Simultaneous Multithreading Support in Embedded Distributed Memory MPSoCs

Rafael Garibotti[1], Luciano Ost[1], Remi Busseuil[1], Mamady kourouma[1], Chris Adeniyi-Jones[2], Gilles Sassatelli[1], Michel Robert[1],

[1] LIRMM (CNRS-University of Montpellier II) – 161 rue Ada, Cedex 05 - 34095 Montpellier, France
{garibotti, ost, busseuil, kourouma, sassatelli, robert}@lirmm.fr

[2] ARM, Ltd. - Cambridge, Cambridgeshire, GB
Chris.Adeniyi-Jones@arm.com

ABSTRACT

Scalability and programmability are important issues in large homogeneous MPSoCs. Such architectures often rely on explicit message-passing among processors, each of which possessing a local private memory. This paper presents a low-overhead hardware/software distributed shared memory approach that makes such architectures multithreading-capable. The proposed solution is implemented into an open-source message-passing MPSoC through developing a POSIX-like thread API, which shows excellent scalability using application kernels used for benchmarking in shared-memory systems. This approach efficiently draws strengths from the on-chip distributed private memory that opens the way to exposing the multithreading programmability/capabilities of that component as a general-purpose accelerator.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles – advanced technologies, VLSI (very large scale integration).

General Terms

Design, Experimentation, Performance, Verification.

Keywords

Programmability, Multithreading, Distributed memory organization, NoC-based MPSoCs.

1. INTRODUCTION

MPSoC have become the de-facto platform template in many application domains ranging from consumer market products such as smart phones to hard real-time systems. In most cases, high performance allied to low power consumption [1] is required. Due to the increasing complexity of both platform architecture and application software, programming such multiprocessor systems is a challenge and demands for better programming model facilities [2]. Parallel programming models for embedded systems have long been explored and some used in high-performance computing (HPC) have been ported and adapted to embedded systems, such as the Message Passing Interface (MPI). The use of such APIs allows software designers to model explicit parallelism and synchronization between tasks/processor nodes at a reasonable level of abstraction while allowing better utilization of hardware resources [3].

DAC'13, May 29 - June 07, 2013, Austin, TX, USA.

The *objective* of this paper is exploring the opportunity of developing a POSIX-like threads API (Pthreads) [4], very popular in general-purpose and high-performance computing, onto a distributed private memory NoC-based MPSoC. This API commonly used for simultaneous multithreading (SMT) on symmetric multiprocessing (SMP) systems assumes a coherent shared memory architecture that each processor may access at any time. As the target architecture is of distributed private memory type, the proposed approach relies on additional hardware that makes it possible to expose a logically shared memory architecture based on physically distributed memories. Different from ccNUMA architectures (cache-coherent Non-Uniform Memory Machines) that use complex cache-coherence protocols for enforcing strict memory consistency, our approach targets embedded devices and therefore aims at minimizing additional hardware through a software-oriented approach. Since the resulting architecture retains its purely distributed nature, clusters operating in this distributed shared memory mode (referred to as vSMP for Virtual Symmetric Multiprocessing) are of arbitrary size and geometry, and can be decided at run-time for better suiting application needs.

The *contributions* of this paper may be summarized as follows: (*i*) providing multithreading capability onto a RTL distributed private memory NoC-based MPSoC, (*ii*) implementation of a hardware module that enables vSMP clusters definition at run-time, along with a software-based memory consistency approach. These contributions allow us to evaluate the resulting system, demonstrating its efficiency in terms of scalability of platform and application, as well as better throughput for several application/benchmarks scenarios.

This paper is organized as follow: Section 2 presents related work in MPSoCs programmability. Section 3 describes the basic concepts inherent to the adopted NoC-based MPSoC along with proposed API and hardware. Section 4 describes the experimental setup. Section 5 gives results in term of performance scalability and analyzes the overhead created. Finally, Section concludes and points out future work directions.

2. RELATED WORK

Several studies have been aimed at improving MPSoCs programmability by including/extending different APIs primitives, as well as proposing tools that can help designers to program parallel embedded applications. For instance, Ceng et al. [3] propose the MAPS framework that can be employed to guide application designers during the parallelization process. A similar framework that allows to explore different parallelization alternatives at high-level is described in [5]. Both approaches generate a parallel version of a sequential C code, which is validated on the top of a shared-memory system. In turn, Marongiu et al. [2] extended the OpenMP API, allowing allocating array data over multiple distributed scratchpad memories (SPMs) based on profile information to optimize data allocation. Two MPSoC models were implemented in a SystemC full-system simulator in order to validate the proposed OpenMP API (e.g. primitives, compiler). The proposed approach differs from Marongiu's in four main aspects: (*i*) processors are interconnected by a cross-bar, while a mesh NoC is employed in the present work, which makes our system more scalable while consuming less energy and area when compared to a cross-bar; (*ii*) no OS/library support for thread creation and management is available in Marangiu's approach, (*iii*) only benchmarks that fit entirely in cache memories were used in [2] since no cache coherence protocol is supported in his work, while our system implements cache coherence protocol, allowing to execute applications that deal with larger memory footprint (e.g. MJPEG) and (*iv*) the proposed Pthreads approach was evaluated in a real NoC-based MPSoC that has already been prototyped in different FPGAs and also ported to ASIC to evaluate the area overhead, while Marangiu's approach uses a simulator to validate his idea. Ophelders [6] proposes a software cache coherence protocol that was validated on a dual-core ARM9 architecture using the SPLASH2 benchmark. Ophelders work differs from ours in two aspects. First it considers a system with MMU, and a 4-set associative cache, allowing high-level memory management; while our approach relies on compact scalar MMU-less processor architecture with direct mapped caches, which impacts on less memory utilization. Second, our approach to memory coherence is software-based and relies on flushes and invalidations, not allowing remote execution that implies in less memory access.

To the best of our knowledge this work presents the first purely distributed memory NoC-based MPSoC that supports multithreading through POSIX thread API while dealing with cache/memory coherence at OS level. The proposed approach improves the system programmability; while better scalability and performance can be achieved once high visibility permits improving cache coherence and management.

3. MULTITHREADING PLATFORM

3.1 Platform Description

OpenScale[1] is a homogeneous message-passing NoC-based MPSoC with distributed private memories. Each node comprises a 32 bit pipelined CPU with configurable instruction and data caches. This core implements the Microblaze ISA architecture and features a timer, an interrupt controller, a RAM and optionally an UART [7], as illustrated in Figure 1.

1 Available for download at: www.lirmm.fr/ADAC

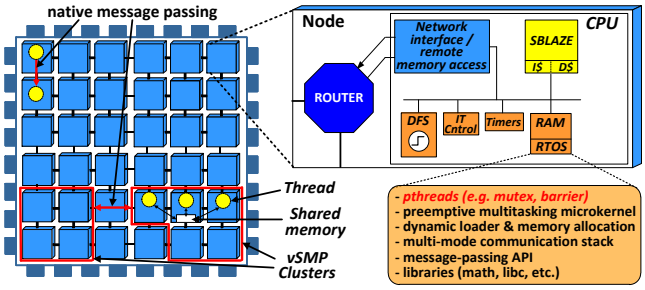


Figure 1 - Adopted NoC-based MPSoC architecture.

The platform RTOS is a pre-emptive priority-based micro-kernel. Each processing element (PE) runs its RTOS independently, and communications and synchronizations are made through a light implementation of the MPI message-passing library. Applications are represented through Khan Process Network, a standard representation in such a message-passing oriented system. Examples of services supported in the RTOS are: (*i*) run-time dynamic applications loading, (*ii*) preemptive round-robin scheduler based on thread credits, (*iii*) system-monitoring mechanisms (e.g. processor workload) and (*iv*) API with drivers support (e.g. UART). The platform supports a number of features such as task-migration and distributed decision-making capabilities [7].

3.2 Multithreading hardware support

In order to enable simultaneous multithreading (SMT) on any multiprocessor architecture, a logically shared memory space must be exposed to the software. Moreover, as the chosen architecture is organized into a distributed private memory, the hardware must also be modified to enable access to remote memories from any node participating to the vSMP cluster (Virtual Symmetric Multiprocessing). Figure 1 shows an example of vSMP clusters that may be defined at run-time using arbitrary size and geometry. They may communicate with each other using message passing, but be careful with the placement, because as shown in Figure 1, perhaps the PE between clusters may incur a communication performance penalty, depending if this PE is using the NoC to communicate with another PE or not.

The proposed module that serves this purpose, called remote memory access (RMA), is composed of an RMA-Send and an RMA-Reply, which together enable remote memory access through the NoC, more two asynchronous FIFOs are used for communication between both modules and the NoC. The RMA module contains a cache miss handling protocol that comprises three main steps: (*i*) whenever a cache miss occurs, the CPU issues a cache line request routed through the NoC, (*ii*) the host RMA reads the desired cache line (i.e. instruction/data), which is sent back to the remote CPU that resumes thread execution as soon as cache line is received (*iii*), as illustrated in Figure 2.

The remote memory access protocol has a latency of 182 clock cycles at zero NoC load, from the cache line request to the remote thread execution, which is depicted in details in Figure 3, remembering that this latency only occurs when trying to access the shared memory and not accessing the local memory during a cache miss. The protocol and its implementation have been optimized for latency and not throughput, as cache miss traffic is rather more latency-sensitive. Low to moderate latencies are ensured up to an end-to-end bandwidth of 90MB/s at 500MHz.

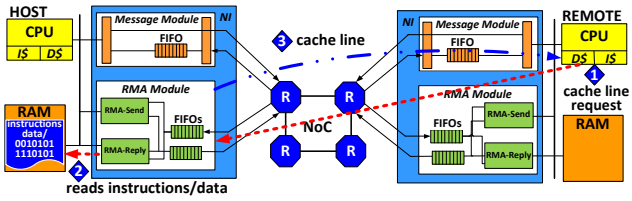


Figure 2 - Cache miss RMA protocol.

Note that the host RMA is kept busy during 64 cycles: since there is one input fifo and output fifo, a new request can be serviced before the cache line write to NoC (64 cycles), which corresponds to cache line request acquisition (15 cycles) and local memory read (14+35 cycles), as shown in Figure 3. This results in a maximum theoretical bandwidth of 250MB/s when several requests are aliased.

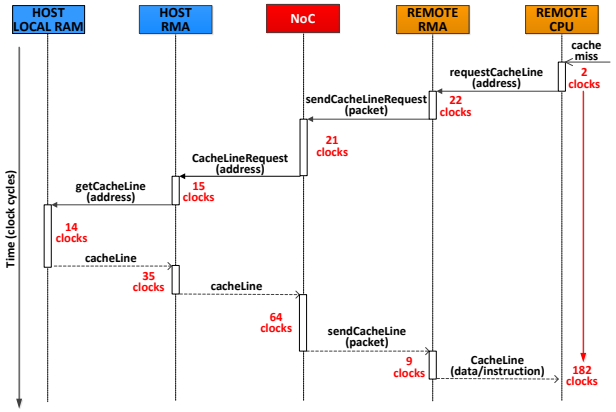


Figure 3 - Cache miss RMA protocol in a READ transaction.

3.3 Multithreading software support

The in-house RTOS described in 3.1 was modified to support a configurable processor memory mapping that permits specifying the ratio of local private data versus local shared data, made visible to all processors in the cluster as shown in the Figure 4.

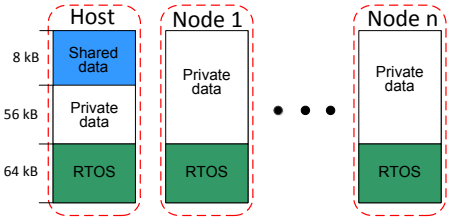


Figure 4 - DSM node memory organization.

Moreover, a subset of the widely used POSIX multithreading API has been ported into the microkernel, as shown in Table 1. The implementation of this API was focused on three categories: thread creation, mutexes and barriers for thread mutual exclusions and synchronizations. The memory consistency model relies only on the memory coherence guaranteed at each API function call: which corresponds to a relaxed memory consistency model, i.e. having a coherent memory whenever synchronization is made. The protocol operates as follows:

- During a thread creation, data caches are flushed on the caller side and invalidated on the executer side.
- During a mutex lock, cache lines that possibly contain shared data that can be accessed between the locking and unlocking are invalidated.
- During a mutex unlock, those same lines are flushed.
- During the barrier calls, the shared memory is also flushed and invalidated.

Table 1 - Implemented Pthread primitives.

Pthread primitives	Description
pthread_create()	Creates a new thread in the calling process
pthread_exit()	Terminates the calling thread
pthread_join()	Waits for the specified thread to terminate
pthread_mutex_init()	Initializes the specified mutex
pthread_mutex_destroy()	Destroys the specified mutex
pthread_mutex_lock()	Locks the mutex object
pthread_mutex_unlock()	Unlocks the mutex object
pthread_barrier_init()	Initialize a barrier object
pthread_barrier_wait()	Synchronizes participating threads at the barrier pointed to by the barrier argument

This is the smallest protocol that ensures cache coherency hardware and software support: the invalidation and flush operations of any cache line are performed only if the cache line tag corresponds to the address specified by the instruction. This condition avoids unnecessary cache flushes / invalidations of cache lines containing unrelated data.

4. EXPERIMENTAL SETUP

4.1 Application kernels

Three application kernels often employed in multithreaded architectures benchmarking were selected: Smith Waterman (DNA sequence-alignment matching application), MJPEG encoder and FFT (Fast Fourier Transform). A pthread implementation was developed for each, in which threads granularity is sound. MJPEG threads process entire images, whereas Smith-Waterman implementation relies on worker threads that run sequence alignment algorithm on different data sets. FFT is made of a number of threads, each of which performs a number of butterfly operations for minimizing data transfers. Table 2 gives benchmark-specific figures expressed in terms of single-thread processing time (using 16kB of cache), computation to data ratio and code size. Due to the different natures of those applications, execution time varies greatly and ranges from less than 2 million clock cycles (4ms at 500MHz) to more than 13 million clock cycles (26ms at 500MHz).

Table 2 - Performance information of adopted applications.

Application name	Single thread execution time	Computation to Data Ratio	Instruction size of a thread
MJPEG	5.3 Mega cycles	5.34	52kB
Smith Waterman	13.2 Mega cycles	7.08	3.8kB
FFT	1.9 Mega cycles	4.18	5kB

The second important factor provided by this table is the computation to data ratio. This number is the ratio between the number of actual compute instructions (e.g. add, mul, jump, etc.)

over the number of memory instruction (load / store) executed by a thread, which gives an overview of the importance of instruction loading in regards of data loading. Hence, an application having a high computation to data loading ratio would be instruction accesses dominated, with therefore proportionally less data accesses. Note that Smith Waterman has the highest computation data, when compared to the others. Fourth column in Table 2 shows the thread code size of each application. A thread with large code size would potentially lead to more instruction cache misses, because a smaller fraction of it would fit in the cache.

4.2 Reference Platforms

GEM5 Simulator [8] was used to produce the reference platforms. It was chosen because its good tradeoff between simulation speed and accuracy, besides modeling a real Realview Platform Baseboard Explore for Cortex-A9 (ARMv7 A-profile ISA) [9].

To create the desired ARM system, the CPU model and Linux bootloader have been modified to enable configurations with more than 4 cores and different interconnection network topologies were evaluated: (*a*) bus-based and (*b*) mesh. The reference platforms are configured as follows: (*i*) up to 8 ARM Cortex-A9, (*ii*) CPU running at 500MHz, (*iii*) Linux Kernel 2.6.38, (*iv*) 16kB private L1 data and instruction caches, (*v*) 32bits channel width, (*vi*) DDR physical memory running at 400MHz and (*vii*) 256MB unified L2 cache (only for mesh network) where the MOESI Hammer [10] is the cache coherence protocol used in these topologies. It is noteworthy that to avoid any traffic with DDR physical memory during benchmark execution was selected this huge L2 cache size.

4.3 Platform Setup

The platform is configured as follows: (*i*) 3x3 processor array, NoC with 4 position input buffers and 32 bits channel width, (*ii*) processor cache size was set to 4kB, 8kB and 16kB, 8 words per lines, direct mapped caches, (*iii*) 500MHz frequency for both processor nodes and NoC routers and (*iv*) CPU with hardware multiplier, divider and barrel shifter. Figure 5 shows the thread mapping used for the experiments. In those, shared data reside in the top-left processor node, and are accessed by the threads running in other nodes (one per node).

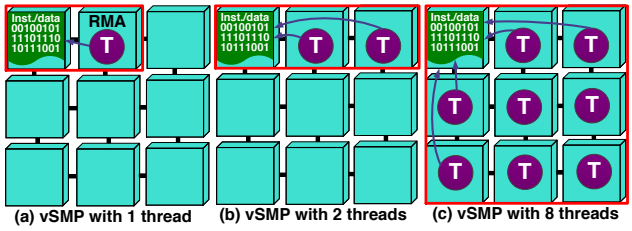


Figure 5 - Adopted scenario.

These mappings incur higher cache miss latencies as all traffic converges and flows through south and east ports of the top-left processor node. Results have however shown that mapping influence on performance remains below 5% in our setup, because of the NoC link bandwidth (1GB/s) corresponding to less than 20% bandwidth usage in all used configurations. All results were gathered on a synthesizable RTL VHDL description of the architecture. Note that all features inherent to prototype (e.g. run-time vSMP cluster definition) were also evaluated in FPGA (i.e. simplistic scenarios due to the memory limitation).

5. RESULTS

5.1 Speedup

The first experiment evaluates the platform scalability considering all applications as shown in Figure 7. Architecture scalability was evaluated and compared to different interconnection network topologies made from 1 to 8 ARMv7 CPU cores with cache sizes matching those of our vSMP MPSoC. In order to maximize speedup, one thread per node was used in all experiments, thereby avoiding performance penalties resulting from context switching. Figure 7a shows near-perfect linear speedup for the Smith-Waterman application, for all platforms. As being very compute oriented (as shown in the third column of Table 2), few data communication occur and thread code fit in the local caches for all tested configurations (as shown in the fourth column of Table 2), resulting in limited cache miss rate, but even under these conditions, our proposed MPSoC has a better performance compared to the mesh reference platform in which reached his maximum speed up of 6.3. In turn, b shows a similar speedup for cache sizes of 8kB and 16kB in the MJPEG, this can be easily explained by observing the behavior of the application shown in Figure 6, where it looks totally scalable, since the barriers would be found only at the end of each processing frame. But even though a scalable application, the shared-bus platform reaches a plateau from 5 cores, while the mesh platform reaches a plateau from 6 cores, they are due to the bus saturation. The same behavior is also observed with the proposed accelerator when using cache sizes of 4kB, but this will be explained in details in Section 5.2.

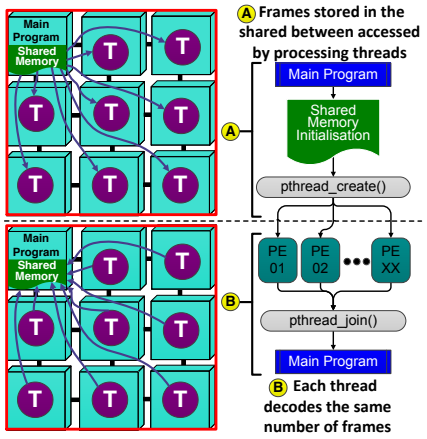
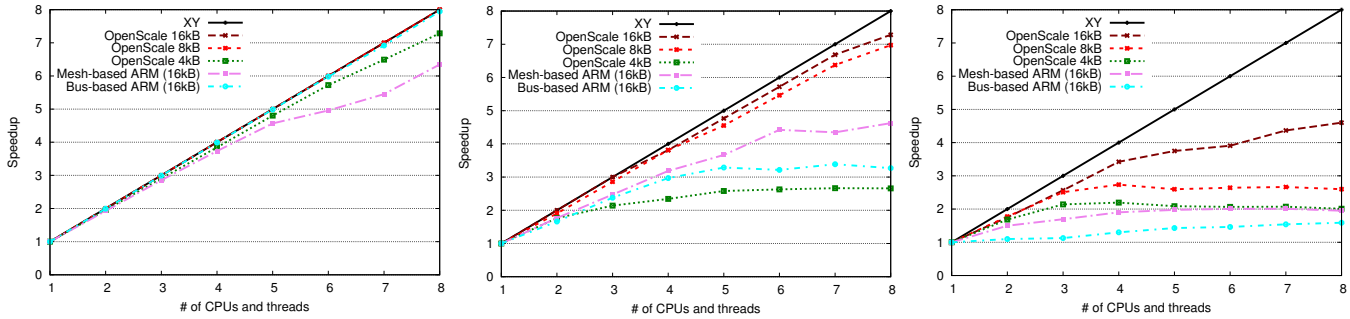


Figure 6 - Behavior of MJPEG Pthread implementation.

Figure 7c shows the performance scalability of FFT, regarding the adopted scenario (Figure 5). Due to the sequential synchronizations, the parallelization capability of FFT is low. Thus, with a 16kB cache configuration, the application can only achieve a speedup of 4.6. However, such results show a better performance when compared to other systems, which no one overcomes a speed up of 2.8. Different from the previous benchmarks, the FFT application was employed to explore synchronization primitives (e.g. Mutexes and Barrier) when multiple threads issue concurrent accesses to same data/variables (e.g. during a read/write operation). In this situation, Mutexes are used to allow exclusive Pthread data access and Barrier are used to coordinate the parallel execution of threads during their synchronization.



(a) Smith Waterman (b) MJPEG (c) FFT

Figure 7 - Architecture scalability, considering different applications and varying the cache memory size in 4kB, 8kB and 16 kB.

Another example to demonstrate this synchronization is shown in Figure 8, where the Mutex primitive is used to identify each thread (Thread_id) before starting their execution by using a single variable (shared_var) into the Host shared memory (i.e. instructions / data). This part of the application is inherently sequential due to the time of creation and identification of threads.

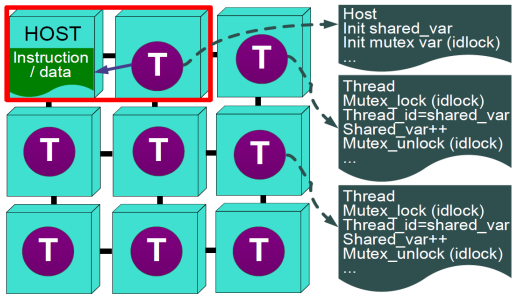


Figure 8 - Example of Mutex variable implementation.

5.2 RMA and NoC Throughput

Figure 9 shows the average bandwidth during MJPEG execution for the two used NoC links at the host level (south and east) as well as the RMA, which is the aggregated bandwidth of those. Thread mapping plays an important role in the NoC usage, as explained in Section 4.3.

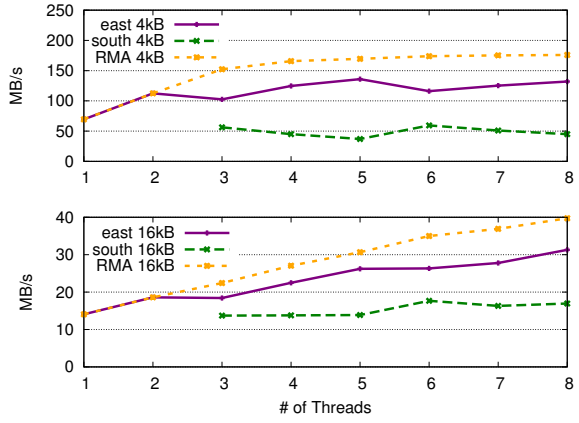


Figure 9 - Average bandwidth for the MJPEG application.

It can be observed that south link is unused for 1 and 2 threads, because of the decided mapping. But as the number of threads grows, the south link begins to be massively used (XY routing) causing most of the host data flows through this south NoC link. Although overall the bandwidth grows linearly versus the number of threads for 16kB cache sizes, which is similar to 8kB. Using 4kB cache sizes results in a significant bump in bandwidth usage, mostly because of much increased instruction cache miss rate. A plateau is observed from 4 threads at around 200MB/s, which is about 80% of the maximum theoretical bandwidth of the RMA module (250MB/s). This explains the plateau observed in speedup in Figure 7b, because of the RMA saturation.

Figure 10 shows the average bandwidth for the FFT application. Similar to 4kB MJPEG scenario, saturation threshold around 200MB/s is achieved when more than 4 threads are executed considering 4kB cache configuration (similar to 8kB cache). This explains the plateau observed for speedup illustrated in Figure 7c. However, the average bandwidth of the 16kB scenario does not reach the saturation threshold, which infers that the maximum speed up of 4.6 (Figure 7c) is not related to the communication, but rather to the application behavior explained in Section 5.1 and shown in Figure 6.

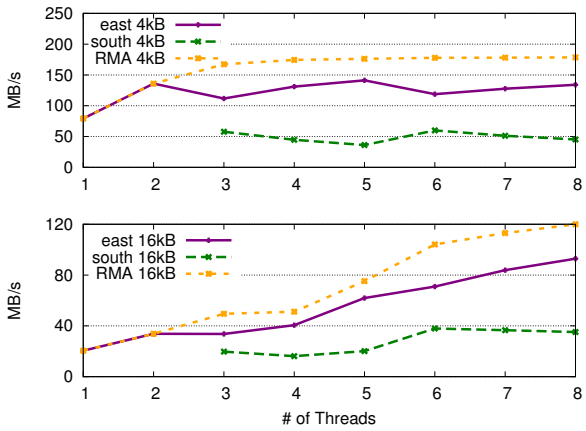


Figure 10 - Average bandwidth for the FFT application.

5.3 Miss Rate and Cache Miss Latency

Figure 11 shows both the average instruction and data cache miss latencies for MJPEG. Excellent scalability is observed for 2 cache configurations (8kB and 16kB), whereas 4kB configuration shows a steep latency increase due to the RMA saturation (Section 5.2). The average cache miss latencies for (a) instructions and (b) data for the FFT application for 4kB, 8kB and 16kB cache configuration is illustrated in Figure 12.

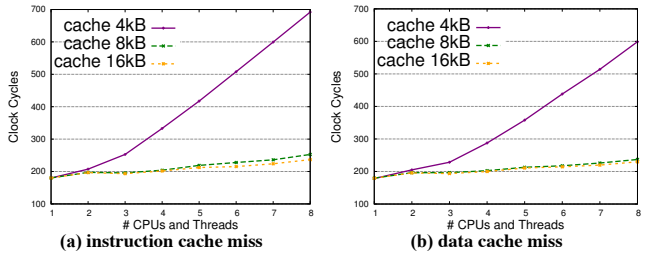


Figure 11 – Average instructions and data cache miss latencies for MJPEG.

Note that the saturation factor is related to the instruction cache miss latency. For instance, for the 4kB and 8kB scenarios the instruction cache miss latency increases almost linearly from 4 to 8 threads. In turn, due to the small number of data cache miss the latency does not have the same behavior.

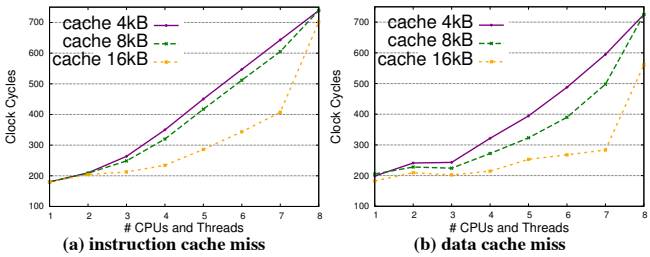


Figure 12 – Average instructions and data cache miss latencies for FFT.

5.4 Communication Overhead

As pointed out during the introduction, the placement of the desired cluster may interfere in the system performance. To demonstrate this situation the following scenarios are proposed: (*i*) 3 different clusters are defined at run-time by running the same FFT application without any communication between them, (*ii*) the application running in the first and third clusters has been modified to exchange results between them, while the second cluster will not run, and (*iii*) is a mix between the first two experiments, as shown in the Figure 13.

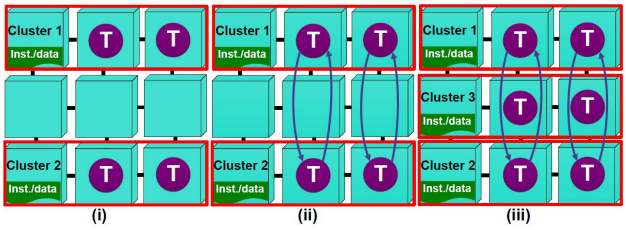


Figure 13 – Scenarios used to explore communication interference.

Table 3 shows that the communication overhead is 9.82%, and when the second cluster is running the first and third clusters are affected by the system and the execution time increases more 19.32%, showing qualitatively as interference can impact on the system.

Table 3 – Execution time considering scenarios with and without communication interference between clusters.

Scenarios	Execution Time (*Clock cycles*)	Interference
i	2356281	------
ii	2578409	9.42 %
iii	3076780	19.32 %

5.5 Placement Overhead

Another experiment which shows the placement overhead is demonstrated in Figure 14, where three scenarios are proposed varying the host position: (*i*) default scenario, where two threads will communicate through a port while all other threads will communicate using another, (*ii*) similar to previous, this scenario will decrease NPU-Host distance from 4 to 3 hops and (*iii*) the best scenario, where NPU-Host distance is balanced and in the worst case is just 2 hops away.

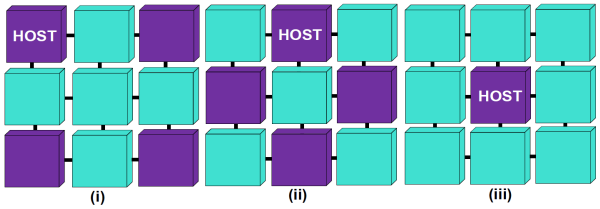


Figure 14 – Scenarios used to explore host position.

As displayed in Table 4, the best gain using cache size of 8kB or 16kB is about 3% by changing only the host position, this can be explained by the fact that the MJPEG application is scalable and the NoC was not saturated in these scenarios. But when the NoC is saturated (4kB cache size), the placement can be very weighty, showing a gain of almost 10% using the best configuration.

Table 4 – Larger differences exploring the host position.

Cache Size	Performance best case		
	Host_00	Host_10	Host_11
4 kB	100%	98.81%	90.51%
8 kB	100%	97.10%	96.84%
16 kB	100%	98.62%	96.90%

5.6 Area Overhead

To evaluate the area overhead of the proposed idea, two approaches were used: (*i*) ASIC and (*ii*) FPGA.

In these results, the processor architecture was shifted to a floating-point capable low power embedded processor [11]. Moreover, the synthesizable version has no optional IO (e.g. no UART). Table 5 shows that this contribution adds an area overhead between 5,72% and 15,59%, using respectively 256kB and 64kB of RAM in a 40nm CMOS Technology.

An additional analysis was performed using a Xilinx Spartan-3 FPGA (XC3S1000), where system with the RMA module incurs an area overhead ranging from 7% to 15% also depending on the memory size.

Table 5 – PE area evaluated at 40nm CMOS Technology.

Memory Size	Low-Power core with FP	Low-Power core with FP + RMA	Area Overhead
64 kB	0.2803 mm^2	0.3240 mm^2	15.59 %
128 kB	0.4392 mm^2	0.4829 mm^2	9.94 %
256 kB	0.7650 mm^2	0.8088 mm^2	5.72 %

6. CONCLUSION

This paper presents a multithreading approach for a distributed memory NoC-based MPSoC. The proposed solution was validated on FPGA and ASIC and incurs a limited area overhead (around 10%), because it relies on a relaxed memory consistency model that permits avoiding complex cache coherence protocols. This combined to the intrinsically low latencies of on chip NoC communications results in a promising solution that shows good performance scalability and enables facilitated programming through the use of POSIX thread API.

Future explorations will aim at investigating power efficiency and scalability with hundreds of processor nodes, besides also evaluate qualitatively our approach against NUMA systems. Some enhancements of this design are also envisioned, such as aliasing of both data and instruction spread over several processor nodes so as to better balance memory accesses, avoid code replication (such as microkernel) and achieve higher performance.

7. ACKNOWLEDGEMENT

The research leading to these results has received fund- ing from the European Communitys Seventh Frame- work Programme (FP7/2007-2013) under the Mont-Blanc Project: http://www.montblanc-project.eu, grant agree- ment no 288777.

8. REFERENCES

[1] Marculescu, R., Ogras, U.Y., Li-Shiuan Peh, Jerger, N.E. and Hoskote, Y. 2009. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28(1), 3-21.

[2] Marongiu, A. and Benini, L. 2010. An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. *IEEE Transactions on Computers*, vol. 62(1), 222-236.

[3] Ceng, J. 2008. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Design Automation Conference* (DAC), 754-759.

[4] 9945-1I. 1996. The POSIX threads standard.

[5] Baert, R. 2009. Exploring Parallelizations of Applications for MPSoC platforms using MPA. In *Design, Automation and Test in Europe Conference* (DATE), 1148-1153.

[6] Ophelders, F. 2009. A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs. Master's Thesis, Eindhoven University of Technology.

[7] Busseuil, R., Barthe, L., Almeida, G. M., Ost, L., Bruguier, F., Sassatelli, G., Benoît, P., Robert, M. and Torres, L. 2011. Open-scale: A scalable, open-source noc-based mpsoc for design space exploration. *International Conference* on *Reconfigurable Computing and FPGAs*. 357–362.

[8] Binkert N. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, vol.39 (2).

[9] ARM. 2011. RealView Platform Baseboard Explore for Cortex-A9 User Guide, DUI0440B.

[10] AMD, 2002. AMD Opteron Shared Memory MP Systems.

[11] ARM, 2009. Cortex-M4 Technical Reference Manual, DDI0439C.