Hybrid Distributed Shared Memory Space in Multi-core Processors

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

Abstract— **On multi-core processors, memories are preferably distributed and supporting Distributed Shared Memory (DSM) is essential for the sake of reusing huge amount of legacy code and easy programming. However, the DSM organization imports the inherent overhead of translating virtual memory addresses into physical memory addresses, resulting in negative performance. We observe that, in parallel applications, different data have different properties (*private* or *shared*). For the private data accesses, it's unnecessary to perform Virtual-to-Physical address translations. Even for the same datum, its property may be changeable in different phases of the program execution. Therefore, this paper focuses on decreasing the overhead of Virtual-to-Physical address translation and hence improving the system performance by introducing hybrid DSM organization and supporting run-time partitioning according to the data property. The hybrid DSM organization aims at supporting fast and physical memory accesses for private data and maintaining a global and single virtual memory space for shared data. Based on the data property of parallel applications, the run-time partitioning supports changing the hybrid DSM organization during the program execution. It ensures fast physical memory addressing on private data and conventional virtual memory addressing on shared data, improving the performance of the entire system by reducing virtual-to-physical address translation overhead as much as possible. We formulate the run-time partitioning of hybrid DSM organization in order to analyze its performance. A real DSM based multi-core platform is also constructed. The experimental results of real applications show that the hybrid DSM organization with run-time partitioning demonstrates performance advantage over the conventional DSM counterpart. The percentage of performance improvement depends on problem size, way of data partitioning and computation/communication ratio of parallel applications, network size of the system, etc. In our experiments, the maximal improvement is 34.42%, the minimal improvement 3.68%.**

Index Terms— **Run-time Partitioning, Hybrid Distributed Shared Memory (DSM), Multi-core, processor**

I. INTRODUCTION

As a general trend, processor development has been shifted from single sequential processor to parallel multi-core systems [1] [2]. NoC based multi-core systems are promising solutions to the modern and future processor design challenges [3] [4] [5]. For instance, in 2007, Intel researchers announced their research prototype multi-core architecture containing 80 tiles arranged as a 10x8 2D mesh network [6]. In multi-core processors, especially for medium and large scale system sizes, memories are preferably distributed, featuring good scalability and fair contention and delay of memory accesses, since the centralized memory has already become the bottleneck of performance, power and cost [7]. In order to reuse huge amount of legacy code and facilitate programming, it's essential to support Distributed but Shared Memory (DSM). 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.

The key technique of Distributed Shared Memory organization is to maintain an address mapping table of translating virtual addresses into physical addresses and hence to implicitly access remote shared data and to provide software programmers with a transparent and global shared memory space. The Virtual-to-Physical address translation table (V2P Table) fully reveals how the DSM space is organized. However, the Virtual-to-Physical (V2P) address translation costs time, which is the inherent overhead of DSM organization. Every memory access operation contains a V2P address translation, increasing the system's processing time and hence limiting the performance. We observe that different data in parallel applications have different properties (*private* or *shared*) and it's unnecessary to introduce V2P address translations for private data accesses. According to the data property, we can obtain two observations:

(1) During the entire execution of parallel applications, some data processed by the local processor core are shared and need to be accessed by other remote processor cores, while other data are private and only

This paper is based on "Run-time Partitioning of Hybrid Distributed Shared Memory on Multi-core Network-on-Chips," by X. Chen, Z. Lu, A. Jantsch, and S. Chen, which appeared in the Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms, and Programming (PAAP), Dalian, China, December 2010. © 2010 IEEE.

The research is partially supported by the FP7 EU project MOSART (No. IST-215244), the Major Project of "Core electronic devices, High-end general purpose processor and Fundamental system software" in China (No. 2009ZX01034-001-001-006), the Innovative Team of High-performance Microprocessor Technology (No. IRT0614), the National Natural Science Foundation of China (No. 61070036), and the National 863 Program of China (No. 2009AA011704).

doi:10.4304/jsw.6.12.2369-2378

used by the local processor core.

(2) Some data are private and only accessed by the local processor core in a certain phase of the entire execution of parallel applications. However, they may change to be shared and accessible to other remote processor cores in another phase of the entire program execution.

The conventional DSM organizes all memories as shared ones, regardless of whether the processed data are only used by its local processor core or not. That is, each memory access operation in the system with conversional DSM organization includes a translation of virtual address to physical address, even if the accessed object is just local. If we get rid of the address translation overhead when the processor core handles the data which only belong to it (i.e. the data are *private*), the system's performance is expected to improve.

Motivated by aforementioned considerations, regarding the observation (1), we introduce a hybrid organization of Distributed Shared Memory in the paper. The philosophy of our hybrid DSM organization is to support fast and physical memory accesses for private data as well as to maintain a global and single virtual memory space for shared data. Considering the observation (2), we propose a run-time partitioning technique. This technique supports programmable boundary partitioning of private region and shared region of the Local Memory to change the hybrid DSM organization, based on the data property of parallel applications. It ensures fast physical memory addressing on private data and conventional virtual memory addressing on shared data, improving the performance of the entire system by reducing V2P address translation overhead as much as possible. We analyze its performance by formulating the run-time partitioning of hybrid DSM organization. The experimental results of real applications show that the hybrid DSM organization with run-time partitioning demonstrates performance advantage over the conventional DSM counterpart. The percentage of performance improvement depends on problem size, way of data partitioning and computation/communication ratio of parallel applications, network size of the system, etc. In our experiments, the maximal improvement is 34.42%, the minimal improvement 3.68%.

The rest of the paper is organized as follows. Section II discusses related work. Section III introduces our multi-core NoC platform and its hybrid DSM organization. Run-time partitioning is proposed in Section IV. Section V reports simulation results with application workloads. Finally we conclude in Section VI.

II. Related Work

As one form of memory organization, Distributed Shared Memory (DSM) has been attracting a large body of researches. However, we note that up to today there are few researches on NoC based multi-core chips. In [8], our previous work implemented a Dual Microcoded Controller to support flexible DSM management on multi-core processors. It off-loads DSM management from the main-processor to a programmable co-processor. In [9], Matteo explored a distributed shared memory architecture suitable for low-power on-chip multiprocessors. His work focused on the energy/delay exploration of on-chip physically distributed and logically shared memory address space for MPSoCs based on a parameterizable NoC. In our view, we envision that there is an urgent need to support DSM because of the huge amount of legacy code and easy programming. Monchiero also pointed out that the shared memory constitutes one key element in designing MPSoCs (Multiprocessor System-on-Chips), since its function is to provide data exchange and synchronization support [9]. Therefore, in the paper, we focus on the efficient organization and run-time partitioning of DSM space on multi-core processors.

Regarding memory partitioning, in [10], Xue explores a proactive resource partitioning scheme for parallel applications simultaneously exercising the same MPSoC system. His work combined memory partitioning and processor partitioning and revealed that both are very important to obtain best system performance. In [11], Srinivasan presented a genetic algorithm based search mechanism to determine a system's configuration on memory and bus that is energy-efficiency. Both Xue and Srinivasan addressed memory partitioning in combination with other factors, e.g. processors and buses. In [12], Mai proposed a function-based memory partitioning method. Based on pre-analysis of application programs, his method partitioning memories according to data access frequencies. Different from Xue's, Srinivasan's and Mai's work, this paper considers memory organization and partitioning according to data property of real applications running on multi-core processors. In [13], Macii presented an approach, called address clustering, for increasing the locality of a given memory access profile, and thus improving the efficiency of partitioning and the performance of system. In the paper, we improve the system performance by partitioning the DSM space into two parts: *private* and *shared*, for the sake of speeding up frequent physical accesses as well as maintaining a global virtual space. In [14], Suh presented a general partitioning scheme that can be applied to set-associative caches. His method collects the cache miss characteristics of processes/threads at run-time so that partition sizes are varied dynamically to reduce the total number of misses. We also adopt the run-time adjustment policy, but we address partitioning the DSM dynamically according to the data property in order to reduce Virtual-to-Physical (V2P) address translation overhead. In [15], Qiu also considered optimizing the V2P address translation in a DSM based multiprocessor system. However, the basic idea of his work is to move the address translation closer to memory so that the TLBs (Translation Lookaside Buffers: supporting translation from virtual to physical addresses) do not have consistency problems and can scale well with both the memory size and the number of processors. In the paper, we address reducing the total V2P address transaction overhead of the system as much

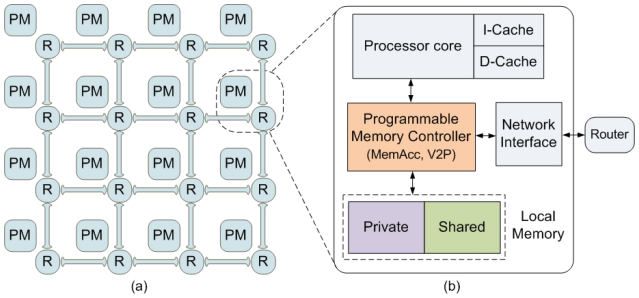

Figure 1. (a) Multi-core Processors, and (b) Processor-Memory Node

as possible by ensuring physical accesses on private data.

III. Multi-core Processors with Hybrid Distributed Shared Memory

In our multi-core processors, memories are distributed at network nodes but partly shared. Fig. 1 (a) shows an example of our multi-core processors with hybrid organization of Distributed Shared Memory. 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 [16]. The microarchitecture of a PM node is illustrated in Fig. 1 (b). Each PM node consists of a processor core with tightly coupled caches, a network interface, a programmable memory controller and a local memory. As can be observed, memories are distributed in each node and tightly integrated with processors. All local memories can logically form a single global memory address space. However, we do not treat all memories as shared. As illustrated in Fig. 1 (b), the local memory is partitioned into two parts: *private* and *shared*. And two addressing schemes are introduced: *physical addressing* and *logic (virtual) addressing*. The private memory can only be accessed by the local processor core and it's physical. All of shared memories are visible to all PM nodes and organized as a DSM addressing space and they are virtual. For shared memory access, there requires a Virtual-to-Physical (V2P) address translation. Such translation incurs overhead. The philosophy of our hybrid DSM organization is to support fast and physical memory accesses for frequent private data as well as to maintain a global and single virtual memory space for shared data.

We illustrate the organization and address mapping of hybrid DSM in Fig. 2. As can be seen, a PM node may use both physical and logical addresses for memory access operations. Physical addresses are mapped to the local private memory region, and logical addresses can be mapped to both local shared and remote shared memory regions. For ***#k*** Node, its hybrid DSM space is composed of two parts. The first one is its private memory which is physical addressing. So the logic address is equal to the physical address in the private memory. The second part maps all shared memories. The mapping order of these shared memories is managed by the Virtual-to-Physical address translation table. Different PM nodes may have different hybrid DSM space. For instance, in the hybrid

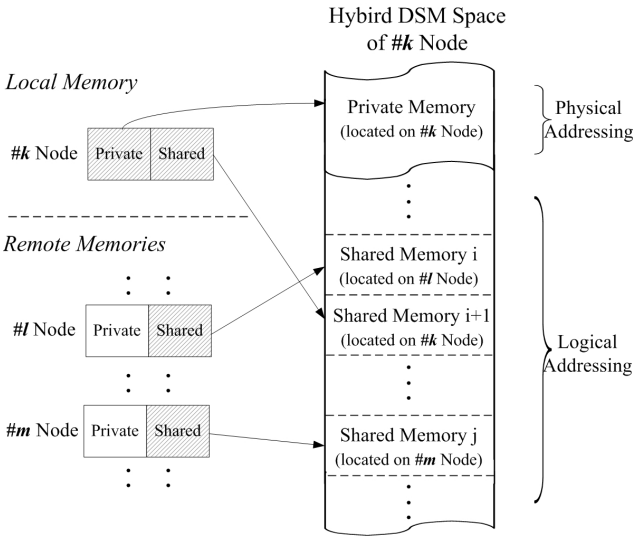

Figure 2. Hybrid organization of Distributed Shared Memory

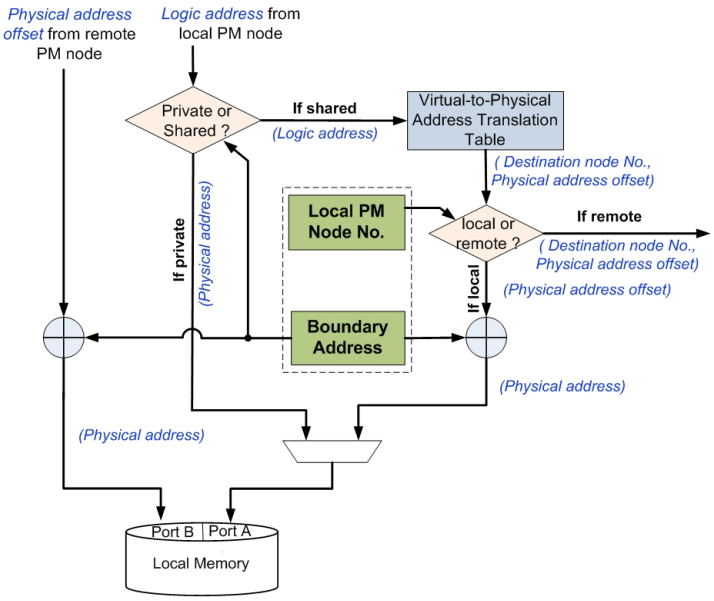

Figure 3. memory addressing flow

DSM space of ***#k*** Node, its local shared memory region is mapped logically as Shared Memory $i+1$ following Shared Memory i which is corresponding to the remote shared memory region in ***#l*** Node.

To support the hybrid DSM organization, the multi-core architecture maintains two registers for each PM node: **Local PM Node No.** and **Boundary Address** (see in Fig. 3). In each PM node, **Local PM Node No.** denotes the number of the local PM node, while **Boundary Address** denotes the address of boundary of the private region and the shared region in the Local Memory. **Boundary Address** can be configured at run-time to support dynamic re-organization of hybrid DSM space.

Fig. 3 shows the memory addressing flow of each PM node. As shown in the figure, each PM node can respond to two memory access requests concurrently from the local PM node and the remote PM node via the network.

In the beginning, the local PM node starts a memory access in its hybrid DSM space. The memory address is logic. The local PM node firstly distinguishes whether the address is private or shared. If private, the requested memory access hits the private memory. In the private memory, the logic address is just the physical address, so the address is forwarded directly to the Port A of the Local Memory. If the memory access is "write", the datum is stored into the Local Memory in the next clock cycle. If the memory access is "read", the data is loaded out of the Local Memory in the next clock cycle. The physical addressing is very fast. In contrast, if the memory address is shared, the requested memory access hits the shared part of the hybrid DSM space. The memory address first goes into the Virtual-to-Physical address translation table (V2P Table). The V2P Table records how the hybrid DSM space is organized and is responsible for translating the logic address into two pieces of useful information: *Destination Node No.* and *Physical Address Offset*. As shown in Fig. 2, the shared part of hybrid DSM space is composed by all shared memory regions of all PM nodes. *Destination Node No.* is used to obtain which PM node's shared memory is hit by the requested memory address. Once the PM node with the target shared memory is found, *Physical Address Offset* helps position the target memory location. *Physical Address Offset* plus **Boundary Address** in the target PM node equals the physical address in the target PM node. As shown in the figure, *Destination Node No.* and *Physical Address Offset* are obtained out of the V2P Table. Firstly, we distinguish wether the requested memory address is local or remote by comparing *Destination Node No.* with **Local Node No.**. If local, *Physical Address Offset* is added by **Boundary Address** in the local PM node to get the physical address. The physical address is forwarded to the Port A of the Local Memory to accomplish the requested memory access. If the requested memory access is remote, *Physical Address Offset* is routed to the destination PM node via the on-chip network. Once the destination PM node receives remote memory access request, it adds the *Physical Address Offset* by the **Boundary Address** in it to figure out the physical address. The physical address is forwarded to the Port B of the Local Memory. If the requested memory access is "write", the data is stored into the Local Memory. If the requested memory access is "read", the data is loaded from the Local Memory and sent back to the source PM node.

IV. Run-time Partitioning

Our multi-core processors support configuring **Boundary Address** at run-time in order to dynamically adjust the DSM organization when the program is running. Programmers can insert the following inline function in their C/C++ program to change the **Boundary Address** register.

```
void Update_BADDR(unsigned int Value);
```

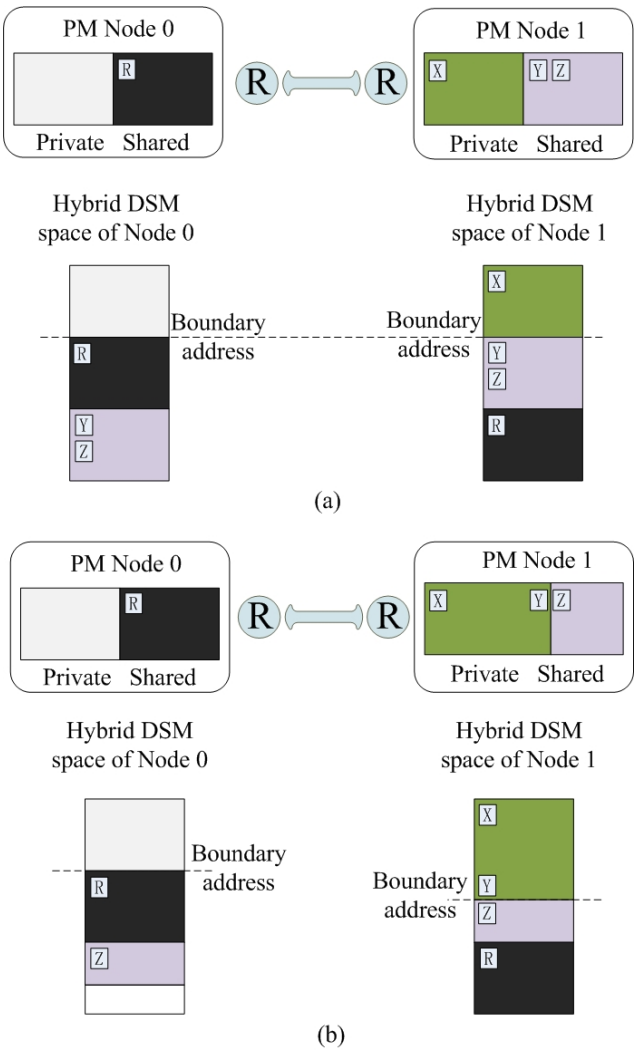

Figure 4. A short example of run-time DSM partitioning

A. Idea Description

Fig. 4 shows a short example of run-time partitioning of hybrid DSM space. Assume that there are two PM nodes in a 1x2 network. The hybrid DSM space of PM Node 0 contains its private memory region, its shared memory region and the shared memory region of PM Node 1, while the hybrid DSM space of PM Node 1 equals its Local Memory plus the shared memory region of PM Node 0. In the beginning (see Fig. 4 (a)), datum 'X' is in the private region of the Local Memory in PM Node 1, while datum 'Y' and 'Z' are in the shared region of the Local Memory in PM Node 1. Therefore, 'X' is invisible to PM Node 0, while 'Y' and 'Z' are accessible to PM Node 0. When PM Node 0 access 'Y', the *Destination Node No.* (i.e. PM Node 1) and *Physical Address Offset* of 'Y' are obtained from the V2P Table of PM Node 0 in the first phase. A remote request of memory access is sent to PM Node 1. Once PM Node 1 receives the request, the *Physical Address Offset* plus the **Boundary Address** in PM Node 1 equals the real physical address of 'Y'. Assume that the **Boundary Address** in PM Node 1 is re-configured to a larger value during the program

execution (i.e, the private memory part is enlarging) so that 'Y' becomes private. In this situation (see Fig. 4 (b)), PM Node 0 cannot access 'Y'. The procedure illustrated by Fig. 4 can be used to improve the system performance. For instance, PM Node 0 acts as a producer, PM Node 1 as a consumer. PM Node 0 produces several data which will be consumed by PM Node 1. In the beginning, the property of these data are *shared*, PM Node 0 firstly stores them into the shared region of the Local Memory of PM Node 1. After that, these data are only used by PM Node 1, it's unnecessary for PM Node 1 to access them in logic addressing mode. By changing the boundary address, we can make them be private. The Virtual-to-Physical address translation overhead is averted so that the system performance is improved.

The boundary address configuration is flexible to software programmers. When changing the boundary address, programmers need to pay attention to guarantee memory consistency. In the example of Fig. 4, changing the boundary address must be after PM Node 0 accomplish storing 'Y'. This can be guaranteed by inducing a synchronization point before PM Node 1 starts re-writing the **Boundary Address** register.

B. Producer-Consumer Mode

With concurrent memory addressing, the run-time partitioning provides software programmers with a *Producer-Consumer Mode*. The *Producer-Consumer Mode* features low-latency and tightly-coupled data transmission between two PM nodes. Concurrent memory addressing allows that one PM node produces data and in the meanwhile another PM node consumes the data that have been produced. During data transmission, the property of transmitted data may be changed. The run-time partitioning dynamically adjusts the DSM organization according to the changed data property and hence reduce V2P address translation overhead so as to improve the efficiency and performance of data transmission.

Fig. 5 illustrate the *Producer-Consumer Mode* under two cases. Assume that there are two PM nodes in a 2x1 network and m data blocks transferred from PM Node 0 to PM Node 1. The hybrid DSM space of PM Node 0 contains its private memory region, its shared memory region and the shared memory region of PM Node 1, while the hybrid DSM space of PM Node 1 equals its Local Memory plus the shared memory region of PM Node 0. The PM Node 0 produces m data blocks one by one, while the PM Node 1 consumes the m data blocks one by one once one of them are ready.

In Fig. 5, the m data blocks are located in the Local Memory of PM Node 1. As shown in Fig. 5 (a), all data blocks are with the property of "*shared*" since all data blocks haven't been produced by PM Node 0 yet. The **Boundary Address** indicates that B_1, ..., B_m are in the shared memory region of PM Node 1 and accessible to PM Node 0. In the beginning, PM Node 0 produces B_1. The values of B_1 are written into the shared memory region of PM Node 1. After PM Node 0 finishes producing B_1, there is a synchronization point that informs PM Node 1 of the readiness of B_1. Once B_1 are ready

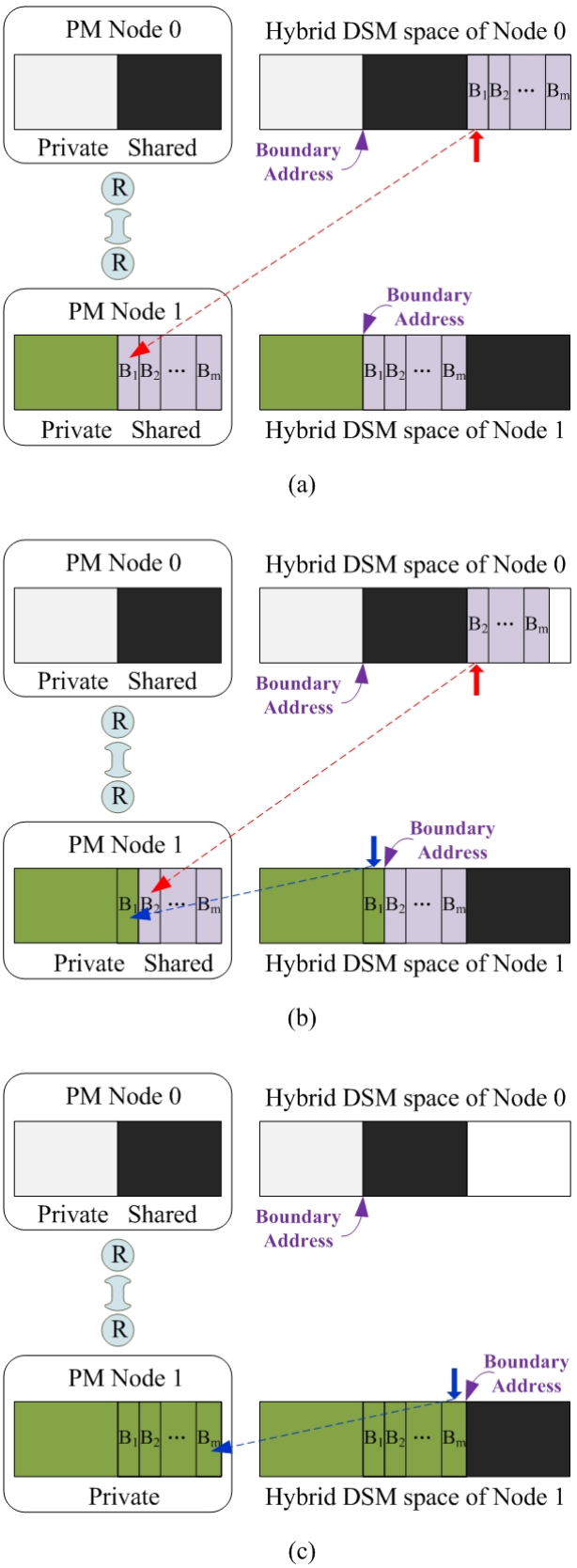

Figure 5. Producer-Consumer Mode

(see Fig. 5 (b)), PM Node 1 re-configures its **Boundary Address** to make B_1 be "*private*". After the run-time boundary address configuration, the hybrid DSM spaces of PM Node 0 and PM Node 1 are both changed. PM Node 0 only can access B_2, ..., B_m rather than B_1, and PM Node 1 can access B_1 in fast *physical addressing* scheme and hence eliminate V2P address translation overhead that would be induced in conventional DSM organization. The Programmable Memory Controller (see Fig. 1) allows concurrent memory references from both the local PM node and the remote PM node via the on-chip network. As shown in Fig. 5 (b), while PM Node 0 produces B_2, PM Node 1 is consuming B_1. The rest can be done in the same manner. So while PM Node 0 produces B_i, PM Node 1 is consuming B_{i-1} (i = 2,...,m). At last, PM Node 1 is consuming B_m, as shown in Fig. 5 (c). A this time, all data blocks becomes "*private*" and the entire Local Memory of PM Node 1 is private and invisible to PM Node 0. The hybrid DSM space of PM Node 0 only contains its Local Memory.

In Fig. 5, assume that the total transferred data size is fixed. Larger block size and hence smaller block number results in less overhead of run-time boundary address configuration but less concurrency. Less overhead of run-time boundary address configuration leads to positive performance, less concurrency negative performance. Therefore, there is a tradeoff to determine a optimized block size and boundary address configuration number.

C. Performance Analysis

Following the example shown in Fig. 4, we formulate the run-time partitioning of hybrid DSM organization of PM Node 1 and discuss its performance. To facilitate the analysis and discussion, we first define a set of symbols in TABLE I.

TABLE I.
DEFINITIONS AND NOTATIONS

N	data size processed by PM Node 1.
x	ratio of *private* data (e.g. 'X' in Fig. 4) to total data.
y	ratio of data with changeable property (e.g. 'Y' in Fig. 4) to total data.
z	ratio of *local shared* data (e.g. 'Z' in Fig. 4) to total data.
r	ratio of *remote shared* data (e.g. 'R' in Fig. 4) to total data.
t_{mem}	cycles of accessing the Local Memory once.
t_{v2p}	cycles of Virtual-to-Physical address translation.
t_{com}	cycles of network communication.
t_p	cycles of accessing a *private* datum. ($t_p = t_{mem}$)
t_{ls}	cycles of accessing a *local shared* datum. ($t_{ls} = t_{v2p} + t_{mem}$)
t_{rs}	cycles of accessing a *remote shared* datum. ($t_{rs} = t_{v2p} + t_{mem} + t_{com}$)
t_{bp}	cycles of changing the **Boundary Address** once.
m	number of boundary address configuration during program execution.

We firstly formulate the data access delay of conventional DSM organization. Memories are thought to be shared in conventional DSM organization and hence V2P address translation overhead is involved in every local or remote memory access. In Fig. 4, PM Node 1's accessing 'X', 'Y', 'Z', and 'R' includes V2P address translation overhead. Therefore, we can obtain its data access delay by Formula (1) below.

$$\begin{aligned} T_1 &= N \cdot (x + y + z) \cdot t_{ls} + N \cdot r \cdot t_{rs} \\ &= N \cdot t_{mem} + N \cdot t_{v2p} + N \cdot r \cdot t_{com} \end{aligned} \tag{1}$$

Formula (1) is subject to

$$x + y + z + r = 1$$

For our introduced hybrid DSM organization and its run-time partitioning, Virtual-to-Physical address translation overhead is only included when shared data are accessed. Therefore, in Fig. 4, PM Node 1's accessing 'X' and 'Y' doesn't need V2P address translation. However, the data access delay contains the extra overhead of changing the boundary address. We can get the data access delay by Formula (2) below.

$$\begin{aligned} T_2 &= N \cdot (x + y) \cdot t_p + N \cdot z \cdot t_{ls} + N \cdot r \cdot t_{rs} + t_{bp} \cdot m \\ &= N \cdot t_{mem} + N \cdot (z + r) \cdot t_{v2p} + N \cdot r \cdot t_{com} + t_{bp} \cdot m \end{aligned} \tag{2}$$

Formula (2) is subject to

$$\begin{cases} x + y + z + r = 1 \\ m \leqslant N \cdot y \end{cases}$$

Then, we can obtain the performance gain (γ: defined as average reduced execution time of accessing a datum) by Formula (3) below.

$$\gamma = \frac{T_1 - T_2}{N} = (x + y) \cdot t_{v2p} - \frac{t_{bp} \cdot m}{N} \tag{3}$$

From Formula (3), we can see that

(i) Compared with conventional DSM organization, our hybrid DSM organization gets rid of Virtual-to-Physical (V2P) address translation of private memory accesses and hence obtain performance improvement. Run-time partitioning further eliminate the V2P address translation of memory accesses of data whose data property is changeable, but it induces extra overhead of changing the boundary address.

(ii) If private data or data whose data property is changeable take a lager proportion in parallel programs (i.e. x and y increase), the hybrid DSM organization demonstrates higher performance advantage.

(iii) For the same size of parallel programs (x, y, z, and r are fixed) on hybrid DSM organization, hardware implementation of V2P address translation obtains higher performance gain than software implementation (t_{v2p} of hardware solution is greater than that of software solution).

(iv) The re-configuration of **Boundary Address** induces negative performance. However, avoiding frequently changing the boundary address results in little value of $\frac{t_{bp} \cdot m}{N}$ and hence alleviates its negative effect on performance.

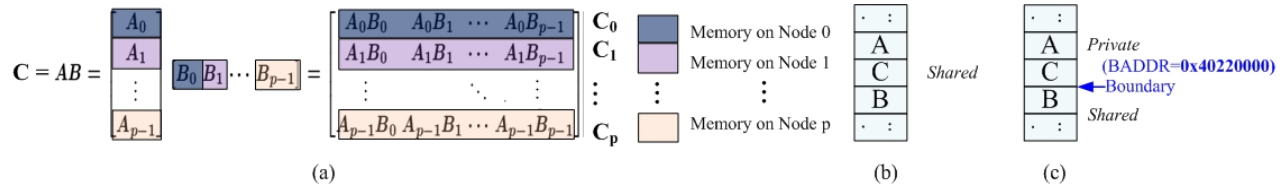

Figure 6. (a) Memory allocation for matrix multiplication, (b) conventional DSM organization, and (c) Hybrid DSM organization with run-time partitioning

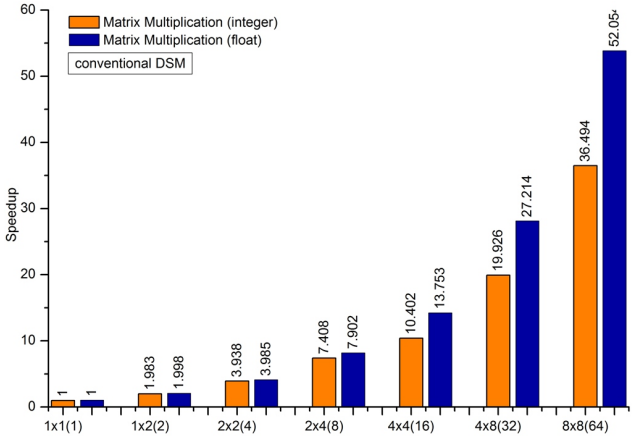

Figure 7. Speedup of matrix multiplication

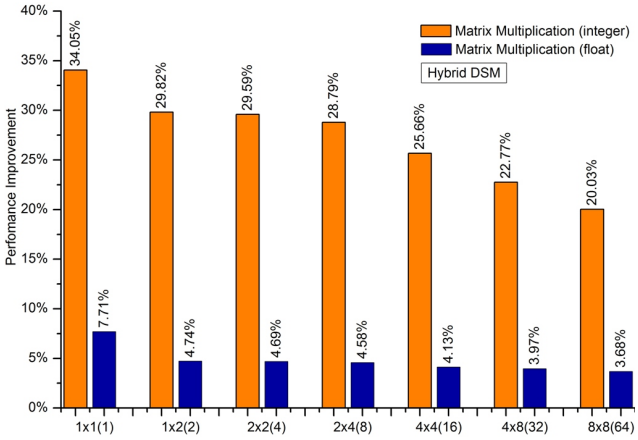

Figure 8. Performance improvement for matrix multiplication

V. Experiments and Results

A. Experimental Platform

We constructed a multi-core experimental platform as shown in Fig. 1. The multi-core processor uses the LEON3 [17] as the processor in each PM node and uses the Nostrum NoC [18] as the on-chip network. The LEON3 processor core is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Nostrum NoC is a 2D mesh packet-switched network with configurable size. It serves as a customizable platform.

B. Application 1: Matrix Multiplication

The matrix multiplication calculates the product of two matrix, A[64, 1] and B[1, 64], resulting in a C[64, 64] matrix. We consider both integer matrix and floating point matrix. As shown in Fig. 6 (a), matrix A is decomposed into p equal row sub-matrices which are stored in p nodes respectively, while matrix B is decomposed into p equal column sub-matrices which are stored in p nodes respectively. The result matrix C is composed of p equal row sub-matrices which are respectively stored in p nodes after multiplication.

Fig. 6 (b) shows the conventional DSM organization and all data of matrix A, C and B are shared. Fig. 6 (c) shows the hybrid DSM organization. The data of matrix A and C are private and the data of matrix B are shared. Because the sub-matrices of matrix A and C are only accessed by their own host PM node and matrix B are accessed by all PM nodes. In this experiment, the system size increases by a factor of 2 from 1, 2, 4, 8, 16, 32 to 64. While mapping the matrix multiplication onto multiple cores, we perform a manual function partitioning and map the functions equally over the cores.

Fig. 7 shows the performance speedup of the matrix multiplication with conventional DSM organization. When the system size is increased from 1 to 2, 4, 8, 16, 32 and 64, the application speedup (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).) is from 1 to 1.983, 3.938, 7,408, 10.402, 19.926 and 36.494 for the integer computation, and from 1, 1.998, 3.985, 7.902, 13.753, 27.214, 52.054 for the floating point computation. The relative speedup for the floating point multiplication is higher than that for the integer computation. This is as expected because when increasing the computation time, the portion of communication delay becomes less significant, thus achieving higher speedup. Fig. 8 shows performance improvement of the hybrid DSM organization with respect to the conventional DSM organization. We calculate the performance improvement using the following formula:

$$\text{Perf. Impr.} = \frac{\text{Speedup}_{\text{hybrid DSM}} - \text{Speedup}_{\text{conventional DSM}}}{\text{Speedup}_{\text{hybrid DSM}}}$$

For the integer matrix, the performance increases by 34.05%, 29.82%, 29.59%, 28.79%, 25.66%, 22.77%, and 20.03%, for 1x1, 1x2, 2x2, 2x4, 4x4, 4x8 and 8x8 system size, respectively. For the floating point matrix, the performance is increased by 7.71%, 4.74%, 4.69%, 4.58%, 4.13%, 3.97%, and 3.68% for 1x1, 1x2, 2x2,

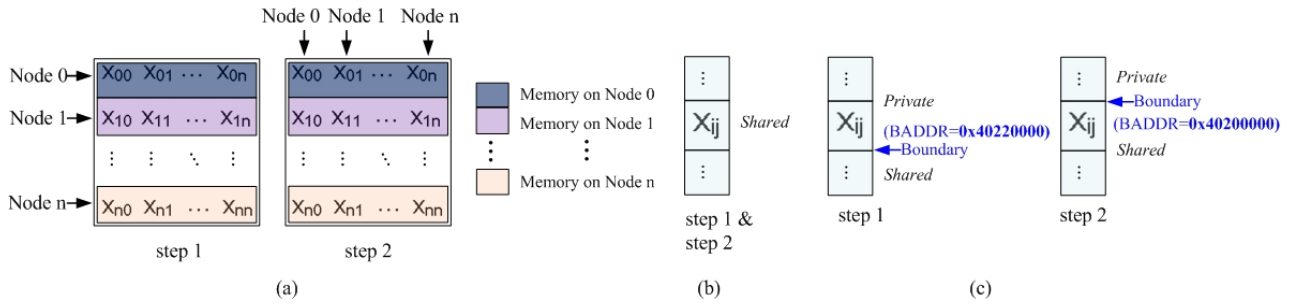

Figure 9. (a) Memory allocation for 2D DIT FFT, (b) conventional DSM organization, and (c) Hybrid DSM organization with run-time partitioning

2x4, 4x4, 4x8 and 8x8 system size, respectively. The improvement for the floating point is lower because the floating point has a larger percentage of time spent on computation, thus reducing the communication time in memory accesses achieves less enhancement. Note that, the single core case has a higher improvement because all data accesses are local shared for the conventional DSM organization and private for the hybrid DSM organization.

C. Application 2: 2D DIT FFT

We implement a 2D radix-2 DIT FFT. As shown in Fig. 9 (a), the FFT data are equally partitioned into n rows, which are stored on the n nodes, respectively. According to the 2D FFT algorithm, the application first performs FFT on rows (Step 1). After all nodes finish row FFT (synchronization point), it starts FFT on columns (Step 2). We experiment on two DSM organizations. One is the conventional DSM organization, as shown in Fig. 9 (b), for which all FFT data are shared. The other is the hybrid DSM organization, as illustrated in Fig. 9 (c). Since the data used for row FFT calculations at step 1 are stored locally in each node and are only to be used for column FFT calculations at step 2, we can dynamically re-configure the boundary address (BADDR in Fig. 9) at run time, such that, the data are private at step 1 but become shared at step 2.

Fig. 10 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 2, 4,, 8, 16, 32, and 64, the speedup with the conventional DSM organization goes up from 1 to 1.905, 3.681, 7.124, 13.726, 26.153 and 48.776. The speedup for the hybrid DSM organization is normalized with the single core with the conventional DSM organization. As Fig. 10 shows, for the different system sizes, 1, 2, 4, 8, 16, 32 and 64, the performance improvement is 34.42%, 16.04%, 15.48%, 14.92%, 14.70%, 13.63% and 11.44%, respectively. Note that, the single core case has a higher improvement because all data accesses are local shared the conventional DSM organization and private for the hybrid DSM organization, and there is no synchronization overhead.

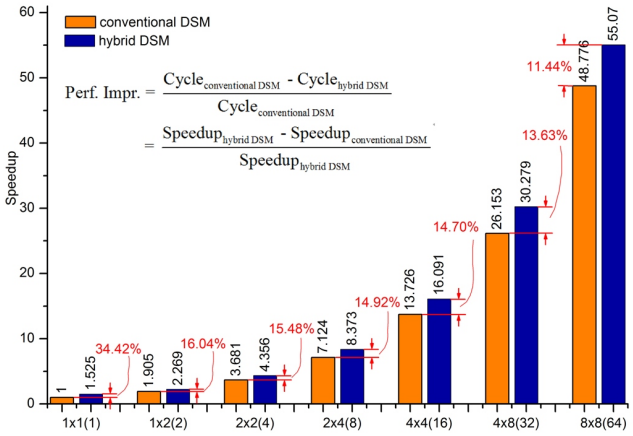

Figure 10. Speedup and performance improvement of 2D DIT FFT

D. Application 3: Wavefront Computation

Wavefront Computation is performed to validate the producer-consumer mode. Wavefront Computations are common in scientific applications. Given a matrix (see Fig. 11 (a)), the left and top edges of which are all 1, the computation of each remaining 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. We use the same method as [19] to parallelize the Wavefront Computation, the rows of the matrix are assigned to PM nodes in a round-robin fashion (see Fig.11 (b) and (c)). With this static scheduling policy, to compute an element, only the availability of its above neighbor needs to be checked (synchronized). For instance, PM node 0 computes the elements in row 1. PM node 1 cannot compute the elements in row 2 until the corresponding elements in row 1 has been figured out by PM node 0. After finishing the computation in row 1, PM node 0 goes on to compute the elements in row 3 according to the round-robin scheduling policy. In our experiment, we conduct a 64×64 matrix in two DSM organization.

For conventional DSM organization, as shown in Fig. 11 (b), all elements are *shared* and memory accesses on them incur V2P address translation overhead. For hybrid DSM organization, we apply *Producer-Consumer Mode* described in Subsection IV-B. As Fig. 11 (c)

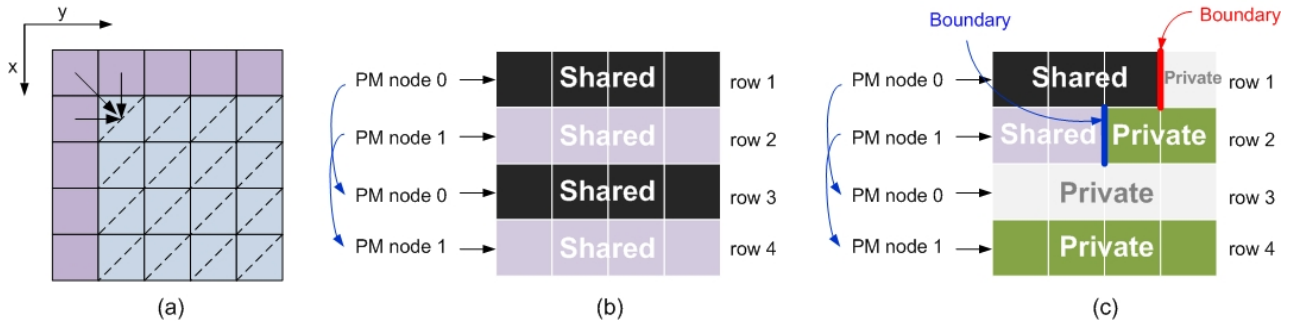

Figure 11. (a) Memory allocation for Wavefront Computation, (b) conventional DSM organization, and (c) Hybrid DSM organization with run-time partitioning

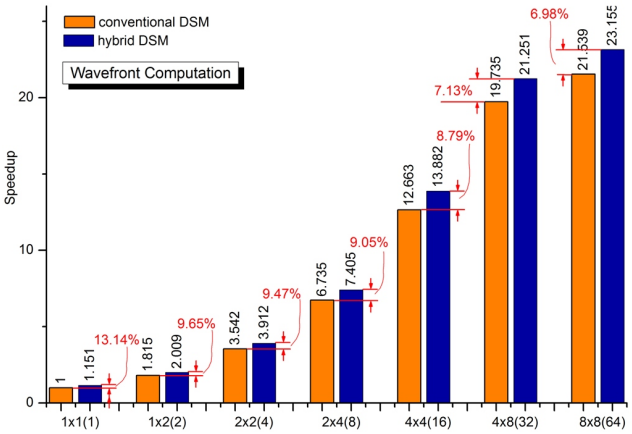

Figure 12. Speedup and performance improvement of Wavefront Computation

illustrates, the elements, which haven't been computed by their host PM node yet, are *private*. The host PM node accesses them in fast *physical addressing scheme*. After the host PM node accomplishes computation on them, it re-configure the boundary address to make them be *shared* so that other PM node could access them. There is a synchronization point after the host PM node re-configure the boundary address. Compared with aforementioned Matrix Multiplication and 2D DIT FFT, Wavefront Computation has a number of boundary address adjustment.

Fig. 12 shows the speedup and performance improvement of the Wavefront Computation. As the system size increases from 1 to 2, 4,, 8, 16, 32, and 64, the speedup with conventional DSM organization goes up from 1 to 1.815, 3.542, 6.735, 12.663, 19.735 and 21.539 and the speedup with hybrid DSM organization is from 1.151 to 2.009, 3.912, 7.405, 13.882, 21.251 and 23.155. Since synchronization-intensive, the speedup of Wavefront Computation is lower than that of Matrix Multiplication and 2D DIT FFT, especially as the system size becomes larger. To calculate the performance gain induced by hybrid DSM organization, the speedup for hybrid DSM organization is normalized with the execution time of single PM node in conventional DSM organization. As is shown in Fig. 12 , for the different system sizes of 1, 2, 4, 8, 16, 32 and 64, the performance improvement is 13.14%, 9.65%, 9.47%, 9.05%, 8.79%, 7.13% and 6.98%, respectively.

From Fig. 12, we can see that

- In Wavefront Computation algorithm, all PM nodes act as both a producer and a consumer. As shown in Fig. 11 (b) and (c), PM node 0 acts as a producer who computes the elements in row 1. PM node 1 acts as a consumer who uses the elements in row 1. Meanwhile PM node 1 also a producer who computes the elements in row 2 which are used by PM node 0 when it computes the elements in row 3. It fits the *producer-consumer mode* in Subsection IV-B. Using *producer-consumer mode* of run-time partitioning in hybrid DSM organization, we could reduce and hide V2P address translation overhead by fast *physical addressing scheme* and concurrent memory addressing, respectively.
- In conventional DSM organization, the obtained speedup is 21.539 for 8×8, a little higher than 19.735 for 4×8. It's the same with hybrid DSM organization. This is because synchronization overhead (it's mainly waiting time) and network communication delay become dominating as the system size is scaled up gradually and degrade the performance.
- The single PM node case has a higher improvement because all data accesses are local shared for the conventional DSM organization and private for the hybrid DSM organization, and there is no synchronization overhead.

VI. Concluding Remark

DSM organization offers ease of programming by maintaining a global virtual memory space as well as imports the inherent overhead of Virtual-to-Physical (V2P) address translation, which leads to negative performance. Observing that it's unnecessary to perform V2P address translation for private data accesses, this paper introduces hybrid DSM organization and run-time partitioning technique in order to improve the system performance by reducing V2P address translation overhead as much as possible. The philosophy of our hybrid DSM organization is to support fast and physical memory accesses for private data as well as to maintain a global and single virtual memory space for shared data. The run-time partitioning supports re-configuration of the hybrid DSM organization

by dynamically changing the boundary address of private memory region and shared memory region during the program execution. The experimental results of real applications show that the hybrid DSM organization with runtime partitioning demonstrates performance advantage over the conventional DSM counterpart. The percentage of performance improvement depends on problem size, way of data partitioning and computation/communication ratio of parallel applications, system size, etc. In our experiments, the maximal improvement is 34.42%, the minimal improvement 3.68%. In the future, we shall extend our work to cover more applications.

REFERENCES

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

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

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

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

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

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

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

[8] X. Chen, Z. Lu, A. Jantsch, and S. Chen, "Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller," in *Proc. of the Conf. on Design, automation and test in Europe (DATE'10)*, 2010, pp. 39–44.

[9] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architecture for noc-based multiprocessors," in *Proc. of the 2006 Int'l Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2006, pp. 144–151.

[10] L. Xue, O. Ozturk, F. Li, M. Kandemir, and I. Kolcu, "Dynamic partitioning of processing and memory resources in embedded mpsoc architectures," in *Proc. of the Conf. on Design, automation and test in Europe (DATE'06)*, 2006, pp. 690–695.

[11] S. Srinivasan, F. Angiolini, M. Ruggiero, L. Benini, and N. Vijaykrishnan, "Simultaneous memory and bus partitioning for soc architectures," in *Proc. of IEEE Int'l Conf. on SoC (SoCC'05)*, 2005, pp. 125–128.

[12] S. Mai, C. Zhang, and Z. Wang, "Function-based memory partitioning on low power digital signal processor for cochlear implants," in *Proc. of IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS'08)*, 2008, pp. 654–657.

[13] A. Macii, E. Macii, and M. Poncino, "Improving the efficiency of memory partitioning by address clustering," in *Proc. of the Conf. on Design, automation and test in Europe (DATE'03)*, 2003, pp. 18–23.

[14] G. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomputing*, vol. 28, no. 1, pp. 7–26, Apr. 2004.

[15] X. Qiu and M. Dubois, "Moving address translation closer to memory in distributed shared-memory multiprocessors," *IEEE trans. Parallel and Distributed Systems*, vol. 16, no. 7, pp. 612–623, 2005.

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

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

[18] A. Jantsch *et al.*, "The nostrum network-on-chip," in *http://www.ict.kth.se/nostrum*.

[19] W. Zhu, V. Sreedhar, Z. Hu, and G. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proc. of the 34th annual Int'l Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 35–45.

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. He is now a joint PhD candidate majoring Microelectronics both in the National University of Defense Technology (NUDT), Changsha, China, and KTH-Royal Institute of Technology, Stockholm, Sweden. His research interests include computer architecture, microarchitecture, VLSI design, system-on-chips, network-on-chips, Distributed Shared Memory.

Shuming Chen received BSc., MSc. and PhD. 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.

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 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.