Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 719–732
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Exploration of distributed shared memory architectures for NoC-based multiprocessors

Matteo Monchiero, Gianluca Palermo, Cristina Silvano *, Oreste Villa

Dipartimento di Elettronica e Informazione, Politecnico di Milano, I-20133 Milano, Italy

Received 24 November 2006; received in revised form 10 January 2007; accepted 15 January 2007
Available online 30 January 2007

Abstract

Multiprocessor system-on-chip (MP-SoC) platforms represent an emerging trend for embedded multimedia applications. To enable MP-SoC platforms, scalable communication-centric interconnect fabrics, such as networks-on-chip (NoCs), have been recently proposed. The shared memory represents one of the key elements in designing MP-SoCs to provide data exchange and synchronization support.

This paper focuses on the energy/delay exploration of a distributed shared memory architecture, suitable for low-power on-chip multiprocessors based on NoC. A mechanism is proposed for the data allocation on the distributed shared memory space, dynamically managed by an on-chip hardware memory management unit (HwMMU). Moreover, the exploitation of the HwMMU primitives for the migration, replication, and compaction of shared data is discussed. Experimental results show the impact of different distributed shared memory configurations for a selected set of parallel benchmark applications from the power/-performance perspective. Furthermore, a case study for a graph exploration algorithm is discussed, accounting for the effects of the core mapping and the network topology on energy and performance at the system level.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Multiprocessor systems-on-chip; Network-on-chip; Design space exploration; Low-power design

1. Introduction

Multiprocessor system-on-chip (MP-SoC) architectures are emerging as appealing solutions for embedded multimedia applications [10]. In general, MP-SoCs are composed of core processors, memories and some application-specific coprocessors. Communication is provided by advanced interconnect fabrics, such as high performance and efficient networks-on-chip (NoCs) [10]. To meet the system-level design constraints in terms of performance, cost and power consumption, a careful design of the memory subsystem and the communication layer is mandatory.

The main focus of this paper is the exploration of distributed shared memory architectures suitable for low-power on-chip multiprocessors based on the NoC approach. The three main characteristics of our target MP-SoC architecture are:

* Corresponding author. Tel.: +39 02 2399 3692; fax: +39 02 2399 3411.
E-mail address: silvano@elet.polimi.it (C. Silvano).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.01.008

- The communication, that is, based on the PIRATE NoC [9] architecture composed of a set of parameterized interconnection elements, that can generate different on-chip micro-network topologies. The router architecture is based on a crossbar switch, the switching policy is wormhole, the routing is distributed and the algorithm is static and table-based.
- The memory subsystem, that is, based on a Non Uniform Memory Access (NUMA) paradigm. The latency of memory accesses is non-uniform, because the system is built around a network interconnect and the memory space is composed on physically distributed and logically shared memory modules. Besides the shared memory space, the memory model is composed of private memory spaces for each processing element (PE). The shared portion of the addressing space represents one of the key elements of the MP-SoC to support synchronization and data exchange between PEs.
- The on-chip hardware memory management unit (HwMMU), designed to dynamically manage data allocation/deallocation on the physically distributed shared memory space. The memory management features are implemented in an hardware module so as to reduce the overhead due to the operating system memory management in terms of energy and performance. The on-chip HwMMU makes the memory utilization more efficient especially for embedded applications whose memory requirements can change significantly during run-time.

For the target multiprocessor architecture, the paper explores different on-chip network topologies and a variable number of distributed shared memory modules. Although the origins of the considered interconnect topologies and memory architectures can be tracked back to the field of multiprocessing systems, a different set of design constraints must be considered when adopting these architectures to the system-on-chip design context. Energy consumption and area constraints become mandatory as well as high throughput and low latency.

In this paper, our previous work [22] has mainly been extended in two directions. First, we discuss in Section 4.2 a HwMMU design supporting migration/replication/compaction policies for shared data. Second, we discuss in Section 5.3 an experimental case study regarding the analysis of a Breadth-First Search (BFS) algorithm for graph exploration. This case study takes into account the actual mapping of the system cores onto three different NoC topologies.

The design space exploration has been carried out by using *GRAPES* [31], a network-centric MP-SoC modelling, simulation and power estimation platform. GRAPES features a cycle-accurate simulation environment for power and performance evaluation of MP-SoC architectures based on a configurable network-on-chip. Our analysis enables us to identify power/performance trade-offs that represent critical issues for the design of network-based MP-SoCs. In particular, the experimental results show how an optimized design of the on-chip memory subsystem can save both system energy and execution time depending on the intrinsic nature of the target application (communication- or computation-oriented). The exploration of NoC topologies can identify energy/delay trade-offs and communication bottlenecks.

The paper is organized as follows. Some previous works are discussed in Section 2 while, in Section 3, the target system-on-chip multiprocessor architecture is presented, focusing on the interconnection and the memory subsystem. The proposed on-chip memory management unit is discussed in Section 4, while experimental results are shown and discussed in Section 5. Finally, some conclusions are pointed out in Section 6.

2. Related work

Most of the previous researches in embedded systems have focused on static allocation and how to synthesize memory hierarchies for a SoC [4]. For applications whose memory requirements change significantly during run-time, static allocation of memory makes the on-chip memory utilization inefficient and system modification very difficult [13].

Scratchpad memories have been proposed as an alternative to large L2 caches [15], especially for embedded systems. When adopting scratchpad memories, a software controlled local storage is provided instead of L2 cache. The shared memory design targeted in this paper can be seen as a set of scratchpad memories, managed as a typical multiprocessor shared memory.

Several works focus on scratchpad/L2 design. The authors of Ref. [14] present an algorithm to optimally solve the memory partitioning problem by dynamic programming. Regarding to multiprocessor systems, Kandemir et al. [20] present a com-

piler strategy to optimize data accesses in regular array-intensive applications for a system based on scratchpads. In [21], the authors propose compositional memory systems based on the partitioning and exclusive assignment of the L2 lines to running threads. The authors of [17,24] propose task/data migration schemes for NoC-based systems, based on compile-time profiling.

These approaches are based on static or, for more complex approaches, based on dynamic profiling of the application, but they are not so effective when the memory utilization changes dynamically with the system load, especially in shared memory multiprocessors [5]. On the other hand, the dynamic memory management can consume a great amount of the program execution time, especially for object-oriented applications, without providing deterministic timing behavior. Nevertheless, static techniques can be considered orthogonal to ours, and can be applied on the top of our hardware platform.

Many researchers have proposed hardware accelerators for dynamic memory management. The literature shows hardware implementations proposed in [12,28,1]. Chang et al. [27] have implemented the *malloc*(), *realloc*(), and *free*() C-Language functions in hardware. They also proposed an hardware extension integrated in the future microprocessors to accelerate the dynamic memory management [16].

In [25,26], the authors propose a memory allocator/deallocator (namely, SoCDMMU) suitable for real-time systems. The main characteristic is that by using the SoCDMMU, the memory management is fast and deterministic. The main limitation of the SoCDMMU architecture is that all memory accesses need to pass through the SoCDMMU, becoming a bottleneck for multiprocessors.

De La Luz et al. [19] explore a multi-bank memory system and describes an automatic data migration strategy which dynamically places the arrays with temporal affinity into the same set of banks.

Our work differs, since we choose to exploit the NoC to enable efficient dynamic memory management. We use a message passing mechanism and a hardware MMU, while the authors of [19] proposes a non-trivial software algorithm executed by the processor.

3. Target on-chip multiprocessor architecture

In this section, we present the architecture of the target multiprocessor system-on-chip, focusing on the central points of our analysis: the interconnection network, the organization of the memory subsystem and the memory management unit.

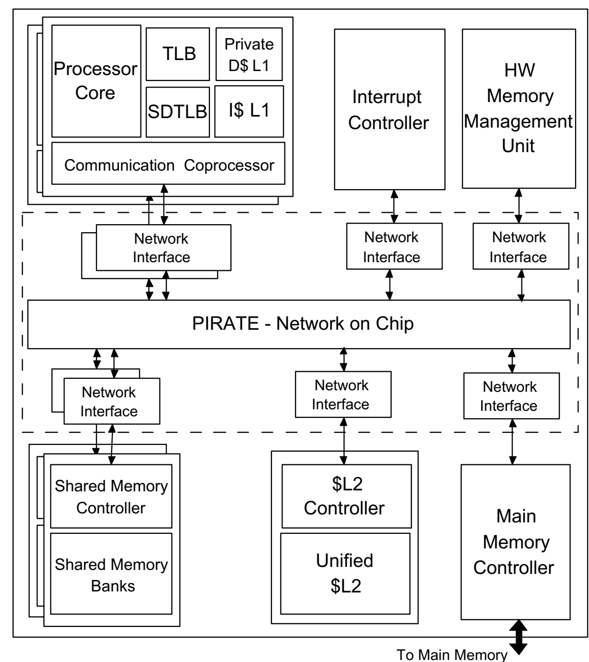


Fig. 1. Target on-chip multiprocessor architecture.

Fig. 1 shows the architecture of the on-chip multiprocessor composed of several processing elements (PEs), the memory modules and the HwMMU. The core of the system is the scalable communication-centric interconnect fabrics (namely, PIRATE NoC [9]). The PIRATE NoC is connected to each module through the corresponding network interface (NI), that implements the translation of communication protocols and the data packeting.

The memory space is composed of a shared memory space and separate private memory spaces for each processing element. Private data are cached to improve performance without requiring the need of cache coherency support.

Each processing element (PE) is composed of the core processor, the L1 private data cache, the L1 instruction cache, the shared data translation look-aside buffer (SD-TLB), the TLB for private data and instructions, and the communication coprocessor interfacing the NI. Two different TLBs have been introduced to support different virtual-to-physical addresses translation schemes (such as different page sizes). The L2 cache (unified for private data of each processor and instructions) is centralized on a dedicated module interfacing the network. The network also interfaces the off-chip main memory through the on-chip main memory controller.

The memory modules for shared data are distributed across the NOC topology providing non-uniform memory accesses. For each shared memory module, the corresponding memory controller interfaces the network. The proposed NoC-based architecture is scalable in terms of number of processing elements and distributed shared memory modules. The on-chip centralized MMU module supports memory allocation/deallocation dynamically. Finally, global asynchronous events are managed by the interrupt controller module.

3.1. Interconnection

The interconnect is the key element of the on-chip multiprocessor system, since it provides a low latency communication layer, capable of minimizing the overhead due to thread spawning and synchronization. This approach, when compared to bus-based solutions, solves physical limitations due to wire latency providing higher bandwidth and parallelism in the communication.

PIRATE [9] is a Network-on-Chip composed of a set of parameterized interconnection elements (routers) connected by using different topologies (such as ring, mesh, torus, etc.). The router internal architecture is based on a crossbar topology, where the $N \times N$ network connects the N input FIFO queues with N output FIFO queues. The number of each input (output) FIFO queue is configurable as well as queue length (at least one). The incoming packets are stored in the corresponding input queue, then, the packets are forwarded to the destination output queue. The I/O queues and the crossbar are controlled by the router controller, that includes the logic to implement routing, arbitration and virtual-channels.

The PIRATE router is 3-stage pipelined architecture that implements table-based routing and wormhole switching. The PIRATE network interface implements the adaptation between the asynchronous protocol of the network and the specific protocol adopted by each IP-node composing the system. The network interface is also responsible for the packeting of the service requests coming from the IP nodes. The service level is best effort and there is a round robin priority-based arbitration mechanism.

3.2. Memory subsystem

The memory space is composed of separate private memory spaces and shared memory space. A private memory space exists for each PE and it cannot be seen by any other PE in the system except for the owner. Shared portions of the addressing space are mainly used for synchronization and data exchange. While private memories map to L1/L2 caches and off-chip main memory, shared memory is only implemented by on-chip RAM modules distributed on the NoC (without caching).

Private data are cached to improve performance without requiring the need of cache coherency support. Two levels of caching are provided: the L1-caches for private data and instructions are placed on each PE, while one L2 unified (private data and instructions) cache is centralized on a dedicated module interfacing the network.

The memory modules for shared data are distributed on the NoC topology providing non-uniform memory accesses (*NUMA approach*). Shared data are not cached to avoid the overhead of cache coherency support. Globally, data locality is exploited for private data by L1 and L2 (where no coherency support is required), while shared data access locality is assured by distributing the shared memory modules on-chip. Shared memories are directly exposed to the programmer, which can explicitly place shared data or synchronization variables.

4. Memory management unit

In the proposed target architecture, shared data are dynamically managed and shared data allocation/deallocation mechanisms are provided by the on-chip centralized HwMMU module. A virtual memory mechanism decouples the logical addressing space, as seen by each processors, from its physical addressing implementation.

The memory management features have been implemented in a hardware module to reduce the overhead due to the operating system in terms of energy and performance [26]. The HwMMU makes the memory management fast and deterministic (if we do not consider the network latency). The dynamic management makes memory utilization more efficient especially for embedded applications whose memory requirements can change significantly during run-time.

The HwMMU holds the table of the allocated shared pages (*Shared Page Table*) and the virtual to physical addresses correspondence for each shared page. The SD-TLB in each PE is used to cache this information and in our target architecture

it has a fully associative structure. When a SD-TLB miss occurs, physical page information are fetched from the HwMMU and the corresponding SD-TLB entry is updated.

Additionally the HwMMU holds a table called the *Memory State Table* (*MST*) where the state values (*active* or *sleep*) of the memory modules are stored. When different HwMMU operations are performed, the memory modules can be power-managed to reduce power consumption. The table also stores the number of pages allocated in each memory. Allocation and deallocation are performed at the granularity of the page size. This design choice reduces the memory fragmentation. In our current implementation, the HwMMU is the only unit in the system that can change the memory power states. This means that the shared memory *sleep* and *active* signals are not exposed to the software.

4.1. HwMMU primitives

In the following, the different primitives supported by the HwMMU are described. There are two types of primitives that the on-chip HwMMU can execute: *Allocation/Deallocation* and *Move/Copy*. These primitives are described hereafter, while Table 1 compares, for each primitive, the execution time required by the HwMMU and by the corresponding software routine to completely execute the primitive from the user application. The execution times have been calculated (for the target architecture described in Section 5) by assuming that one-hop network distance between each module and that each memory involved in the operation is already in a *active* state. For the computation of execution times related to the software case, data structures used for the memory management have been assumed as already stored locally on private data caches. This assumption represents a lower bound for the execution time of software routines since, typically, data structures to support memory management in shared-memory multiprocessors are stored in shared memory. The results reported in Table 1 show how the dynamic memory management provided by the HwMMU requires at least half execution time with respect to the software routine.

The centralized HwMMU does not represent a bottleneck for memory accesses, because, in the target architecture, SD-TLBs are used on each PE for locally caching address translations. The HwMMU is used for allocation/-deallocation/-copy/-move for shared data and in case of SD-TLB misses to access the shared page table.

4.1.1. Allocation/deallocation

The basic allocation and deallocation primitives for the shared data have the following structures:

MMU_MALLOC(MEM, size)
MMU_FREE(address)

where *size* is the number of data bytes to allocate, *MEM* is the number of the memory module where data must be allocated, and *address* is the pointer of the space to release. By setting to -1 the value of *MEM*, the memory allocation is assigned to HwMMU. In all cases (when MEM is equal to -1 or not) the allocation request is sent to the HwMMU.

The number of the shared memory modules (*MEM*) represents an unique logic identifier for each shared memory module. This value does not represent a topological information regarding the position of the module on the network. For this reason while using an explicit *MEM* value we must assume to know at software level the topology of the system in order to map our allocation on a particular memory module.

Table 1
Comparison of execution time required by MMU primitives implemented by on-chip HwMMU or SW routine

Command	On-chip HwMMU	SW routine
	Execution time (clock cycles)	Execution time (clock cycles)
MMU_MALLOC(*MEM*, *size*)	61	$\geqslant 220$
MMU_FREE(*address*)	58	$\geqslant 96$
MMU_COPY(*MEM*, *address*, *number*)	$66 + 54886 \times \text{page}$	$\geqslant 220 + 94208 \times \text{page}$
MMU_MOVE(*MEM*, *address*, *number*)	$69 + 54886 \times \text{page}$	$\geqslant 316 + 94208 \times \text{page}$

Assuming a 5 ns clock period for the PEs. Variable *page* is the number of 4 KB-pages to be moved/copied.

The allocation of a page on a memory module is performed after checking the state of the memory on the *MST*. If the memory module is already active, it is used. Otherwise an *active* signal is sent to the memory.

If the memory is not available for accepting the request an invalid value is returned. It is thus responsibility of the software layer to manage the exception.

When the allocation operation is completely assigned to the HwMMU, several strategies can be adopted to choose which shared memory module can be used. Round-robin strategy is the default strategy, but power-aware strategies are also possible.

4.1.2. Move/copy

The HwMMU can be used not only to allocate/deallocate space, but also to *move* and *copy* shared data. These two operations are similar but, while the *move* operation needs to free the original space of the data to be moved, the *copy* operation does not. The basic *copy* and *move* primitives have the following structures:

MMU_COPY(MEM, address, number)
MMU_MOVE(MEM, address, number)

where *address* is the page virtual address, *number* is the number of pages to be copied/moved (starting from *address*), while *MEM* is the number of the memory module where the pages must be placed. When *MEM* is equal to -1, the decision on the destination memory module is left to the HwMMU.

The *MMU_MOVE* primitive starts allocating space in the destination shared memory module indicated by *MEM*. Once performed the allocation, the HwMMU broadcasts an invalidate request to SD-TLB locations referring to the pages to be moved in order to force each memory request to those pages to pass through the HwMMU. Then, the copy of the pages can be performed and, if a request occurs during the copy, the HwMMU delays the response until the move operation ends. The source pages can be deallocated and the new page address is updated in the HwMMU table. After the *MMU_MOVE* operation a *MMU_FREE* operation, as the one explained in the Section 4.1.1, is performed.

The *MMU_COPY* operation starts allocating space in the destination memory module indicated by *MEM*. The copy of the pages can then be done and, at the end of the operation, the new page address is sent back to the PE that required the copy. Performing a move primitive, the virtual address of the page is the same before and after the operation, while the copy primitive returns the new page address.

The *MMU_COPY* and *MMU_MOVE* primitives can be useful to exploit memory locality (by data migration and duplication) and space compaction.

4.2. Exploitation of the HwMMU primitives

In this section, we discuss how *MMU_COPY* and *MMU_MOVE* primitives can be used to exploit the memory locality (by data migration and duplication) and space compaction. The use of these features of HwMMU primitives combined with the use of the power states for the shared memory banks can result in the reduction of both execution time and energy consumption.

The simplest way to take advantage of the *MMU_COPY* and *MMU_MOVE* primitives is to use them as dedicated DMA functions to manage memory transfers in the shared memory space. This feature can result useful for instance when designing pipelined applications on multi-buffered data structures. While a buffer is transferred from a shared memory to another, following the processing pipeline, another buffer is computed by the processing elements. In such way, since transfers are assigned to the HwMMU, the latency of the transfers can be completely overlapped with useful computation.

Other more sophisticated uses of the HwMMU primitives are supported in our proposed architecture. In the rest of this section, we explain the mechanisms to support the *migration*, *replication* and *compaction* of shared data.

The *migration* strategy can be used when shared data are located in shared memory modules *far* (in terms of network hops) from a PE that is currently using them for most of the time. In this case, shared data could travel in all the network before reaching the interested PE. This results in a high latency of the memory accesses (which causes the PE to stall) as well as a high network traffic. Our solution suggests to use the HwMMU to move the data in a shared memory *near* to the PE which is using them for most of the time.

We consider a system where the topology of the architecture is known at application-level by all the PEs. Our technique reminds the explicit data

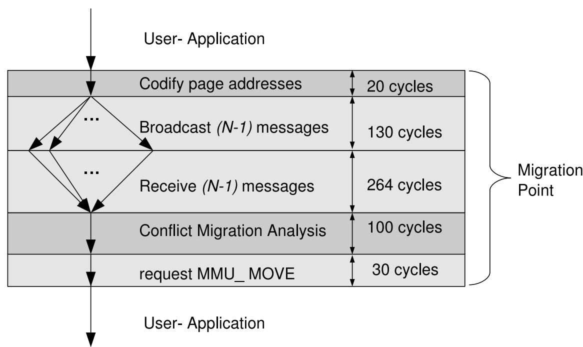


Fig. 2. Different steps of the user-level *migration point* algorithm. Clock cycles are calculated for a configuration of 8 PEs connected with a mesh network where the lists of virtual page addresses are composed of 4 addresses encoded in messages of 4 Bytes.

migration approach used in distributed shared memory multiprocessors [23]. The proposed idea consists of inserting in the application some explicit migration points, exploiting the above described HwMMU primitives. Fig. 2 shows the structure of the migration point algorithm as seen from each processing element of the system.

In each migration point, each PE performs a global communication exchange with the other PEs in the system through a broadcast message. During this all-to-all exchange each PE sends to the other PEs a list of the most recently used virtual page addresses. At at the end of this phase, each PE knows which are the virtual page addresses that each other PE is using. For all the virtual pages addresses that do not generate a migration conflict (i.e. are not requested by an other PE) each PE sends a *MMU_MOVE* primitive to move the pages in a shared memory module which is *near* to its location on the network. If a migration conflict is detected on a virtual page address, i.e. a PE detects that a virtual page address is also required by another PE, a different strategy can be employed as for instance the *replication* of the page using the *MMU_COPY* primitive. In this case, problems of shared pages coherence must be explicitly managed at software level by the user application. The explicit migration points can be added manually by the user or automatically by the compiler, and they can be placed after global synchronization points or at the beginning of high parallel data-intensive portions of code.

In a system with pages of 4 KB and a total shared memory size of few MBytes, the number of shared memory pages is not dramatically large. In this case, considering a list of 2–4 pages, we can encode the broadcast message in a single word. Putting the migration points after global synchronization or before data-intensive segments of code, we ensure low utilization of the network interconnect that can deliver the broadcast messages in few clock cycles. In Fig. 2, we report the number of clock cycles spent in every phase of the algorithm for a configuration of 8 PEs connected with a mesh network topology. The lists of virtual page addresses are composed of 4 addresses encoded in messages of 4 Bytes.

It is worth noting that at the end of the migration points, the computation can restart without explicit wait for the completion of the move operations performed by the HwMMU. The mechanism of invalidation of the SD-TLB implemented in the *MMU_MOVE* allows the PEs to continue in the computation, except when a SD-TLB miss occurs. If either the programmer or the compiler can overlap the migration points with useful computation (for instance when working on local data) the migration cost can be reduced significantly.

Another technique that can take advance of the HwMMU is the *compaction* of shared data. When multiple shared memory modules are partially used, a *MMU_MOVE* primitive can be issued in order to compact data on a single module. In fact, for an energy efficient use, all the memories involved in the data compaction which do not contain any more pages can be put in *sleep* state.

5. Experimental results

In this section, first, we give an overview of the experimental setup used to obtain the results (Section 5.1), second we discuss the exploration results of the target MPSoC by varying the memory architecture (in Section 5.2). Finally, in Section 5.3, we present a case study for a graph exploration application, also showing the exploration results of NoC topologies.

5.1. Experimental setup

The simulations have been carried out at the cycle-accurate level. Power model parameters have been estimated by gate-level simulation (with 130 nm STMicroelectronics CMOS technology) and extracted by experimental data (from ARM core [11] and from CACTI for memories [32]). Accuracy of power models can be directly derived

Table 2
Parameters of the target architecture

Parameter	Value
Number of PEs	8
PE clock frequency	200 MHz
NoC clock frequency	400 MHz
NoC data-path size	1 Byte
NoC topology	Mesh
Memory interface and control logic clock frequency	200 MHz
Shared memory space	512 KB
SD-TLB size	8 entry
SD-TLB replacement policy	LRU
Shared memory page size	4 KB

from the accuracy of the models used for each module ([9,11,32]).

The parameters of the target MP-SoC architecture chosen for the experiments are reported in Table 2. The system is composed of 8 PEs and each PE is an ARM7 core with a clock frequency of 200 MHz. The 200 MHz clock frequency has also been used for shared memories interfaces and control logic. The *PIRATE-NoC* has a 400 MHz clock to minimize the latency, while not dramatically impacting the power consumption. The network data-path size is equal to one byte to reduce the hardware complexity of the on-chip network, while the NoC topology has been configured as a mesh. The mesh topology includes $(12 + n)$ nodes, where n is the number of distributed shared memory modules (varying from 1 to 5 in the experiments reported in Section 5.2). Three different NoC topologies have been explored in Section 5.3.2.

The mapping of the IPs and memory modules to the nodes of the network topology has been optimized for each benchmark in order to minimize the average network latency by using an *in house* developed tool [31].

All the reported results have been obtained by using the dynamic features of the on-chip HwMMU by setting to -1 the MEM field in the MMU primitives. The algorithm used by HwMMU for data allocation is round-robin.

A set of kernels have been extracted from the SPLASH-2 suite [29] of parallel benchmark applications to be used as application benchmarks in Section 5.2. The SPLASH-2 suite has been released to facilitate the study of centralized and distributed shared address space for multiprocessing. Two other applications have been written by using PARMACS macros [30]: a matrix multiplication algorithm (*Matrix*) and a vector normalization (*Norm*). PARMACS macros offer basic primitives for synchronization, creation of parallel processes and shared memory allocation. Table 3 summarizes the selected set of benchmarks with a short description of the functionality and the origin of each benchmark (SPLASH-2 suite or PARMACS-based).

The main characteristics of the selected benchmarks have been reported in Fig. 3 for the target architecture with a single shared memory module. For each benchmark, the breakdown of the normalized execution time spent by each processor in computation, shared memory accesses and synchronization has been reported. Based on the results shown in Fig. 3, the selected benchmarks can be grouped in computation-intensive (*FFT* and *Norm*), memory-intensive (*Matrix*) and synchronization-intensive (*LU-1*, *LU-2* and *Radix*).

5.2. Exploration of distributed shared memory architectures

In this section, we show the results of the exploration of the distributed shared memory subsystem.

Fig. 4 shows the shared memory request latency for the *Matrix* application by varying the number of distributed shared memory modules from 1 to

Table 3
The selected set of benchmarks

	Description	From
`FFT`	Complex 1 D radix-$\sqrt{n}$ 6-step FFT	SPLASH-2
`LU-1`	Lower/upper triangular matrix factorization	SPLASH-2
`LU-2`	*idem*, but optimized for spatial locality (*contiguous blocks*)	SPLASH-2
`Radix`	Integer radix sort	SPLASH-2
`Matrix`	Matrix multiplication	PARMACS-based
`Norm`	Parallel vector normalization	PARMACS-based

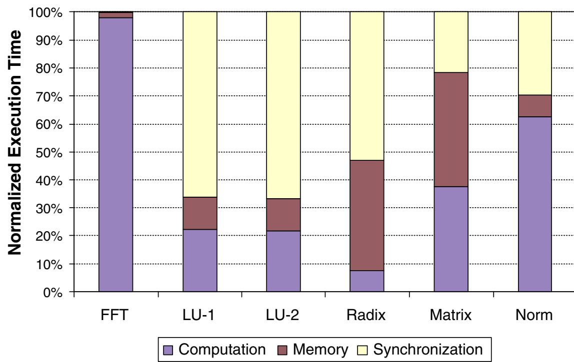


Fig. 3. Breakdown of normalized execution time spent in computation, shared memory accesses and synchronization for the selected set of benchmarks.

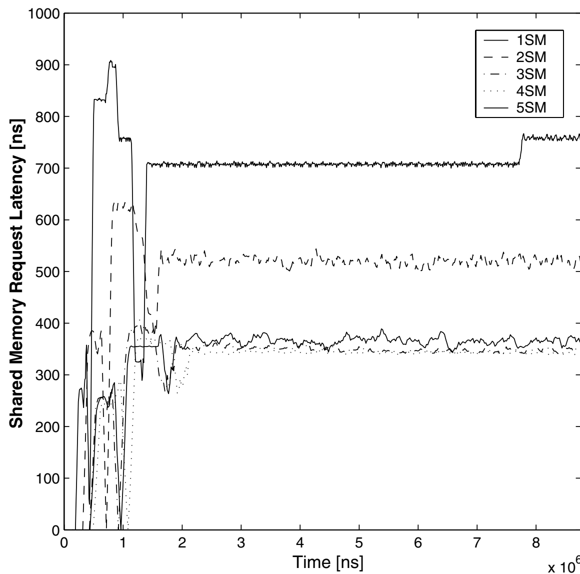


Fig. 4. Shared memory request latency for Matrix benchmark by varying the number of distributed shared memory modules from 1 to 5.

5. Each point has been computed by considering the average value of shared memory request latency in 10 µs time interval. *Matrix* benchmark has been chosen since it is memory-intensive, so the effects of memory exploration are more evident. The shared memory request latency represents the latency necessary to receive the response of a memory request. It includes the network request latency (from source to memory module), the memory latency, the network response latency (from memory back to the source of request) and the possible latencies due to contentions.

Except for the initial behavior (in the time slot between 0 ns and 1.5×10^6 ns), where the high values of request latency are mainly due to cold start SD-TLB misses, the shared memory request latency is almost stationary and its average value mainly depends on the number of physical memories. In Fig. 4, it can be observed that using a single physical memory the request latency, on average, reaches the higher value (712 ns approximately). Increasing the number of physical memories, up to 4, we can observe a reduction in the average request latency (approximately 512 ns, 360 ns, 348 ns, respectively, for 2, 3 and 4 physical memory). The reduction is due to two main factors: first, the reduction in memory contentions, since with a higher number of physical memories is less probable to have two accesses at the same time to the same memory, second, the reduction of the network latency, since with a higher number of physical memory distributed in the network is more probable to find the data at a reduced network distance with respect to a single shared memory. For 5 distributed shared memories, we can see how the average latency (approximately 376 ns) slightly increases with respect to the configurations with 3 and 4 physical memories. This is due to the fact that increasing the number of memories, the network size increases consequently as well as the diameter of the network. In the case of 5 memory modules, the advantages in memory contention and data locality are not enough to balance the disadvantage due to a greater network size.

Fig. 5a and b show performance and energy results obtained by exploring the number of distributed shared memory modules from 1 to 3 across the network. The analysis for 4 and 5 distributed shared memories are not shown because a negligible performance speedup has been obtained for the selected set of benchmarks (as confirmed by Fig. 4 for the *Matrix* benchmark).

For the selected benchmarks, Fig. 5a reports the execution times for 2 and 3 distributed shared memory modules, where the reported values have been normalized with respect to a single shared memory. For all the benchmarks, the distribution of shared data implies shorter execution time. This is more evident for *LU-1*, *LU-2*, *Radix* and *Matrix* (for which the gain is up to 53%), even if some small improvements can be observed also for the remaining benchmarks (up to 3%).

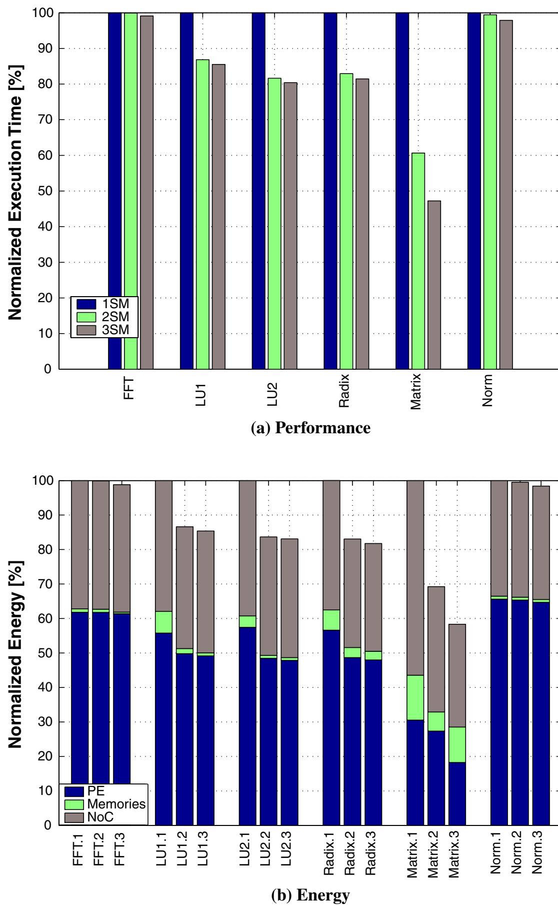


Fig. 5. (a) Performance, and (b) energy breakdown for the selected set of benchmarks. In (b), each benchmark has been labelled with ⟨*bench*⟩·*x*, where *x* is the number of distributed shared memory modules.

Distributing shared data in physical memories is more effective for *LU-1*, *LU-2*, *Radix* and *Matrix*, since these benchmarks are communication-based (as seen in Fig. 3) due to large synchronization overhead (for *LU-1*, *LU-2* and *Radix*) or due to many accesses to shared data (for *Matrix*). In particular, *Matrix* obtains the highest performance advantage with respect to *LU-1*, *LU-2* and *Radix* because it manages a large amount of shared data. In fact, the other three benchmarks show a high communication rate, but limited to synchronization structures, that have small size. This behavior is more evident by increasing the number of physical memories. In fact, passing from 2 to 3 physical memories, *Matrix* continues to reduce the execution time (approximately 14%), *LU-1*, *LU-2* and *Radix* gains only approximately 2%. For *FFT* and *Norm* benchmarks, the advantage in distributing shared data is not evident due to the computation-based nature of these applications. The time spent in communication is negligible and so the advantage of a reduced shared memory request latency does not affect significantly the system performance.

Regarding energy, Fig. 5b shows energy results for systems including 2 and 3 shared memory modules normalized with respect to the system with a single shared memory. The corresponding energy breakdown between PEs, shared memory subsystem and NoC is also shown. Energy contributions due to HwMMU, L2 cache, interrupt controller and main memory controller are not shown since their values are constant by varying the number of shared memories. Energy saving in Fig. 5b is mainly due to two aspects: the first one is due to execution time reduction; the second one is related to the reduced energy cost of memory accesses for configurations of the shared memory distributed on 2 and 3 memory modules. The NoC energy is reduced, despite its complexity increases. In fact, more routers are needed to serve more memories, but the locality of the communication (in the system with distributed memories) trade-offs the energy consumption due to the additional routers. *FFT* and *Norm* do not show significant energy saving (up to 3%), while *LU-1*, *LU-2*, *Radix* and *Matrix* show up to 42% energy saving. The motivation of the small energy savings for *FFT* and *Norm* is the same shown for performance and proved from energy breakdown: these two benchmarks are characterized by high computation and a few memory accesses.

5.3. Case study: a graph exploration algorithm

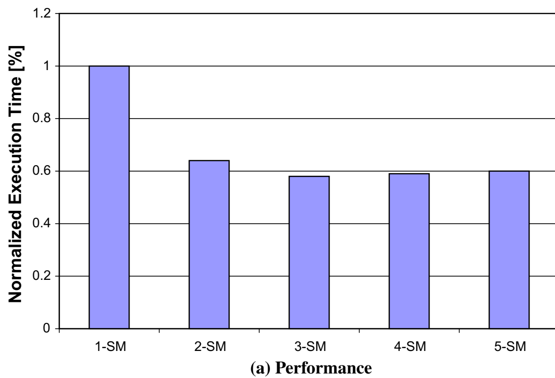
Many important applications require to navigate data structures represented by graphs with semantic information. Search or exploration algorithms play an important role while looking for nodes or paths with a certain property. Breadth-First Search (BFS) [8] is of particular importance among different graph search methods and it is widely used in several applications [2,3,6,7]. A common query that arises in analyzing a semantic graph, for example, is to determine the nature of the relationship between two vertexes in the graph, and such a query can be answered by finding the shortest path between those vertexes using BFS. Further, BFS can be used to find a set of paths between two vertexes whose lengths are within a certain range.

Algorithm 1. Perform a BFS exploration, starting from vertex r

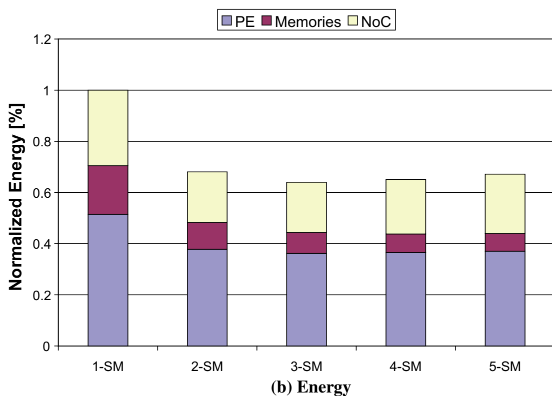
unmark all vertexes
mark the root vertex r
$L \leftarrow 0$
$Q \leftarrow r$
$Qnext \leftarrow \{\}$
repeat
  while $Q \neq \{\}$ **do**
    pop a vertex v from Q
    for all unmarked neighboring vertexes n of v
      mark n
      add n to $Qnext$
    end for
  end while
  $Q \leftarrow Qnext$
  $L \leftarrow L + 1$
until $Q = \{\}$

The sequential BFS algorithm (Algorithm 1) visits all the vertexes reachable from a selected root vertex by using the following strategy: the root vertex, from which the exploration starts, is added to a queue Q. All the vertexes on Q are sequentially examined and the neighbors of each vertex that are not already touched by the algorithm are appended to another queue called Qnext. Vertexes appended to Qnext are marked in a structure in order to skip them while the algorithm proceeds. Note that at the first level, only the root vertex is present in Q. When Q is empty, the level in the exploration is increased by one, Q and Qnext are swapped and the algorithm is executed again. The entire algorithm terminates either when Qnext is empty or when a vertex of interest is found.

In this paper, we evaluate a parallel implementation of the same algorithm where the queues Q and Qnext, as well as the marked structure, are shared among the PEs. A ticket-based mechanism is implemented on the Q using a fetch and increment operation in order to get a chunk of vertexes to explore. Parallelism is obtained by exploring in parallel different neighbor lists on the same level of BFS explo-

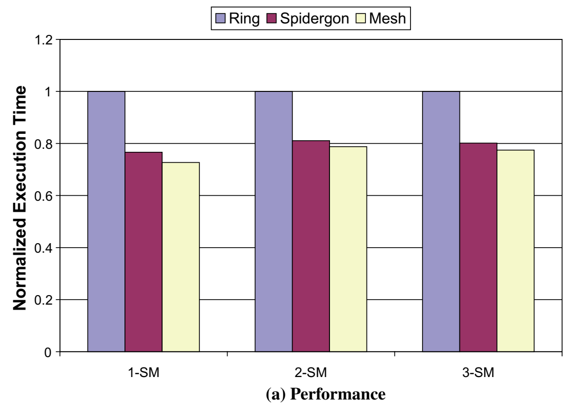


(a) Performance

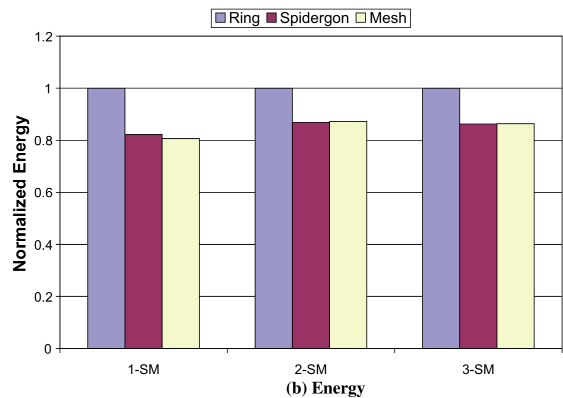


(b) Energy

Fig. 6. Performance (a) and energy (b) breakdown for our graph exploration algorithm.



(a) Performance



(b) Energy

Fig. 7. Performance (a) and energy (b) for our graph exploration algorithm by varying NoC topologies and distributed shared memory modules from 1 to 3.

ration. The BFS algorithm seems to us especially relevant, since it has poor locality properties. This is due to the fact that there is small data reuse while the graph is being explored. For this reason, caching of shared data would be less effective, motivating our approach based on distributed on-chip shared memories.

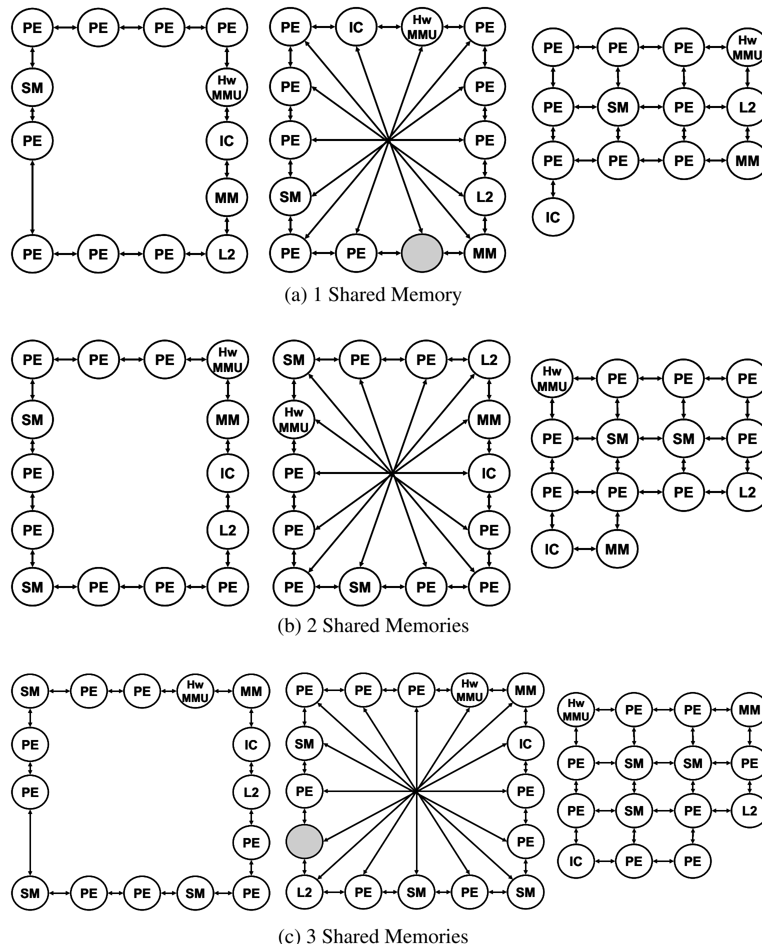
5.3.1. System energy and performance evaluation

Similarly to Fig. 5, the performance and energy for the BFS are shown in Fig. 6. Increasing the number of shared memories up to 5, both delay and energy can be reduced by 40% approximately. A minimum can be found for the 3-SM configuration, while, for a larger number of memories, both metrics slightly get worse. This is due to the fact that the overhead due to fetching data from a far memory becomes noticeable.

5.3.2. Exploration of NoC topologies

Since the NoC is one of the the key concepts in the design of the target system, in this case study, a more detailed analysis of the NoC is discussed. For our architecture – where the latency of any memory access is not uniform – it is especially important to consider the mapping of the cores into the network tiles, and the topology, because this influences the network diameter and the average connectivity.

Three different NoC topologies have been considered: ring, spidergon [18] and mesh. The number of distributed memory modules has been varied from 1 to 3 (corresponding to 13- to 15-node topologies). This because the 3-SM architecture results the optimal one. In Fig. 7, for each memory configuration, the reported values have been normalized with respect to the ring-based multiprocessor architecture.



(a) 1 Shared Memory

(b) 2 Shared Memories

(c) 3 Shared Memories

Fig. 8. Mapping of the cores to the explored network topologies (ring, spidergon and mesh) by varying the shared memories from 1 to 3. The nodes represent PE (processing element), SM (shared memory), MM (main memory), L2 (L2 Cache), IC (interrupt controller), and HwMMU (hardware memory management unit). (a) 1 Shared memory; (b) 2 shared memories and (c) 3 shared memories.

The exploration results reported in Fig. 7a show how, due to the reduced average number of hops, mesh offers the shortest execution time, followed by spidergon and ring. This behavior is evident for all memory configurations.

A similar trend has been reported in Fig. 7b, where the energy saved by the reduced execution time is more than the energy spent for a more complex network (passing from 3-port router required by ring, through 4-port router for spidergon up to 5-port router for mesh).

Finally, to better understand the previous results, Fig. 8 shows, for each one of the explored topology, the mapping of the cores into the network for 1–3 shared memories. The gray circles in the spidergon topology represent routers not connected to any physical IP, due to the fact that the spidergon topology includes an even number of nodes, as explained in [18].

6. Conclusions

This paper focuses on the power/performance exploration of the distributed shared memory approach in MP-SoC architectures based on the NoC. A distributed shared memory architecture has been proposed, that is suitable for low-power on-chip multiprocessors and supported by an on-chip hardware MMU. The exploitation of the HwMMU primitives has been discussed in the context of migration, replication, and compaction of shared data. The scalability of the proposed architecture with respect to the number of distributed memory modules has been evaluated in terms of power and performance. Furthermore, a BFS graph exploration algorithm has been used as case study to shown how, the system-level latency and energy reflect the influence of network topologies and the corresponding core mappings.

References

[1] J.M. Chang, E.F. Gehringer, A high-performance memory allocator for object-oriented systems, IEEE Transactions on Computers 45 (3) (1996) 106–111.

[2] A. Clauset, M.E.J. Newman, C. Moore, Finding community structure in very large networks, Physical Review E 6 (70) (2004) 066111.

[3] J. Duch, A. Arenas, Community detection in complex networks using extremal optimization, Physical Review E 72 (2005) 027104.

[4] T. Li, W.H. Wolf, Hardware/software co-synthesis with memory hierarchies, IEEE Transactions on Computers 18 (10) (1999) 1405–1417.

[5] V. Milutinovic, P. Stenstrom, Special issue on distributed shared memory systems, Proceedings of the IEEE 87 (3) (1999) 399–404.

[6] M.E.J. Newman, Detecting community structure in networks, European Physical Journal B 38 (May) (2004) 321–330.

[7] M.E.J. Newman, Fast algorithm for detecting community structure in networks, Physical Review E 69 (2004) 066133.

[8] M.E.J. Newman, M. Girvan, Finding and evaluating community structure in networks, Physical Review E 69 (2004) 026113.

[9] G. Palermo, C. Silvano, PIRATE: a framework for power/performance exploration of network-on-chip architectures, Lecture Notes in Computer Science, vol. 3254, Springer, 2004, pp. 521–531.

[10] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh, Performance evaluation and design trade-offs for network-on-chip interconnect architectures, IEEE Transactions on Computers 54 (8) (2005) 1025–1040.

[11] A. Sinha, N. Ickes, A.P. Chandrakasan, Instruction level and operating system profiling for energy exposed software, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 11 (6) (2003) 1044–1057.

[12] E. Von Puttkamer, A simple hardware buddy system memory allocator, IEEE Transactions on Computers C-24 (10) (1975) 953–957.

[13] S. Wuytack, J.L. da Silva, F. Catthoor, G. de Jong, C. Ykman-Couvreur, Memory management for embedded network applications, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18 (5) (1999) 533–544.

[14] F. Angiolini, L. Benini, A. Caprara, Polynomial-time algorithm for on-chip scratchpad memory partitioning, in: CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, California, USA, 2003, pp. 318–326.

[15] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, P. Marwedel, Scratchpad memory: a design alternative for cache on-chip memory in embedded systems, in: CODES '02: Proceedings of the 10th International Workshop on Hardware/Software Codesign, Estes Park, Colorado, USA, 2002, pp. 73–78.

[16] J.M. Chang, W. Srisa-an, C.D. Lo, Introduction to DMMX (dynamic memory management extension), in: Proceedings of ICCD Workshop on Hardware Support for Objects and Microarchitectures for Java, October, 1999, pp. 11–14.

[17] G. Chen, O. Ozturk, M. Kandemir, M. Karakoy, Dynamic scratch-pad memory management for irregular array access patterns, in: DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, 2006, pp. 931–936.

[18] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, A. Scandurra, Spidergon: a novel on-chip communication network, in: SOC 2004: Proceedings of International Symposium on System-on-Chip, Tampere, Finland, November 2004, p. 15.

[19] V. De La Luz, M. Kandemir, I. Kolcu, Automatic data migration for reducing energy consumption in multi-bank memory systems, in: DAC '02: Proceedings of the 39th Conference on Design Automation, New Orleans, Louisiana, USA, 2002, pp. 213–218.

[20] M. Kandemir, J. Ramanujam, A. Choudhary, Exploiting shared scratch pad memory space in embedded multiprocessor systems, in: DAC '02: Proceedings of the 39th Conference on Design Automation, New Orleans, Louisiana, USA, 2002, pp. 219–224.

[21] A.M. Molnos, M.J.M. Heijligers, S.D. Cotofana, J.T.J. Van Eijndhoven, Compositional memory systems for multimedia communicating tasks, in: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe, 2005, pp. 932–937.

[22] M. Monchiero, G. Palermo, C. Silvano, O. Villa, Exploration of distributed shared memory architectures for NoC-based multiprocessors, in: IEEE IC-SAMOS'06: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, July 2006, pp. 144–151.

[23] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, E. Ayguad, User-level dynamic page migration for multiprogrammed shared-memory multiprocessors, in: ICPP '00: Proceedings of the 2000 International Conference on Parallel Processing, 2000, p. 95.

[24] O. Ozturk, M. Kandemir, S.W. Son, M. Karakoy, Selective code/data migration for reducing communication energy in embedded MPSoC architectures, in: GLSVLSI '06: Proceedings of the 16th ACM Great Lakes Symposium on VLSI, Philadelphia, PA, USA, 2006, pp. 386–391.

[25] M. Shalan, V.J. Mooney, A dynamic memory management unit for embedded real-time system-on-a-chip, in: CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, San Jose, California, USA, 2000, pp. 180–186.

[26] M. Shalan, V.J. Mooney, Hardware support for real-time embedded multiprocessor system-on-a-chip memory management, in: CODES'02: Proceedings of the 10th International Workshop on Hardware/Software Codesign, Estes Park, Colorado, 2002, pp. 79–84.

[27] W. Srisa-an, C.D. Lo, J.M. Chang, A hardware implementation of realloc function, in: WVLSI '99: Proceedings of the IEEE Computer Society Workshop on VLSI'99, 1999, p. 106.

[28] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: a survey and critical review, in: Proceedings of International Workshop on Memory Management, Kinross Scotland (UK), September 1995, pp. 1–78.

[29] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the 22th International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, pp. 24–36.

[30] E. Artiaga, N. Navarro, X. Martorell, Y. Becerra, Implementing PARMACS macros for shared-memory multiprocessor environments, in: Technical Report No. UPC-DAC-1997-07, 1997.

[31] V. Catalano, M. Monchiero, G. Palermo, C. Silvano, O. Villa, GRAPES: a cycle-based design and simulation framework for heterogeneous MPSoC, Technical Report No. 45, Politecnico di Milano, 2006.

[32] P. Shivakumar, N.P. Jouppi, CACTI 3.0: An integrated cache timing, power, and area model, Technical Report, HP Labs, 2001.

Matteo Monchiero received the Laurea degree (cum laude) in Electronic Engineering from Politecnico di Milano (Italy) in 2003. He is currently a PhD candidate at Politecnico di Milano, working on the power/performance analysis and optimization of multi-core architectures. His main research interests are in the area of computer architecture, with particular emphasis on multi-core architectures, branch prediction and low-power and thermal-aware microarchitectures. He is a member of Technical Committee on Computer Architecture (TCCA) and on Microprogramming and Microarchitecture (TCUARCH).

Gianluca Palermo received the Laurea degree in Electronic Engineering, in 2002, and the Ph.D. degree in Computer Science, in 2006, from Politecnico di Milano (Italy). He is currently Assistant Researcher at Department of Electronic Engineering and Computer Science of Politecnico di Milano. Previously he was a Consultant Engineer in the Low Power Design Group of AST – STMicroelectronics working on networks on-chip. His research interests include design methodologies and architectures for embedded systems, focusing on power estimation, on-chip multiprocessors and networks on-chip.

Cristina Silvano is an Associate Professor at the Department of Electronic Engineering and Computer Science of the Politecnico di Milano (Italy). She received her Laurea degree in Electronic Engineering from Politecnico di Milano in 1987. From 1987 to 1996, she held the position of Senior Design Engineer at R&D Labs of Bull HN Information Systems in Italy. She received her Ph.D. degree in Computer Engineering from University of Brescia (Italy) in 1999. From 2000 to 2002, she has been an Assistant Professor at the Department of Computer Science, University of Milano. Her primary research interests are in the area of computer architectures and computer-aided design of digital systems, with particular emphasis on low-power design and systems-on-chip.

Oreste Villa received the Laurea degree in Electronic Engineering from University of Cagliari (Italy) in 2003 and M.S. degree on Embedded Systems Design at the ALaRI Institute, Lugano (CH) in 2004. He is currently a PhD Student at Politecnico of Milan (Italy), majoring on design methodologies of multiprocessor architectures for embedded systems. He is also a PhD Intern in the Applied Computer Science Group at Pacific Northwest National Laboratory (PNNL) in Richland (WA-USA), where he is conducting a research on operating systems for clustered multiprocessor architectures.