mini-processors 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-processors are
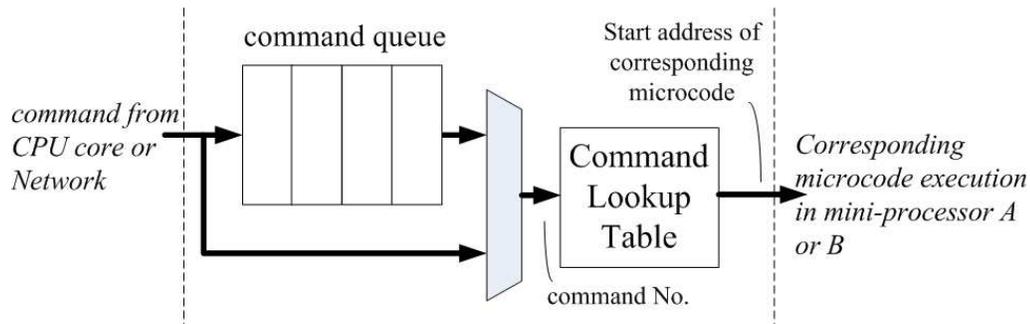
Fig. 1.3 Command-triggered Microcode Execution

microprogrammable. Fig. 1.3 illustrates command-triggered microcode execution. As shown in the figure, there is a *command queue* as well as a *Command Lookup Table (CLT)* in each interface unit. The queues buffer commands from the CPU core or remote cores via the on-chip network. If both the command queue is empty and the mini-processor is idle, the command bypasses the command queue to reach the CLT directly.
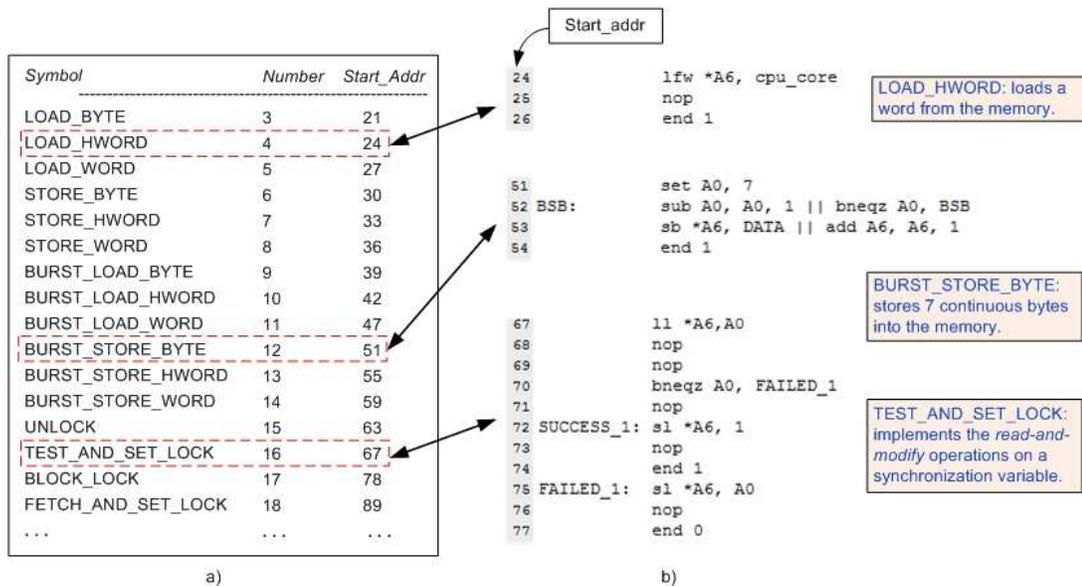

Fig. 1.4 a) Command Lookup Table (CLT) and b) microcode segments

   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 mini-processor is able to know where the current microcode execution starts. Fig. 1.4 a) shows an example CLT. The "*Symbol*" is mnemonic. The command "*Number*" has a one-to-one correspondence with the "*Start_Addr*" of the related microcode. Fig. 1.4 a) lists several commands we have implemented, and Fig. 1.4 b) shows snapshots of three pieces of microcode. As

we can see, "LOAD_HWORD" command with its command No. 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.

As illustrated in Fig. 1.5, the DME works as follows (the microprogram is initially stored in the local memory):
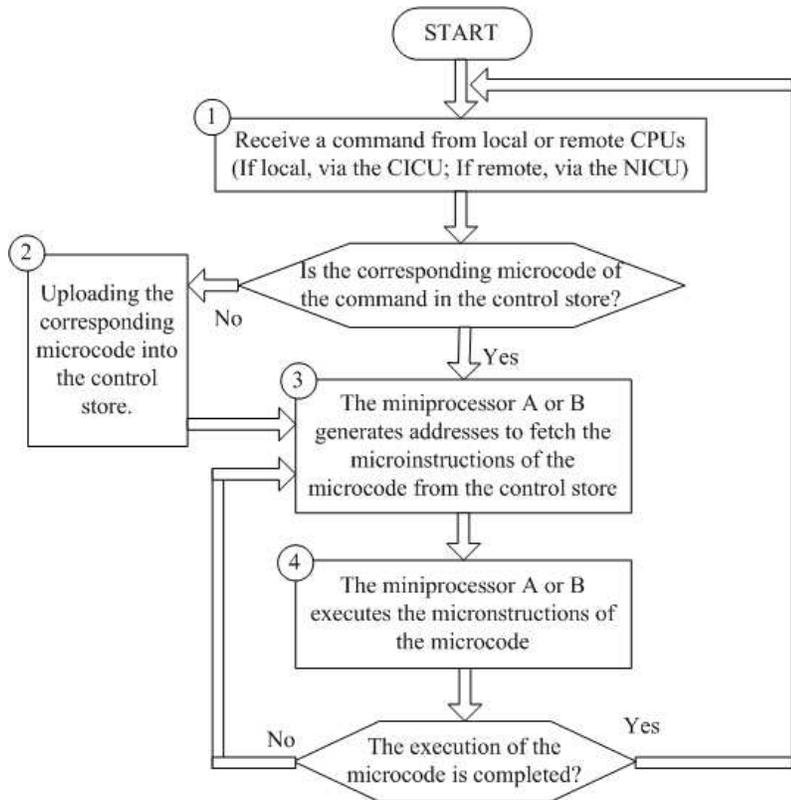
Fig. 1.5 The execution flow of the DME.

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 or B 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 during the entire execution period of the system.
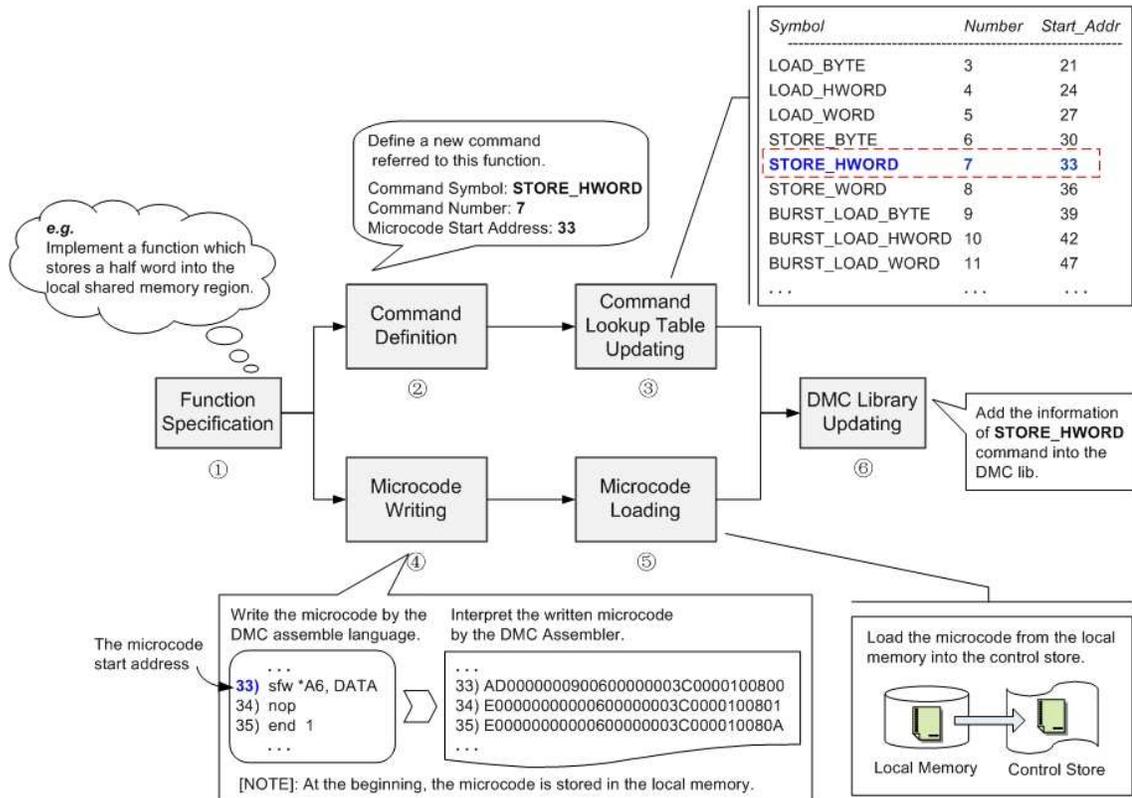
1.3.4 Microprogramming Development Flow

Fig. 1.6 Microprogramming development flow

Developers can develop new micro-programs for the DME in order to customize existing functions for a specific application and/or architecture, or to develop whole new functions. The current, rather rudimentary, tool support consists of an assmbler and a configuration tool. For future development a C compiler and debugging support is planned. Fig. 1.6 depicts the entire flow of microprogramming the DME. The flow consists of 6 steps.

① The user specifies the function to be implemented in the DME.

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*".

④ The user writes microcode in the DME assemble language for the specified function, then translates it into executable binary code by the DME assembler.

⑤ The binary microcode is uploaded from the local memory into the control store. If the programmer omits this step, the code is uploaded automatically at run time when needed.

⑥   The DME 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. 1.6, 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 **STORE_HWORD**, Command No. **7** and the start address **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*". In step ⑤, 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 DME Library. Afterwards, the command **STORE_HWORD** can be used successfully.

1.4 DME Applications

Due to its general purpose nature, a wide range of memory and data management functions can be realized on the DME in many different ways. So far we have experimented with memory partitioning, virtual address space, synchronization, cache coherency, local synchronization, memory consistency, and dynamic memory allocation. We describe some of these functions in the following sections to illustrate the usage and efficiency of the DME. The design and implementation of a complete dynamic memory allocator on the DME is described in chapter **??**.

1.4.1 Private and Shared Memory Partitions

A main source of overhead is the size of the shared memory space. For virtual-to-physical address translation, the size of translation tables are a linear function of the number of pages in the shared address space. Likewise, in a directory based cache coherence protocol, the size of the directory is in some schemes directly proportional to the size of the total shared memory. Moreover, access to shared memory carries a significant latency penalty, compared to private memory access, due to address translations, table look-ups, coherence, and consistency protocols. Consequently, it is desirable to keep the shared memory space as small as possible.

The DME allows the application programmer to specify the size of private and shared memory partitions in a very flexible way. These partitions can be different in different nodes and can be dynamically modified to track the varying needs of an application at run-time.

The DME maintains a register called BADDR (Base Address), that is a pointer dividing the private from the shared address space. If a memory transaction uses an address less or equal to this number, it addresses the private space; otherwise it is a shared address. This pointer can be set differently in different nodes, as illustrated in figure 1.7, and it can be modified dynamically. Local, private memory accesses are handled by mini-processor A only and are very fast. Essentially, they are simply forwarded by the DME to the local memory port within one cycle. Access to shared and remote memory requires more bookkeeping and incur a few cycles delay inside the DME. Requests to a remote memory location are handed over to mini-processor B, which packs the memory transaction into a packet and sends it over to the remote DME through the network. At the far side mini-processor B receives the request, accesses its own local memory, and sends
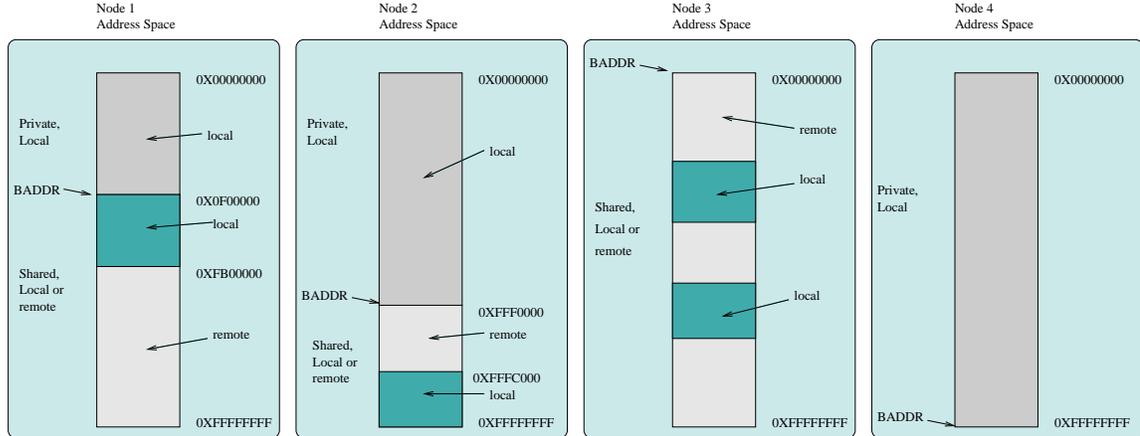
Fig. 1.7 Different nodes may partition their address space differently.

the response back over the network. Locally, mini-processor B receives the response and hands it over to the main processor.

If we have a physical address space only, a memory translation process based on equally sized pages yields the details of the remote memory address. A table in local memory is used to translate the remote address into a node id, a page address in the remote memory, and a page offset.

The dynamic change of the BADDR pointer gives the platform user very high flexibility. Moreover, it allows for the usage of tricks to maximize performance. For instance, consider a producer node that generates an array of data, which another consumer node uses for further processing. The array can be allocated in the shared memory part of the consumer's local memory. The producer writes to this array, allowing the write access latency over the network to be hidden since the producer does not have to wait until the writes complete. Once the full array is generated, the consumer node can change BADDR such that the array now falls into local, private memory. Access to this part would now be very fast because the bookkeeping overhead of shared memory access can be avoided.

Tricks like these can maximize performance, but they can potentially lead to inconsistent system states and hard-to-find errors. Therefore, we consider to implement an automatic table update mechanism triggered by a change to BADDR.

1.4.2 Virtual Address Space

The current version of the DME also supports a virtual address space giving flexibility to application programmers and compilers because the addresses used in a program can be kept separate of other programs and independent of where in the system the program is executed. On the other hand, the management of a virtual address space and the translation from virtual to physical addresses incur quite some overhead in terms of latency, translation tables, and power consumption.

Therefore we allow currently only a few of the possible combinations, listed in table 1.3. Private memory can only be local and with a physical address. In contrast, shared memory requires a virtual address. This policy limits the number of possible cases and allows for a high performance, low overhead implementa-

Table 1.3 Supported combinations of memory access features.

local / private / physical	Supported
local / private / virtual	-
local / shared / physical	-
local / shared / virtual	Supported
remote / private / physical	-
remote / private / virtual	-
remote / shared / physical	-
remote / shared / virtual	Supported

tion. Access to local private memory requires no virtual address translation and is therefore processed in one cycle, essentially passing through the DME from the CPU to local memory. Shared memory access requires a virtual-to-physical address translation and can be local or remote. The current DME implementation performs a virtual-to-physical address translation in 11 cycles.

1.4.3 Synchronization

The Synchronization Supporter in the DME provides the hardware support for memory synchronization. It works with microoperations **ll** and **sc** to guarantee atomic read-and-modify operations on mutex locks. Based on them, higher level synchronization mechanisms can be built.

We have implemented two synchronization primitives: *test-and-set()*, *test-and-set-blocking()*. *test-and-set()* is non-blocking and the programmer may use it to implement a spin lock. However it incurs additional network traffic as the program re-spins the lock, as illustrated in Fig. 1.8 a). *test-and-set-blocking()* 1.8(c) is an improvement over the conventional spin-lock based blocking *test-and-set()* [17] (Fig. 1.8 b)). With the conventional blocking *test-and-set()*, the mini-processor B in the remote node would continue to spin the lock until success while executing the microcode. This does not incur additional network traffic, but other requests from other nodes will be blocked as the mini-processor B continuously polls the lock. Our proposed *test-and-set-blocking()* utilizes the cooperation of the mini-processor B and the NICU. If an acquire of the lock fails, the related command will be placed to the tail of the command queue to wait for the next execution. This avoids incurring additional network traffic and will not block other commands.

1.4.4 Cache Coherency

One of the more challenging problems in a distributed shared memory system is to realize a scalable, high performance, low overhead cache coherence scheme. Snooping based protocols do not scale well with interconnects where broadcasting transactions to all members in the system is prohibitively expensive. Directory based protocols scale relatively well in terms of performance but care has to be taken to avoid large memory overhead due to the directory (see [12, 16] for general discussions of the topic).

As motivated above, we expect the application programmer to keep the shared memory space small. Therefore, we have, in a first stage, implemented a classic and simple directory based cache coherence protocol. The memory overhead due to the directory is modest if the size of the shared memory space is kept within reasonable bounds.
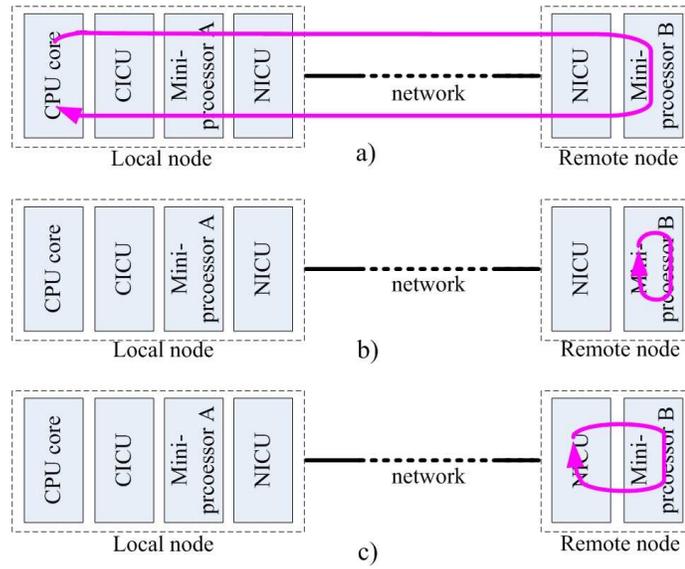
Fig. 1.8 a) *test-and-set()*; b) the conventional blocking *test-and-set()*; c) our proposed *test-and-set-blocking()*
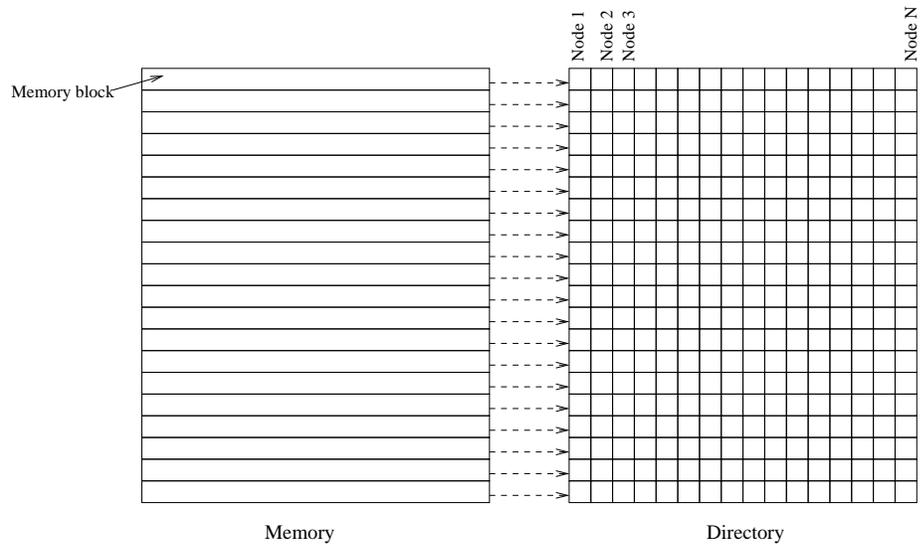
Fig. 1.9 The directory contains an entry for each block of shared memory.

The directory maintains status information about each block of shared memory (figure 1.9). A block in main memory corresponds to a line in the cache. When a node has a shared memory partition, a fraction of the memory has to be sacrificed for the directory. For each block of memory there is one entry in the directory, which contains one bit for each node in the network. This bit is 1, when the corresponding node holds a copy of the block in its local cache (the node is a sharer); 0 otherwise. Thus the size of the directory is $N \times M$ bit, where N is the number of nodes in the network, and M is the number of blocks in the shared memory space.
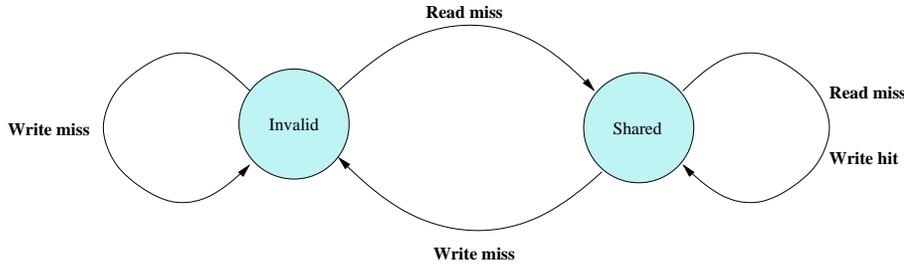

Fig. 1.10 State transition diagram of the SI cache coherence protocol.

We use a simple, write-through, no-allocate policy with a Shared-Invalidate (SI) protocol [13]. On a write, data is always written to main memory which is thus kept up to date. On a write-miss, the date is not written to the local cache (no-allocate). With this policy no node is ever the exclusive owner of a memory block and the corresponding cache coherence protocol can be kept simple (figure 1.10).

This scheme has several advantages: (1) it is easy to implement; (2) the main memory has always the most recent value; (3) a read miss results never in a write to memory. On the downside we have: (1) write is slow, because it always goes to main memory; (2) every write results in a memory transaction even if the same node writes many times to the same location in sequence; (3) more memory transactions appear on the network which has to be able to cope with this load.

We implement several routines on the DME to operate the protocol. These routines realize the directory control and the memory access. The routines generate messages according to the protocol and change the directory states.

Here, three kinds of nodes are involved. The *Home Node* is the node which hosts the target memory and the related directory. The *Local Node* is the node which issues the read/write request. Finally, the *Remote Node* is the node which also maintains the data copy in its cache. When there is a read/write miss in the processor's cache, a read/write request is sent to its DME. If the address is in local, shared space, the request is processed by its own DME. If it is a remote, shared address, the Local Node sends the request to the Home Node. Once the request arrives at the Home Node, the read/write routine for cache coherence is triggered.

For read request, the Home Node always keeps the up-to-date data. The data is returned to the Local Node directly. Figure 1.11(a) shows the read procedure. In a write request (figure 1.11(b)), there may be one or more Remote Nodes that also have the data in their local caches. So the Home Node needs to send invalidation requests to all these Remote Nodes. And once all the invalidation acknowledgements have returned, the Home Node grants the Local Node the right to update the data.

Although this scheme is relatively simple and is not effective in all cases, it can offer significant performance improvements and good scalability properties for applications with a modest amount of shared
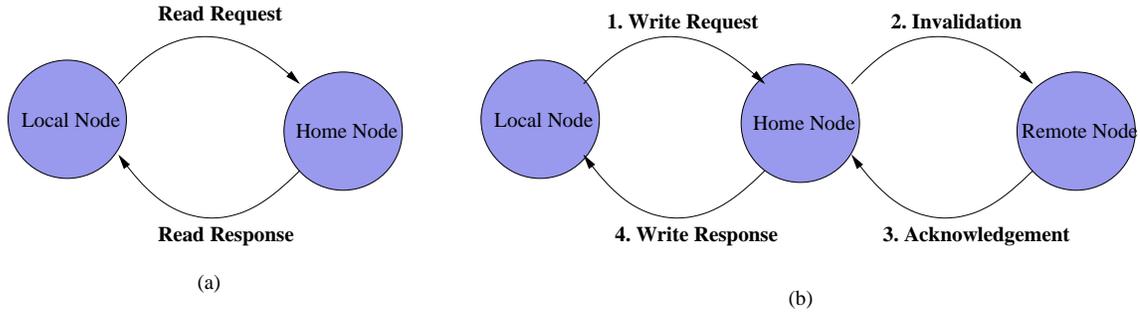
Fig. 1.11 The read procedure (a) and write procedure (b) in the cache coherence protocol.

memory and fairly localized communication and memory access patterns. As future work we plan to realize a hierarchical cache coherence scheme along the lines described in [34].
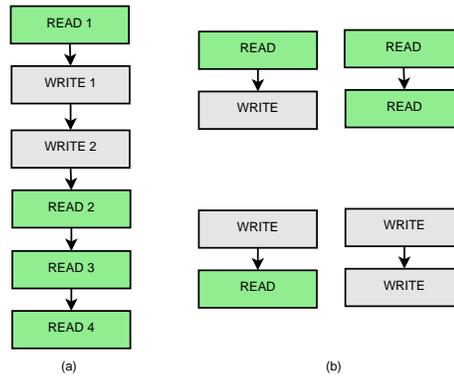
1.4.5 Memory Consistency

The memory consistency model is a contract between the programmer and shared memory parallel system. The programmer follows the rules that are guaranteed by the parallel system to get the predictable results of the shared memory operations. The shared memory access latency can be reduced by the hardware and software optimizations in the system architecture. These performance optimizations can reorder the shared memory operations and the system may give unexpected results. For the expected results, these reordering should be controlled carefully. Different memory consistency models enforce different ordering constraints on the shared memory operations [3, 12]. Here we focus on some of the memory consistency models that are realized in the DME based multi-core systems.

1.4.5.1 Sequential Consistency

The sequential consistency (often called Strong Ordering) defined by Lamport [21] has to maintain the program order among operations of each individual processor and sequential order among multiple processors in the system. It is a strict model and does not allow reordering between shared memory operations in the multi-core systems. A memory operation (read or write) cannot be reordered or overlapped with the following memory operation to the different locations in the shared memory. The shared memory operations are completed according to the program order as shown in figure 1.12(a). The sequential consistency enforces the global orders on shared memory operations as given in figure 1.12(b).

The sequential memory consistency model in the DME based multi-core NoC platform is realized by stalling the processor on the issuance of a shared memory operation till the completion of the preceding operations [27]. Then, the processor issues the next operation. The completion of preceding operation is indicated by the return data or acknowledgment. The program order is maintained due to the strict order between the shared memory operations and sequential order is maintained by read-modify-write memory operation in the system.

Fig. 1.12 a) Strong Ordering b) Global orders to enforce.

The sequential memory consistency model does not allow the performance optimizations [3] in the hardware (write buffer, cache, interconnection network) and in the software (compiler reordering, register allocation) due to the strict order enforcement on the shared memory operations. As a result relaxed memory consistency models emerged that permit such optimizations. The relaxed consistency models relax the program order requirement to allow the possible reordering in the shared memory operations by the system optimizations. The shared memory operations may not complete according to the program order. The overall program correctness is ensured by enforcing ordering constraints on a subset of memory operations. We consider the two relaxed consistency models (weak and release consistency) which relax the strict program order requirement and allow reordering in the shared memory operations.

1.4.5.2 Weak Consistency

The weak consistency model (also called weak ordering) was proposed by Dubois et al. [14] which classifies shared memory operations as *synchronization* and *data* operations. Synchronization operations are related to the special synchronization variables (locks, semaphores) in the shared address space. The lock must be gained exclusively in the multi-processor shared memory systems. Data operations are the load-store operations related to the ordinary shared variables. According to the weak consistency model all previously issued outstanding data operations must be completed before the issuance of synchronization operation. Similarly, previously issued outstanding synchronization operations must also be completed before the issuance of any data operation as depicted in figure 1.13(a). The weak consistency model ensures the final consistent result of the program execution in the multi-processor systems. The weak consistency model enforces some global orders on the shared memory operations as shown in figure 1.13(b). The enforcement of these global orders on the shared memory operations ensures the program correctness in the weak consistency model with the permitted relaxation in data operations.

The transaction counter (TC) based approach [3,27,28] is adapted for the realization of the weak memory consistency model in the DME based multi-core systems. The counter is implemented in the DME hardware in each node to keep track of the outstanding data operations issued between the two synchronization points. It is incremented and decremented by the issuance and completion of data operations correspondingly. It is not affected by the synchronization operations.
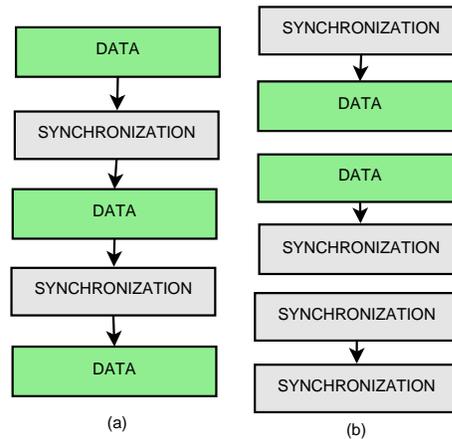
Fig. 1.13 a) Weak Ordering b) Global orders to enforce.

1.4.5.3 Release Consistency

The release consistency model proposed by Gharachorloo et al. [15] is a refinement of the weak consistency and distinguishes synchronization operations as *acquire* and *release* operations. It removes the following two unnecessary ordering constraints and allows further relaxation in the program order as compared to the weak consistency model.

- Non-critical section data to synchronization
- Synchronization to non-critical section data

According to the release consistency model, an acquire operation must be performed before the issuance of any data operation in the critical section (CS) and in the non-critical section (NCS) after it. All the data operations in the critical and non-critical sections prior to the release operation must be completed before the issuance of the release operation as shown in figure 1.14(a). The release consistency model enforces the global orders on the shared memory operations as given in figure 1.14(b).

The release memory consistency model can be realized by using two transaction counters in the hardware of DME in each node of the platform [26]. Two counters in each node correspond to two types of data operations. The transaction counter 1 (TC1) keeps track of outstanding data operations issued in the non-critical section of code by the processor. The transaction counter 2 (TC2) keeps track of outstanding data operations issued within the critical section. Each counter is incremented and decremented by the issuance and completion of the relevant data operations correspondingly. Both these counters are not affected by the acquire and release synchronization operations. "TC1 = 0" indicates the completion of all previously issued outstanding data operations in the non-critical section of code. "TC2 = 0" indicates the completion of all the previously issued outstanding data operations in the critical section of code. The lock acquire operations do not check the counters at the issuance time, while the release operations are not issued by the processor until both these counters become zero.
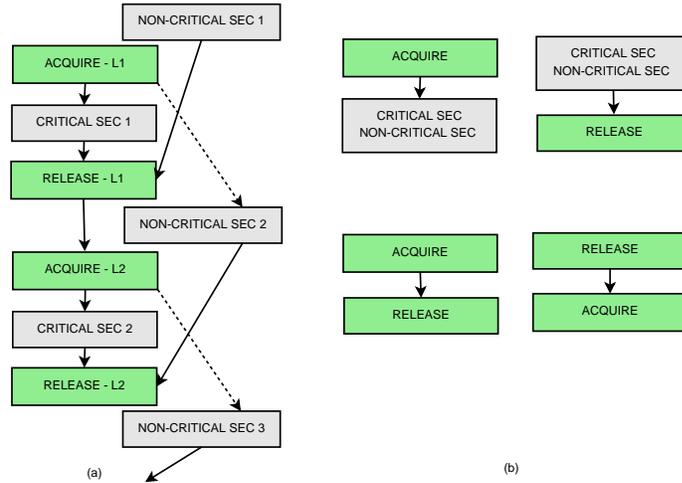
Fig. 1.14 a) Release consistency model b) Global orders to enforce.

1.5 Experiments

We have studied the DME behavior in a number of different experiments. In the following we describe some of these experiments to show, that the usage of the DME is feasible (section 1.5.1), how the DME can cleverly be used to maximize performance (section 1.5.2), and the performance under synchronization constraints (section 1.5.3).

1.5.1 Viability Analysis

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 $C[64,64]$ matrix 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 stored 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.

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 forms a wavefront propagating through the processor array.

Fig. 1.15 shows the performance speedup of the three applications. We observe, that the multi-core NoC achieves fairly good speedup. When the system size increases, the speedup increases almost linearly. The speedup Ω_m is defined as $\Omega_m = T_{1core}/T_{mcore}$, where T_{1core} is the single core execution time as the baseline, T_{mcore} the execution time of m cores.
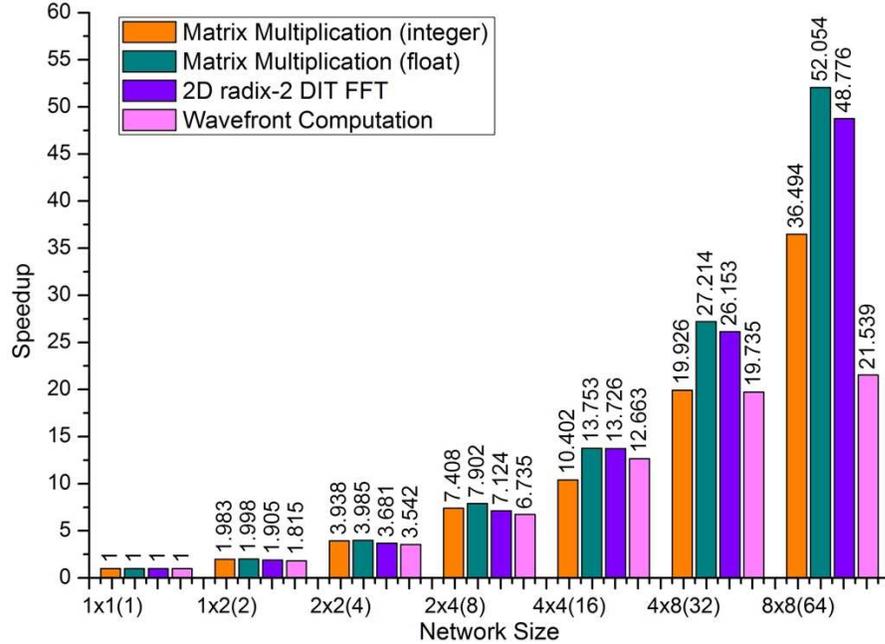
Fig. 1.15 Speedup of matrix multiplication, 2D radix-2 DIT FFT and Wavefront Computation

For instance, the speedup for the matrix multiplication on the 8x8 system is 52.054, which is very close to the ideal speedup of 64. However, as the system size increases, the speedup acceleration slows down. This is due to the growing communication latency which increases linearly with the system size, limiting the performance.

Note, that the speedup for the floating point matrix multiplication is higher than that for the integer matrix multiplication, 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.

Obviously, the speedup figures strongly depend on the application characteristics and on the task partitioning and mapping. We could achieve very good speedup results, because the four applications are inherently easy to parallelize and have been mapped well. This is not true for many other applications. But the experiments demonstrate, that, given applications that can be parallelized and mapped well onto a multi-core platform, the DME, handling the memory access and communication, is not limiting the speedup and does not constitute a bottleneck for system performance.

1.5.2 Performance Optimization

In the following experiment we show, how system performance can be improved by exploiting specific features and the flexibility of the DME.
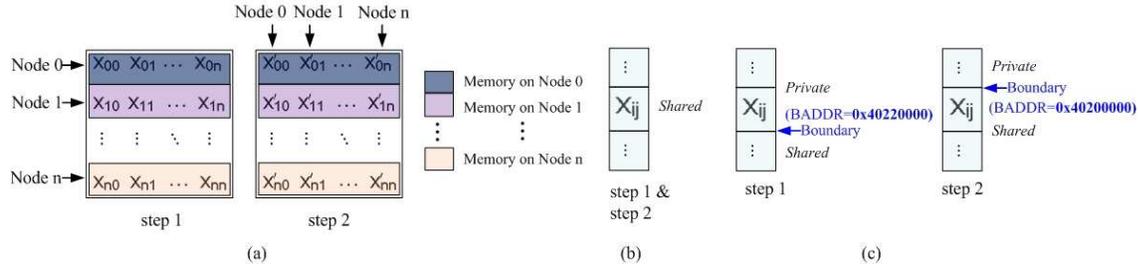
Fig. 1.16 (a) Memory allocation for 2D DIT FFT, (b) conventional DSM organization, and (c) Hybrid DSM organization with run-time partitioning

We implement a 2D radix-2 DIT FFT. As shown in Fig. 1.16 (a), the FFT data are equally partitioned into n rows, which are stored on the n PM nodes, respectively. According to the 2D FFT algorithm, the application first performs FFT on rows (Step 1). After all nodes finish the row FFT (synchronization point), the FFT on columns are started (Step 2).

We experiment with two DSM (Distributed Shared Memory) organizations. One is the conventional DSM organization, as shown in Fig. 1.16 (b), for which all FFT data are shared. The other is the hybrid DSM organization, as illustrated in Fig. 1.16 (c). The data used for row FFT calculations at step 1 are located locally in each PM node. After step 1, they are updated and their new values are to be used for column FFT calculations at step 2. We can dynamically re-configure the boundary address (BADDR in Fig. 1.16) at run time, such that, the data are *private* at step 1 but become *shared* at step 2.

Fig. 1.17 shows the speedup of the FFT application with the conventional DSM organization, and performance enhancement of the hybrid DSM organization with run-time partitioning. As we can see, when the system size increases from 1 to 64, the speedup with conventional DSM organization goes up from 1 to 48.776, and the speedup with hybrid DSM organization improves from 1.525 to 55.070.

For the different system sizes the improvement of the hybrid DSM organization is between 11.44% and 34.42%.

We can summarize the experiment as follows.

- Using run-time partitioning in hybrid DSM organization, a fast *physical addressing scheme* is performed in step 1 of the 2D DIT FFT and the entire virtual address translation overhead is avoided.
- As the system size is scaled up, larger communication delay leads to the decrease of performance improvement.
- The single PM node case has a higher improvement because all data accesses are local and shared for the conventional DSM organization and private for the hybrid DSM organization, and there is no synchronization overhead. Thus, the single PM case also shows the bookkeeping and translation overhead of managing a shared memory space.
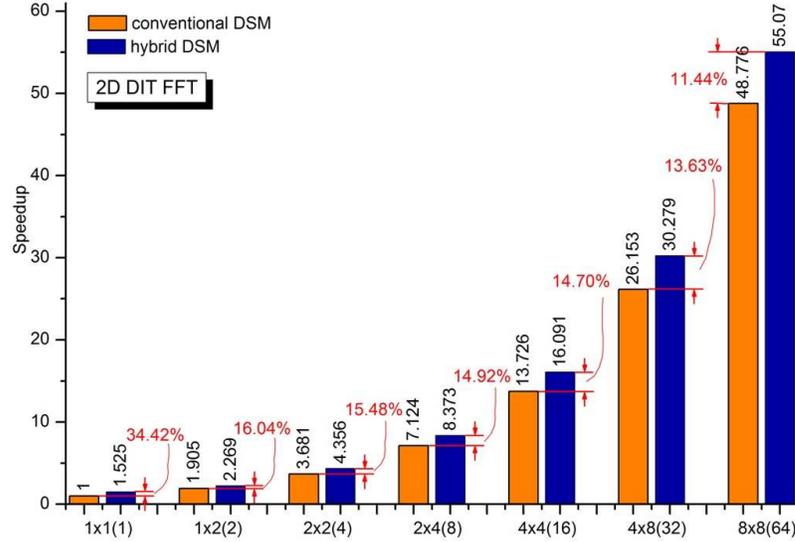
Fig. 1.17 Speedup and performance improvement of 2D DIT FFT

1.5.3 Experiments for Memory Consistency

We analyze the performance of the three consistency models that are realized in the multi-core system. The effects of network size on average and maximum code latencies are investigated. As traffic pattern we use a set of synthetic workloads running on each node in the platform. The sequence of transactions is the same for each node and is shown in figure 1.18. Two nodes constitute hotspots: one for the critical section (CS-node) and the other for the lock (SYNC-node). Each node sends synchronization requests to the SYNC-node. The shared memory locations in the CS-node are protected by the acquire and release operations to the lock in the SYNC-node.

The average and maximum code execution time, which we call *code latency*, for different network sizes is shown in figure 1.19. The code latency increases for all the three consistency models as the network grows from a single core to 64 cores. The average code latency for the release consistency model in the 8×8 network is about 236.59 times of the single core, whereas for the weak and sequential consistency models it is 241.96 and 353.67 times, respectively. The difference between the observed code latencies become obvious as the network size grows. It is due to the further overlapping and program order relaxation in the release consistency as compared to the weak consistency model. The sequential consistency does not allow any kind of reordering in the shared memory operations. The weak consistency model allows the reordering among the shared memory operations in the critical section or non-critical section. The release consistency model permits overlapping among the data operations in the critical and non-critical sections. The release consistency improves the performance by 2.3% and 49.5% on average in the code latencies over the weak and sequential consistency models, respectively, as the system grows from a single core to 64 cores.
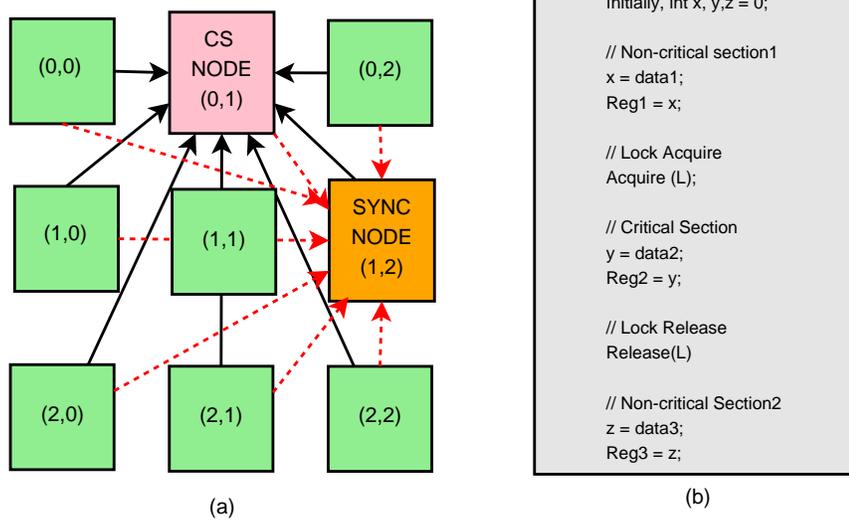
```
Initially, int x, y,z = 0;

// Non-critical section1
x = data1;
Reg1 = x;

// Lock Acquire
Acquire (L);

// Critical Section
y = data2;
Reg2 = y;

// Lock Release
Release(L)

// Non-critical Section2
z = data3;
Reg3 = z;
```

(a)   (b)

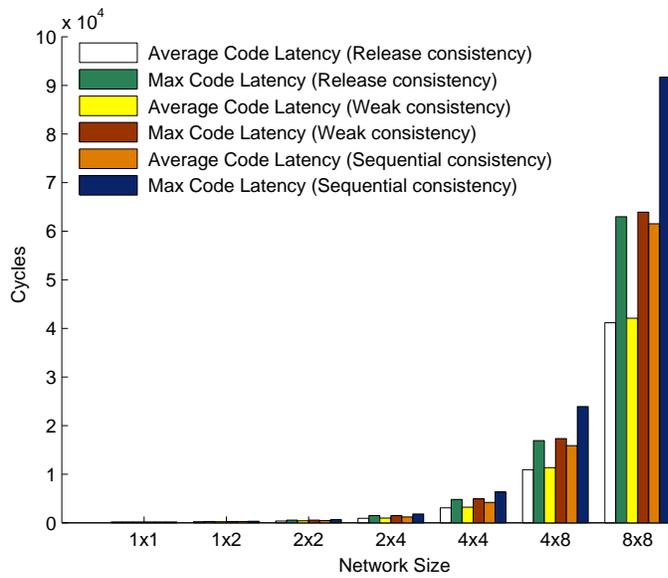**Fig. 1.18** Each node generates the same sequence of transactions.

Fig. 1.19 Execution times of a test program on different architectures.

1.6 Conclusion

Due to the pace of technology development, computation and communication is operated more and more in parallel. Indeed, the degree of parallelism grows at the rate of Moore's law. As a consequence, memory access must follow suit and become more parallel. 3D stacking and other emerging technology facilitates this trend, but has to be matched by a corresponding adaptation in the memory architecture.

We propose a programmable hardware block, called a Data Management Engine, that supports this architectural adaptation in multiple ways. The DME can realize an address space partitioning into private-shared and into local-remote sections. While the local and remote memory is determined by the platform at design time, the partitioning into private and shared sections can be flexibly modified at run-time, and can be used for performance optimizations. Among the many other potential DME applications we present virtual address space management, synchronization, cache coherency, and memory consistency. We also demonstrate the feasibility of the DME in several experiments.

References

1. The Aeroflex Gaisler webpage. http://www.gaisler.com/.
2. Z-RAM technology backgrounder. http://www.innovativesilicon.com/en/pdf/z-ram.pdf.
3. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.
4. Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.
5. Xiaowen Chen, Shuming Chen, Zhonghai Lu, and Axel Jantsch. Area and performance optimization of barrier synchronization on multi-core network-on-chips. In *3rd IEEE International Conference on Computer and Electrical Engineering (ICCEE)*, Chengdu, China, November 2010.
6. Xiaowen Chen, Zhonghai Lu, Shuming Chen, and Axel Jantsch. Run-time partitioning of hybrid distributed shared memory on multi-core network-on-chips. In *The 3rd IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP 2010)*, Dalian, China, December 2010.
7. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Handling shared variable synchronization in multi-core network-on-chip with distributed memory. In *International SOC Conference*, Las Vegas, Nevada, September 2010.
8. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proceedings of the confernece for Design Automation and Test in Europe*, Dresden, Germany, March 2010.
9. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting efficient synchronization in multi-core NoCs using dynamic buffer allocation technique. In *Proceedings of the IEEE Annual Symposium on VLSI*, Kefalonia, Greece, July 2010.
10. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.
11. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *SIGARCH Comput. Archit. News*, 33(2):357–368, 2005.
12. David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufman Publishers, 1999.
13. Pierre Guironnet de Massas and Frédéric Pétrot. Comparison of memory write policies for NoC based multicore cache coherent systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 997–1002, New York, NY, USA, 2008. ACM.
14. Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

15. K. Gharachorloo, D. Lenoski, J. Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer Architecture News*, 18(2):15–26, June 1990.

16. J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3):418 –429, March 1999.

17. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

18. Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.

19. Axel Jantsch, Matthew Grange, and Dinesh Pamunuwa. The promises and limitations of 3-D integration. In Abbas Sheibanyrad, Frédéric Pétrot, and Axel Jantsch, editors, *3D Integartion for NoC-based SoC Architectures*, Integrated Circuits and Systems, chapter 2. Springer, 2011.

20. C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

21. L. Lamport. How to make a multiprocessors computer that correctly executes multiprocessors programs. *IEEE Transaction on Computers*, C-28(9):690–691, September 1979.

22. Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and management of 3 D chip multiprocessors using network-in-memory. *ACM SIGARCH Computer Architecture News*, 34(2):130–141, 2006.

23. C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design and Test of Computers*, 22(6):556–564, November-December 2005.

24. Gabriel Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings for the 35th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2008.

25. G. L. Loi, B. Agarwal, N. Srivastava, S.-C. Lin, and T. Sherwood. A thermally-aware performance analysis of vertically integrated 3-D processor memory hierarchy. In *Proceedings of the 43rd Desigfn Automation Conference*, 2006.

26. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Scalability of transaction counter based relaxed consistency models in NoC based multicore architectures. *ACM SIGARCH Computer Architecture News*, December 2009.

27. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Scalability of weak consistency in NoC based multicore architectures. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, June 2010.

28. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Realization and performance comparison of sequential and weak memory consistency models in network-on-chip based multi-core systems. In *Proceedings of the 16th Asian Pacific Design Automation Conference (ASP-DAC)*, Tokyo, Japan, January 2011.

29. T.G. Rauscher and P.M. Adams. Microprogramming: A tutorial and survey of recent developments. *Computers, IEEE Transactions on*, C-29(1):2 –20, January 1980.

30. Chuan Seng Tan. Three-dimensional integration of integrated circuits - and introduction. In Abbas Sheibanyrad, Frédéric Pétrot, and Axel Jantsch, editors, *3D Integartion for NoC-based SoC Architectures*, Integrated Circuits and Systems, chapter 1. Springer, 2011.

31. R. Stanley Williams. How we found the missing memristor. *IEEE Spectrum*, December 2008.

32. Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

33. Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.

34. Yuang Zhang, Zhonghai Lu, Axel Jantsch, Li Li, and Minglun Gao. Towards hierarchical cluster based cache coherence for large-scale network-on-chip. In *Proceedings of the 4th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, Cairo, Egypt, April 2009.