Efficient Embedded Software Migration towards Clusterized Distributed-Memory Architectures

Rafael Garibotti, Anastasiia Butko, Luciano Ost, Abdoulaye Gamatié, Gilles Sassatelli and Chris Adeniyi-Jones

Abstract—A large portion of existing multithreaded embedded sofware has been programmed according to symmetric shared memory platforms where a monolithic memory block is shared by all cores. Such platforms accommodate popular parallel programming models such as POSIX threads and OpenMP. However with the growing number of cores in modern manycore embedded architectures, they present a bottleneck related to their centralized memory accesses. This paper proposes a solution tailored for an efficient execution of applications defined with shared-memory programming models onto on-chip distributed-memory multicore architectures. It shows how performance, area and energy consumption are significantly improved thanks to the scalability of these architectures. This is illustrated in an open-source realistic design framework, including tools from ASIC to microkernel.

Index Terms—Multicore programmability, software migration, distributed shared memory, performance scalability.

1 Introduction

The coming years will see embedded systems massively adopting manycore architectures with up to thousand of cores. Such architectures will be clusterized to improve system performance and management. The state-of-the-art ARM big.LITTLE technology [1] exploits clusterization to implement a heterogeneous architecture. Core clusterization is also present in embedded manycore platforms such as MPPA [2], HyperCore [3] and STM Platform 2012 [4]. This favors architecture scalability (by cluster replication), and accelerates the recently observed convergence of embedded and high-performance computing (HPC) [5].

Since the number of clusters will increase with the foreseen high number of cores in future systems, distributed memories will likely become mainstream for mitigating memory access bottleneck. This shift from shared monolithic memory to distributed memory will have a significant impact on legacy embedded software migration and future embedded application programming. In [6], the authors already raised the growing software size (billions of object code instructions) in embedded systems over last five decades in automotive, spatial and telecommunication domains. The resulting complexity poses a major challenging question: how to execute such software with scalable performance and energy-efficiency on future clusterized manycore embedded architectures?

The answer to the aforementioned software migration challenge necessarily calls for a cost-effective programmability addressing well the characteristics of embedded applications on those architectures. Embedded applications, unlike general-purpose, are often integrated with microkernels. Their programming on multicore architectures accommodates software/hardware allocation where each task is associated with its own core. This is especially convenient for applications with safety-critical constraints so as to reduce undesired task interferences. On the other hand, usual embedded programming models assume a *centralized shared memory* (CSM) design, i.e. where monolithic memory block is shared by all cores for data sharing and communications. Existing legacy embedded software massively relies on this paradigm. Shared-memory oriented programming has been attractive due to its simplicity: no explicit communications by message passing between execution entities, easy implicit sharing of data object across threads, etc. However, in the perspective of future distributed-memory clusterized architectures, the migration of legacy embedded software and the implementation of emerging embedded applications must reconsider the current practice in programming so as to answer the aforementioned performance and energy-efficiency challenge. Possible solutions can be software translation to new programming languages, compilers or operating systems.

This paper promotes the following contributions:

- an efficient approach based on light and judicious software/hardware modifications enabling the execution of typical shared memory multithreaded embedded software on clusterized *distributed shared memory* (DSM) embedded multicore architectures;
- a POSIX-like threads API providing code portability, where the associated runtime library is modified by defining a new software stack allocation. Then, a logically shared-memory vision is made available on top of the considered physically distributed memory architecture so that applications are transparently executed without significant changes in source code;
- some improvements in performance scalability and energy-efficiency enabled by the proposed approach, considering different benchmarks executing on phy-

- *R. Garibotti, A. Butko, A. Gamatié and G. Sassatelli are with LIRMM lab., Montpellier, France.E-mail: Firstname.Lastname@lirmm.fr*
- *L. Ost was with LIRMM during this work. He is now with Dpt of Engineering at Univ. of Leicester, UK.E-mail: luciano.ost@leicester.ac.uk*
- *C. Adeniyi-Jones is with ARM, Cambridge, UK. E-mail: Chris.Adeniyi-Jones@arm.com*

sically shared and distributed memory clusterized architectures, within a complete realistic environment, including tools from ASIC up to microkernel.

In the rest of this paper, Section 2 discusses related work. Section 3 summarizes the principles of our approach by introducing considered CSM and DSM designs. Section 4 details the implementation of the approach in an open-source framework. Section 5 presents experimental results regarding speedup, area, energy and power consumption. Finally, Section 6 points out conclusions and future work.

2 Related Work

THERE has been a rich literature on software modernization as surveyed in [7]. Two main approaches are distinguished: black-box *versus* white-box. Their application depends on the level of understanding required for moving from source software to target software. Usually the black-box approach only requires the analysis of software inputs and outputs. Then, wrappers enable to adapt the software interface within its target environment. The white-box approach requires a deeper analysis and understanding of software internal details. Typically, it can rely on static code analysis of shared data for code parallelization and optimizations during migration. Concrete examples of white-box techniques are migration of languages or operating systems via automated translation. In [8] and [9] authors proposed a semi-automatic program partitioning of sequential legacy code into process networks that are afterwards distributed onto multicore or MPSoC platforms.

The above approaches work well when target implementations adopt a model that does not radically differ from that of source implementations. The gap between shared-memory and distributed-memory execution models will make such techniques hardly applicable. Indeed, deciding automatically an efficient code and data distribution/parallelization requires a non-trivial analysis going beyond code parsing and translation only. In our solution most of modifications focus on specific runtime library, which is wrapped in such a way that all memory access requests are adequately managed by a specific tiny hardware module named *remote memory access* (RMA). Any access to the (virtual) shared-memory space assumed at application level is translated by RMA into an access to the (actual) distributed-memory. In our opinion, the efficient migration of embedded applications on future manycore architectures should go beyond software by covering hardware part as well for scalable performance and energy efficiency.

The DSM architecture model borrowed here, often relies on software support (compilers and runtime libraries) to achieve memory coherence. Cores have a non-uniform memory access, i.e. memory access latency depends on the memory location relative to the processor. The OmpSs framework [10] exploits program annotations in its compiler and runtime system to enable the execution on clusterized GPU-based systems. In [11], an OpenMP-ready DSM system relies on memory access pattern analysis to offer a low overhead coherence maintenance. Beyond HPC domain, DSM has been recently applied to embedded systems [12] [13] [14] to deal with their increasing number of cores. Its application to on-chip systems must carefully take into account crucial constraints on limited silicon area and power budget.

In [12] authors present a DSM design of multiprocessor System-on-Chip (MPSoC), where each node contains a LEON3 core, private and shared memories, and an engine to handle memory accesses and Network-on-Chip (NoC) communications. They compare the performance gain between their DSM design and a CSM design, with a H.264 decoder running on a 3x3-mesh NoC. Their approach does not address area and power issues, and no programming facility is given for migration. In [13] authors explore the reduction of virtual-to-physical address translation overhead in a NoC based DSM design. They consider a dual micro-coded controller (DMC) similar to the RMA module considered in our approach. While RMA is tiny for minimizing silicon area, DMC has larger size due to dynamic memory partitioning and its maximum theoretical bandwidth is limited by virtual-to-physical address translation. In [14] a cache coherence protocol is defined for large-scale NoC-oriented shared memory architectures. The NoC nodes are grouped into clusters, each of which consisting of a group of L1 cache banks and one L2 cache bank. Cache coherence is enforced hierarchically. The performance evaluation of the implemented memory system shows a speedup of three when the size of processed data increases, but neither execution time nor energy consumption are provided.

Compared to above DSM design approaches for embedded systems, our approach aims to provide an efficient sharing of local memories from different cores so as to scale system performance by aggregating the bandwidth of available distributed memories. It also targets a minimal silicon area and system energy consumption. By concentrating all re-design efforts at runtime software and hardware levels, it facilitates the execution of shared-memory oriented embedded applications on distributed-memory multicore on-chip systems with very limited modifications. This is simpler and cost-effective for legacy software migration.

3 From CSM to DSM: General Approach

FROM a general point of view, large-scale clusterized distributed-memory architectures are hierarchical systems in which every sub-system comprises its local computing resources including a number of cores sharing local memory. The software programming on top of such systems typically considers either explicit communication transactions through message-passing with an API such as MPI; or alternative programming models associated with software techniques for memory coherence between the different local memories. The recent partitioned global address space (PGAS) programming model [15] borrows features of both shared memory and distributed memory programming models. Here program variables share a common address space, and are accessible to all processes (or threads). The address space is logically partitioned in such a way that a notion of proximity to a particular memory section is made available to processes. This provides the necessary locality information for efficient and scalable mappings of data onto both shared and distributed memory hardware.

At sub-system level, a symmetric multiprocessing (SMP) architecture model is usually adopted where all cores uni-

formly have access to the centralized shared memory. Such CSM sub-systems guarantee memory coherence, which implies that modification to any memory location by a core becomes immediately visible to other cores through cache coherence protocols [16]. Shared memory parallel programming API, e.g. POSIX threads, assume this feature.

From now on, we mainly concentrate on sub-system level to illustrate the DSM-based embedded software migration proposed in this paper. We show that DSM provides better performance scalability at that level by removing the bandwidth sharing bottleneck inherent to CSM. Our approach is inspired by the PGAS paradigm. However, unlike PGAS languages, it does not manage data locality via program annotations or macros in order to avoid modifications of legacy code and compiler. Figure 1 shows considered NoC-based tiled multicore architecture. Each tile, represented by a blue square, contains a core. Each cluster represented by red dashed-line boxes is composed of a number of tiles. This number can differ from a cluster to another. Two memory organizations are shown. The left-hand side illustrates a clusterized centralized shared memory design where each cluster comprises one *host core tile* (displayed on top-left in the cluster) that executes the main thread (or task) of an application and stores entire shared memory including all data and application code instructions. The remaining worker threads are executed on the other tiles, and therefore access shared data located on the host core tile remotely.

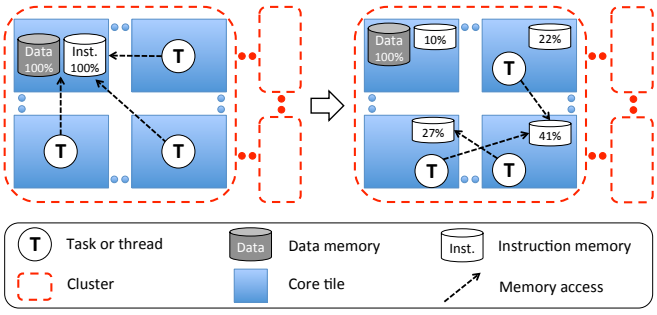


Fig. 1. Clusterized NoC-based multicore architecture: CSM vs. DSM.

The right-hand side of Figure 1 depicts our DSM design approach. The instruction memory storing application code is split over the different tiles within each cluster. This leads to a spatial distribution of memory traffic and decreased instruction cache miss latency compared to CSM design. As application code is read only no cache coherence protocol is required. Each tile shares part of its resulting local instruction memory with the other tiles, thus resulting in a globally unified shared memory address space. The main application thread is executed by the host core tile, which also stores all data memory. The host core tile executes no worker thread but handles creation, synchronization and deletion of worker threads. It, therefore, acts as a memory server and synchronization handler. Shared data are cached and cache coherence is achieved via a relaxed memory consistency model relying on thread synchronization.

The reason behind our decision to distribute only code (and not data) lies in the SIMD nature of the targeted application kernels in which dozens if not hundreds of threads executing the same code are spawned: all those threads fetch the same instructions, albeit at different time instants. Distributing code results in a significant increase in efficiency by avoiding redundancy and enabling large kernels to fit across the multiple memories.

Note that as our approach does not deal with explicit allocation of application in memory, critical code regions may not be homogeneously distributed across the shared memory address space. This implies variable numbers of requests to each distributed memory part.

4 Implementation of our Approach

We consider the open-source and customizable NoC-based MPSoC platform [17] implemented at RTL level. A very interesting feature of this customizable multicore platform is its ability to enable the creation of clusters according to CSM design (see left-hand side of Figure 1). Each cluster is composed of a shared memory and a number of tiles. Applications mapped onto a cluster share memory through the host tile while those mapped onto different clusters transfer data via message-passing. It consists of a tiled multicore platform comprising: a light embedded core with L1 cache memory based on microblaze instruction set architecture; a NoC router based on Hermes [18]; an internal scratchpad memory (SPM) to store application code and microkernel (derived from [19]); a timer, an interrupt controller; and a RMA hardware module, which is responsible for accessing remote SPM. POSIX threads are used as shared-memory parallel programming model.

Given the above platform, we design a new DSM architecture based on the template in Figure 1 so as to compare application code executions on both CSM and DSM architectures for software migration purpose. We present below the main modifications required for achieving it. The resulting open-source research platform is available for download [1].

4.1 Modifications of application, runtime and hardware

To incorporate DSM capabilities in the chosen platform [17], we apply a number of modifications summarized in Figure 2. The concerned parts are highlighted: microkernel, software stack allocation and RMA module.

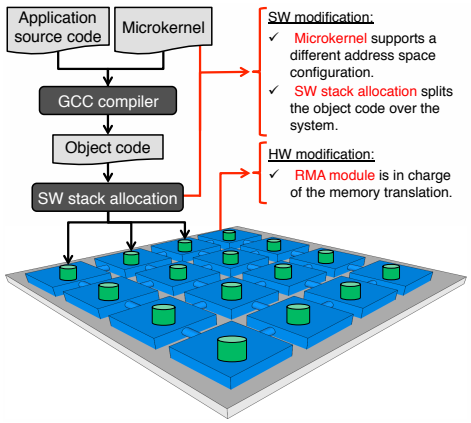


Fig. 2. DSM implementation flow highlighting the modifications implemented in the reference platform.

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

4.1.1 Modification of input application code

Input application code requires no modification compared to generic POSIX thread compliant code, but replacing standard header files with those specific to the proposed architecture as highlighted in Figure 3. Function prototypes and functionalities (see [17]) remain the same, thereby making the approach generic as legacy multithreaded code can be ported over to the DSM architecture with little to no effort. The underlying libraries handle all of the architecture specific mechanisms that ensure proper application execution.

```
#include <stdlib.h>                    #include “libc_API.h”
#include <stdio.h>                     #include “shared_memory_API.h”
#include <string.h>                    #include “pthread_API.h”
#include <time.h>
#include “jpeg_lib.h”                  #include “jpeg_lib.h”
#include “mylib.h”                     #include “mylib.h”
#include “4_Images2.h”                 #include “4_Images2.h”

#define NUM_THREAD 15                  #define NUM_THREAD 15

int main(void){                        int main(void){
   int I;                                 int I;
   int ID[NUM_THREAD];                    int ID[NUM_THREAD];

   //Init Threads                         //Init Threads
   pthread_t thread[NUM_THREAD];          pthread_t thread[NUM_THREAD];
   for(i=0; i<NUM_THREADS; i++)           for(i=0; i<NUM_THREADS; i++)
      ID[i] = I;                             ID[i] = I;

   for(i=0; i<NUM_THREADS; i++)           for(i=0; i<NUM_THREADS; i++)
      Pthread_create (&threads[i], NULL,     Pthread_create (&threads[i], NULL,
Processframe_main, (void *), &ID[i]);  Processframe_main, (void *), &ID[i]);

   for(i=0; i<NUM_THREADS; i++)           for(i=0; i<NUM_THREADS; i++)
      pthread_join (threads[i], NULL);       pthread_join (threads[i], NULL);

   return 0;                              return 0;
}                                      }
```

Fig. 3. MJPEG main program encoding: SMP (left) *vs.* DSM (right).

4.1.2 Runtime software level

The first modification concerns the configuration of the address space in the microkernel. Indeed, the original architecture has one of the following configurations: *i)* each tile has the same SPM size with private and shared memory address spaces, so the host tile can be located in any tile. This flexibility affects the silicon area of the entire system since only the host tile can use the shared memory address space; *ii)* tiles have different SPM sizes, implying more complexity in the microkernel and a mapping preset for host tile. As DSM design has no restriction on private and shared memory address spaces, it is possible to have the smallest required silicon area with the higher flexibility.

In the reference platform, the entire application code is placed in the host tile memory while the other tiles only contain the microkernel object code. These tiles then execute threads whose instructions are located on the host tile, through fetching required data whenever a cache miss occurs. In the DSM implementation, a dedicated script takes care of application code generation. It is responsible for compiling separately application code, resulting object code is then spread across the cluster. For example, a N kB binary is divided in M slices of N/M kB, where M is the number of participating tiles configured to hold a portion of the instruction object code. Each slice is linked to a shared memory address region belonging to a given tile address space. Default mapping performs descending distance installation, i.e. code gets gradually installed from the farthest to the closest tile (with reference to the host tile).

Note that only code gets distributed across the tiles' SPMs: this facilitates implementation as no cache coherence mechanism is required, and allows for a much better exploitation of on-chip SPM as no thread code is replicated.

4.1.3 Hardware level

From a software perspective, the microkernel has a vision of a unified shared memory address space, though physical memories are distributed across tiles. This abstraction is made possible by adapting the RMA hardware module [17] so that each and every tile may both fetch data from remote tiles and serve incoming memory requests.

Whenever the tile-local core L1 cache issues a cache line request, the RMA first checks whether the corresponding data is available locally or remotely, through simple address decoding. Should the address be mapped to a distant tile memory, the RMA issues a request routed through the NoC to the target tile. The remote tile RMA then serves that request and returns desired cache line content.

The RMA is both connected to the NI and the local bus on which it operates similar to a DMA (Direct Memory Access) engine. For performance reasons, the RMA possesses a buffer having for size that of a cache line (256 bits). Furthermore, one key factor of the overall memory performance is the RMA latency, which greatly depends upon NoC features such as channel width. The specific protocol and its implementation have been optimized for latency, not for throughput, as cache miss traffic is rather more latency-sensitive. Low to moderate latencies are ensured thanks to implemented buffers, which mitigate for aliased requests.

Finally, the RMA module requires an addressing scheme to enable distant memory access. For legacy reasons, the memory mapping of the 32-bit range is divided as follows: *i)* the four highest bits are kept for the selection of a local component that is connected to the core, *ii)* bits 20 to 27 are used to select the coordinates of the tile memory in case of a remote access, and *iii)* the last 20 bits are used for memory address. This allows for simpler memory management as local access can be detected using the same strategy for each tile. Moreover, due to this memory mapping, it is possible to address up to 256 (16x16) tiles in a cluster, each of which having up to 1 MB of memory.

4.2 Memory coherence

The POSIX threads API used for both CSM and DSM systems has functions belonging to three main categories: thread creation, mutexes and barriers. Though shared data remain located on the host tile's SPM, consistency has to be insured as those data are cached on the remote tiles.

The relaxed memory coherence model adopted in our DSM approach makes memory coherent at synchronization points, i.e. whenever an API function is called. For that purpose, the API requires shared data be explicitly flagged as such, so that execution confluence is guaranteed. This accounts for the only difference in function prototype with the POSIX thread API, all others being otherwise equivalent. Data consistency is therefore purely handled in software.

According to that explicit flagging of shared data, invalidation and flush of a given cache line occur only if cache line tag corresponds to the address specified by

TABLE 1
Summary of architecture set evaluated throughout all experiments.

Cluster size	4x4
Communication	Dual 32 bits channel NoC @ 500MHz
CPU core	32 bit, 5 pipeline stage Microblaze ISA @ 500MHz
CPU caches	2kB-16kB direct mapped L1 I$ and D$ caches, 256 bit/line
Tile local shared RAM	2kB-16kB
Tile local private RAM	64kB microkernel + 32kB code/data
Thread assignment	1 worker thread/tile for avoiding performance penalties from context switching, main thread on master tile
Target technology	45nm CMOS bulk - FreePDK library [20]
Memory information	Evaluation by using NVSIM tool [22]

the instruction. This condition avoids unnecessary cache flushes/invalidations of cache lines containing unrelated data. An extensive description of the memory coherence model is given in [17].

5 Experimental Results

We evaluate the performance and energy consumption gains of the DSM architecture design proposed in previous sections in comparison with a CSM design.

5.1 Setup information

We consider a synthesizable RTL description in VHDL for both CSM and DSM architecture templates. Table 1 gives the details of the distributed-memory multi-core architecture.

Three application kernels with different profiles are used to evaluate the two platform types: *MJPEG video decoding* from multimedia domain; *Smith-Waterman* (SW) algorithm used to find similar regions in DNA sequences and *advanced encryption standard* (AES), a worldwide used cryptography application kernel. Their evaluation follows the simulation flow illustrated in Figure 4 according to the following steps:

1) a netlist is obtained by logic synthesis of the multi-core architecture using an open source design kit for 45nm CMOS technology [20], the RTL VHDL system description and block constraints to detect, e.g., glitches, slow paths and clock skew;
2) the target application and the microkernel are compiled using GCC compiler (version 4.4 or above). The output is the object code for each input file, mapped onto clusters according to Section 4.1.2;
3) the object code produced by combining the application and the microkernel is simulated with the netlist in the Cadence Incisive Simulator [21], producing performance values;

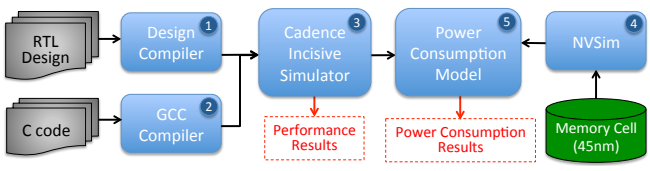


Fig. 4. Simulation flow used to validate the proposed architecture design.

4) memory property is evaluated with NVSim tool [22], e.g. leakage and dynamic power consumption,
5) energy and power consumption results are extracted using the power consumption model based on memory access requests and the NoC data communication volume collected from simulation.

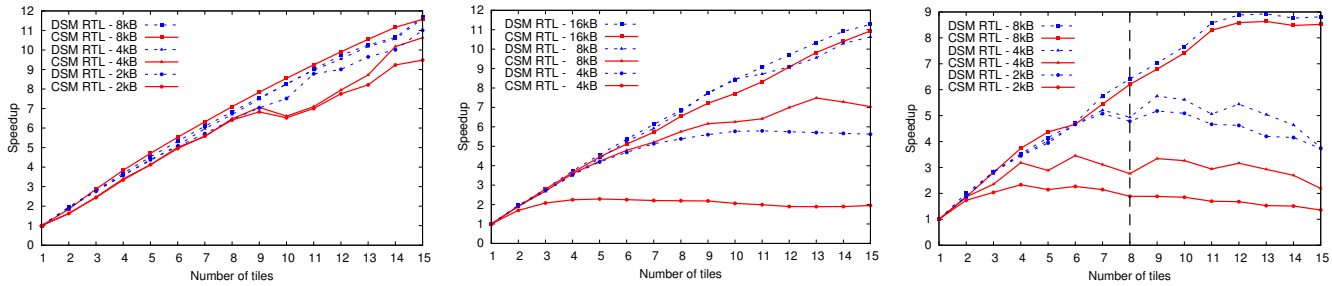
5.2 Speedup Evaluation

Three multithreaded application kernel workloads are executed on CSM and DSM platforms in order to compare performance gains and penalties inherent to both implementations. The considered application kernels have different profiles. SW has small code size and is very compute oriented, which results in limited cache miss rate and near-linear scalability. On the opposite side, MJPEG has a larger code size. The results show good scalability because synchronization barriers are found only at the end of each processing frame. For small cache sizes, an impact on performance is observed with a speedup limitation. Finally, AES has significant amounts of synchronizations where multiple threads issue concurrent accesses to same data/variables, making their parallelization capability low.

Figure 5 shows speedup versus number of tiles in a cluster according to CSM and DSM. The execution time of each benchmark is normalized to the minimal cluster size (1-tile) to facilitate comparison. Results show that by aggregating the bandwidth of available multiple distributed memories, applications present performance improvement of upto 3 times in the DSM design (Figure 5.*b* with 12 core tiles and 4kB cache size). Nevertheless, a plateau is observed for application kernels beyond a certain number of tiles. AES shows a plateau reached from 4 tiles in CSM by using 4kB cache size, whereas in DSM gains are observed up to 8 tiles (see dashed-line in Figure 5.*c*). The resulting performance of DSM also relies on the fact that threads may fetch codes from multiple instruction memory instances, which are distributed along the platform (Figure 6.*a*).

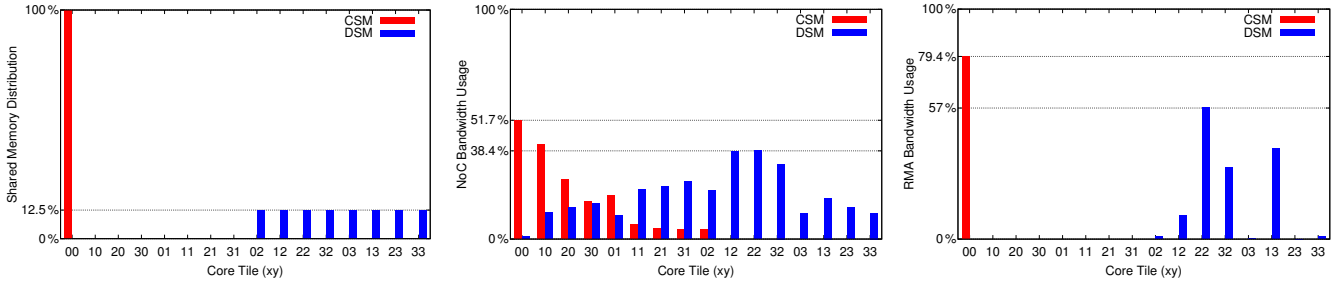
In order to deeply investigate the plateau occurrence on both systems, AES is chosen to have a communication-oriented behavior, besides being the type of application with the biggest challenge for such architectures. This results in high pressure on the communication/memory subsystem. AES kernel is evaluated according to the following monitoring information presented in Figure 6: *i*) shared memory distribution, *ii*) NoC bandwidth usage and *iii*) RMA bandwidth usage. Here, the comparative DSM design is configured by installing application kernel code over 8 tiles out of the 16 tiles available in a cluster (dashed lines in Figure 5.*c*).

Figure 6.*a* clearly shows the application kernel code installed on host core tile (CSM) or distributed across several tiles (DSM). The resulting bandwidth of NoC and RMA shows that in CSM all the traffic naturally converges to the host tile. This leads to a communication and memory access bottleneck. DSM system does not suffer from this as communication load gets distributed across several tiles. Behavior may significantly differ from an application to another as critical code regions are not homogeneously distributed across the address space. The NoC traffic is therefore uneven with peaks observed in specific regions. In



(*a*) Smith Waterman (*b*) MJPEG (*c*) AES

Fig. 5. Performance evaluation for different embedded applications showing the benefits achieved with DSM design.



(*a*) Shared Memory Distribution (*b*) NoC Bandwidth usage (*c*) RMA Bandwidth usage

Fig. 6. Evaluation of different metrics (memory, NoC and RMA Bandwidth usage) for a 4x4 tile array executing AES application with 8 threads.

both cases for this evaluation, NoC usage remains modest because of considered low number of tiles in a cluster.

Figure 6.*c* also shows the average bandwidth usage for the RMA module. In CSM, the observed peak RMA through-traffic is about 80% of its maximum theoretical bandwidth [17]. This explains the observed plateau in application speedup. In DSM system, though bandwidth is more distributed, resulting in better speedup, a similar behavior is observed for one of the participating tiles (i.e. core tile 22 in Figure 6.*c*). This again originates from the logical uneven occurrence of cache misses across the address space, some of the AES core functions being hosted on that tile. As the cluster grows, this RMA tile will become the system bottleneck, causing an increase in cache miss latency. This results in a longer execution time.

Results show that the hardware-level modification (RMA) in DSM plays an important role in speedup enhancement, bringing no performance penalty compared to reference design. An alternative approach can rely on task migration [23]. But, this can incur a possible performance penalty due to the time required for transferring thread code, beyond the challenging implementation of all required mechanisms on the small memory footprint available in embedded systems.

5.3 Area Optimization and Power Consumption

Our approach opens on-chip area saving opportunities in DSM through decreased L1 cache memory sizes while maintaining similar performance compared to CSM system.

Table 2 shows tile area savings resulting from various reductions of the core L1 cache memory size. Up to about 16% savings can be achieved depending on the design decision. At least 2.5% of tile area can be saved as DSM design with 2kB cache memory always achieves higher scalability performance than a 4kB L1 cache memory in CSM. This analysis is important because it helps to decide which configuration is better in an ASIC project.

TABLE 2
Tile area saving vs. L1 Cache Memory.

Chosen L1 cache memory size	Initial L1 cache memory size		
	4 kB	8 kB	16 kB
2 kB	2.43%	7.17%	16.72%
4 kB	———	4.62%	13.95%
8 kB	———	———	8.91%

Beyond the need of performance increase and area minimization, future clusterized manycore embedded architectures will also require energy consumption reduction. In the following, we consider a modeling of power consumption in order to analyze the impact of both CSM and DSM designs on energy consumption. The system power consumption can be divided into 3 parts: local SPM memories, NoC and cores. For simplicity purpose, both designs are assumed processor-agnostic and only power and energy related to memory and NoC are evaluated. Monitors have been used throughout the system to collect the required information.

As a result, energy and power consumption models are created based on [24], leading to the following equations:

$$E_{MEM} = (nb\_RD * E_{read}) + (nb\_WR * E_{write}) + (nb\_tiles * (L_{MEM} * ExecTime)) \quad (1)$$

$$E_{Router} = (nb\_Flit * E_{router}) + (nb\_tiles * St\_E_{router}) \quad (2)$$

$$PowerCons = \frac{E_{MEM} + E_{Router}}{ExecTime} \quad (3)$$

where E_{MEM} and E_{Router} respectively denote the energy consumed by memory and routers while their sum provides

the global consumed energy used to define $PowerCons$, representing the overall power consumption. The meaning of each variable in above equations is given as follows:

nb_tiles	:	total number of core tiles,
$ExecTime$	:	application kernel execution time,
nb_RD	:	total number of reads occurred in all distributed memories in the system,
nb_WR	:	total number of writes occurred in all distributed memories in the system,
E_{read}	:	dynamic energy consumed in one read from the memory obtained through NVSim,
E_{write}	:	dynamic energy consumed in one write to the memory obtained through NVSim,
L_{MEM}	:	memory leakage power obtained with NVSim,
nb_Flit	:	total number of flits passed on all routers,
E_{router}	:	dynamic energy consumed when a flit passes on the router, obtained via synthesis,
St_E_{router}	:	synthesized static energy of router.

To characterize the above models, memory-related values are obtained with the NVSim energy profiling tool [22], while router-related values are obtained through logic synthesis using the FreePDK [20] open-source design kit for 45nm technology.

Figure 7 illustrates power and energy-to-solution figures for the AES application kernel in both CSM and DSM systems. As expected, DSM system incurs a higher power peak for 2kB and 4kB cache sizes because of the shorter execution time. The global consumed energy is, however, less for the same reason, yielding an overall energy efficiency. For the same configuration, a decrease of upto 50% of energy dissipation is observed in DSM compared to CSM systems.

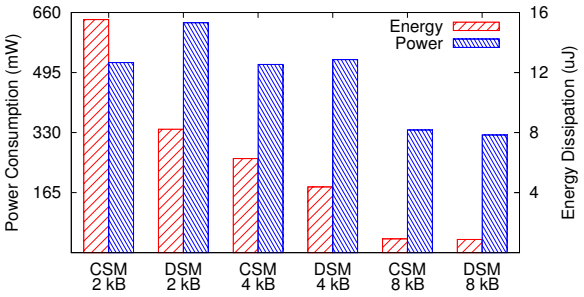


Fig. 7. Energy and power comparison: DSM vs CSM.

A temporal power consumption profile is given in Figure 8. It is important to note the significant power fluctuations in the DSM plot. This is due to the cache misses emitted to different target memory regions, physically located on different tiles: path of different lengths (NoC hop count) are sequentially activated in the NoC, as shown in Figure 6.*b*, then resulting in significant fluctuations in consumed power. Furthermore, the area saving opportunities indicated that a reduction on core L1 cache memory size from 4kB to 2kB in DSM design has a performance equal or higher than in CSM design with 4kB of cache memory. However, Figure 8 shows that this performance increase causes an overhead both on energy and power consumption.

The different experimental results presented above confirm our software migration approach is promising for future clusterized DSM manycore embedded architectures.

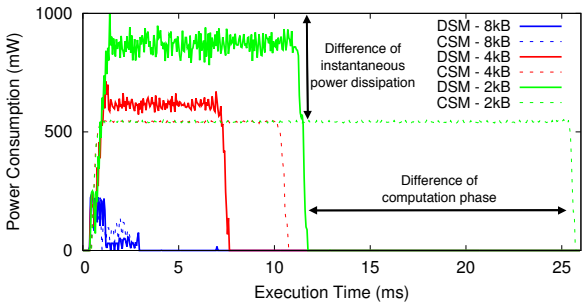


Fig. 8. Instantaneous power dissipation with similar configurations.

Unlike several existing approaches that often advocate application software translation to new programming languages, compilers or operating systems via virtualization, we believe for embedded systems (due to their particularity such as a tight integration of applications with microkernels) hardware level modifications together with limited but judicious adjustments of runtime libraries are the key ingredients for scalable performances and energy-efficiency. The modifications of legacy code in this context are very limited, then avoiding tedious and error-prone code rewriting. For new software, the mainstream shared-memory programming model can still be used to implement future applications on target architectures. We also notice the central role that the RMA hardware module plays in performance improvement when considering clusterized DSM design. Such a mechanism contributes to an overall efficient bandwidth exploitation in the system.

6 Conclusion and Perspectives

This paper addressed the execution of multithreaded software originally written for centralized shared-memory (CSM) architectures onto clusterized distributed shared-memory (DSM) manycore architectures with very limited modifications of their original code. This contributes to answering the challenging software migration issue increasingly required by new architectures. The proposed solution advocated runtime and hardware level modifications to enable scalable performances and energy consumption of applications on clusterized architectures. Experimental results showed that DSM features substantial performance scalability improvements over CSM. In DSM, as SPM on every tile is made available to all others, this allows for further performance benefits, as less data converge to the same tile. The paper also demonstrated the opportunity of reducing implementation cost by means of decreasing cache sizes with DSM strategy. For a same given configuration, a decrease of up to 50% of energy dissipation is observed in DSM compared to CSM.

Future work lies in exploring the use of linker scripts for function-level control of memory mapping, aiming at better distribution of memory traffic across physical memories, which will further reduce the consumed energy. A generalization of our approach by distributing both data and instruction caches is another important perspective. This involves adequate cache coherence mechanisms.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under the Mont-Blanc Project: www.montblanc-project.eu, grant agreement no 288777.

REFERENCES

[1] P. Greenhalgh, *Big.little processing with arm cortex-a15 & cortex-a7*, ARM White Paper, Tech. Rep., 2011. [Online]. Available: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf
[2] Kalray, *MPPA: the supercomputing on a chip solution*, http://www.kalrayinc.com/kalray/products, 2014.
[3] Plurality Ltd., *The HyperCore Processor*, http://www.plurality.com/hypercore.html.
[4] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy and D. Dutoit, *Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications*, Design Automation Conference, June 2012.
[5] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo: Making the case for an arm-based hpc system," *Future Generation Computer Systems*, vol. 36, no. 0, pp. 322 – 334, 2014.
[6] C. Ebert and C. Jones, *Embedded software: facts, figures, and future*, IEEE Computer, vol. 42, no. 4, pp. 42 – 53, 2009.
[7] S. Comella-Dorda, K. Wallnau, R.C. Seacord and J. Robert, *A survey of legacy system modernization approaches*, (Num. CMU/SEI-2000-TN-003). Carnegie-Mellon University, Pittsburgh, PA - USA, 2000. [Online]. Available: http://www.sei.cmu.edu/reports/00tn003.pdf
[8] J.A. Ambrose, J. Peddersen, S. Parameswaran, A. Labios and Y. Yachide, *SDG2KPN: System Dependency Graph to Function-level KPN generation of Legacy Code for MPSoCs*, Asia and South Pacific Design Automation Conference, pp. 267 – 273, January 2014.
[9] W. Sheng, P. Szymanski, R. Leupers and G. Ascheid, *Software Migration for Parallel Execution on a Multicore Tablet: A Case Study*, Symp. on Embedded Multicore SoCs (MCSoC), pp. 1 – 6, Sept. 2013.
[10] J. Bueno, J. Planas, A. Duran and R.M. Badia, *Productive Programming of GPU Clusters with OmpSs*, International on Parallel & Distributed Processing Symposium, pp. 557 – 568, May 2012.
[11] H. Matsuba and Y. Ishikawa, *OpenMP on the FDSM software distributed shared memory*, Europ. Workshop on OpenMP, Sept. 2003.
[12] J. Zhang, Z. Yu, Z. Yu, K. Zhang, Z. Lu, and A. Jantsch, *Efficient distributed memory management in a multi-core H.264 decoder on FPGA*, Int'l Symposium on System on Chip, pp. 1 – 4, Oct. 2013.
[13] X. Chen, L. Zhonghai, A. Jantsch, C. Shuming, C. Shenggang and G. Huitao, *Reducing Virtual-to- Physical address translation overhead in Distributed Shared Memory based multi-core Network-on-Chips according to data property*, In ACM journal Computers & Electrical Engineering, vol. 39 (2), pp. 596 – 612, February 2013.
[14] Z. Yuang, L. Li, Y. Shengguang, D. Lan, L. Xiaoxiang and G. Minglun, *A scalable distributed memory architecture for Network on Chip*. In Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS08), pp. 1260 – 1263, November 2008.
[15] J. Breitbart, *Dataflow-like synchronization in a PGAS programming model*, In Int'l Parallel and Distributed Processing Symposium, Workshops & PhD Forum, pp. 762 – 769, May 2012.
[16] M.M. Martin, M.D. Hill and D.J. Sorin, *Why on-chip cache coherence is here to stay*. Communications of the ACM, 55(7), pp. 78 – 89, 2012.
[17] R. Garibotti, L. Ost, R. Busseuil, M. Kourouma, C. Adeniyi-Jones, G. Sassatelli and M. Robert, *Simultaneous Multithreading Support in Embedded Distributed Memory MPSoCs*, In 50th ACM/IEEE Design Automation Conference (DAC-13), pp. 1 – 7, June 2013.
[18] F. Moraes, N. Calazans, A. Mello, L. Muller and L. Ost *Hermes: an infrastructure for low area overhead packetswitching networks on chip*, Integration, the VLSI Journal, vol.38(1), pp. 69 – 93, October 2004.
[19] *Plasma most mips i(TM)*, www.opencores.org/project,plasma.
[20] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D Franzon, M. Bucher, S.Basavarajaiah, Oh Julie and R. Jenkal, *FreePDK: An Open-Source Variation-Aware Design Kit*, Int'l Conf. on Microelectronic Systems Education, pp. 173 – 174, June 2007.
[21] Cadence, www.cadence.com/products.
[22] X. Dong, C. Xu, Y. Xie, and N.P. Jouppi, *NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, no. 7, pp. 994 – 1007, July 2012.
[23] W. Quan and A.D. Pimentel, *A system-level simulation framework for evaluating task migration in MPSoCs*, In International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 1 – 9, October 2014.
[24] J. Hu and R. Marculescu, *Energy-aware mapping for tile-based NoC architectures under performance constraints*, In Asia and South Pacific Design Automation Conference, pp. 233 – 239, January 2003.

Dr. Rafael Garibotti is currently a Postdoctoral Fellow at Harvard University, USA. He received his Ph.D. and MSc. Degree in Microelectronics, respectively from University of Montpellier II and EMSE, France and his BSc. Degree in Computer Engineering from PUCRS, Brazil. His research activity focuses on programmability and design of parallel architectures, such as accelerators.

Anastasiia Butko is currently a PhD candidate at LIRMM in France. She works on adaptive multiprocessor architectures for embedded systems. She received her MSc. degree in Microelectronics from Université de Montpellier II in France, and MSc and BSc degrees in Design of Electronic Digital Equipment from NTUU "KPI" in Ukraine.

Dr. Luciano Ost is a lecturer in embedded systems at the University of Leicester. Ost received his PhD degree in computer science from PUCRS, Brazil in 2010. After the completion of his doctorate degree, he worked as a research assistant and then as associate professor at the University of Montpellier II/LIRMM in France, until joining the University of Leicester. His research interests include, among others, design of NoC-based multiprocessor systems. He is a member of the IEEE.

Dr. Abdoulaye Gamatié is currently a Research Scientist at CNRS / LIRMM in France. He received his Ph.D. degree in Computer Science in 2004 from Université de Rennes 1 in France. His research activity focuses on the design of energy-efficient multicore/multiprocessor architectures for embedded and high-performance computing. He is the author of a reference book on synchronous programming of embedded applications with Signal language. He co-authored more than 50 articles in refereed international conferences and journals.

Dr. Gilles Sassatelli occupies a senior scientist position at CNRS LIRMM. He is currently the leader of the Adaptive Computing Group composed of 6 permanent staff researchers and 20 Ph.D. candidates. His research work is focused on reconfigurable and multiprocessor architectures for embedded and high performance computing. He has published more than 150 publications in a number of peer-reviewed renowned international conferences and journals.

Chris Adeniyi-Jones works in ARM's corporate Research and Development division. He joined ARM in 1994 initially working on cycle accurate processor models, later moving to the compiler team in the Software Development Division. There he worked on implementing C/C++ language features and specialized in code generation optimized for both code-size and performance. Chris' areas of research include many-core architectures, reconfigurable computing and software power optimization. He is the ARM coordinator for the Mont-Blanc project which is looking at High-Performance Computing based on Energy-efficent embedded technology.