Novel Techniques for Smart Adaptive Multiprocessor SoCs

Luciano Ost, Rafael Garibotti, Gilles Sassatelli, Gabriel Marchesan Almeida, Remi Busseuil, Anastasiia Butko, Michel Robert, Jürgen Becker

Novel Techniques for Smart Adaptive Multiprocessor SoCs

Luciano Ost, *Member, IEEE,* Rafael Garibotti, Gilles Sassatelli, *Member, IEEE,*
Gabriel Marchesan Almeida, Rémi Busseuil, *Member, IEEE,*
Anastasiia Butko, Michel Robert, Jürgen Becker, *Senior Member, IEEE*

Abstract—The growing concerns of power efficiency, silicon reliability and performance scalability motivate research in the area of adaptive embedded systems, i.e. systems endowed with decisional capacity, capable of online decision making so as to meet certain performance criteria. The scope of possible adaptation strategies is subject to the targeted architecture specifics, and may range from simple scenario-driven frequency/voltage scaling to rather complex heuristic-driven algorithm selection. This paper advocates the design of distributed memory homogeneous multiprocessor systems as a suitable template for best exploiting adaptation features, thereby tackling the aforementioned challenges. The proposed solution lies in the combined use of a typical application processor for global orchestration along with such an adaptive multiprocessor core for the handling of data-intensive computation. This paper describes an exploratory homogeneous multiprocessor template designed from the ground up for scalability and adaptation. The proposed contributions aim at increasing architecture efficiency through smart distributed control of architectural parameters such as frequency, and enhanced techniques for load balancing such as task migration and dynamic multithreading.

Index Terms—Adaptive systems, Multithreading, Adaptive Hardware and Systems, NoC-based Multiprocessor

1 Introduction

Driven by the rapidly increasing Non-Recurring Engineering (NRE) and fixed manufacturing costs, the semiconductor industry shows signs of fragmentation driven by specialization. This leads to the emergence of platform-based design in which off-the-shelf SoCs (Systems-on-Chips) are integrated in a multitude a branded products. Popular platforms such as TI OMAP, Qualcomm Snapdragon are such prominent examples in the domain of consumer-market smartphones and tablets. These SoCs, typically referred to as MPSoCs (Multi-processor SoCs) are built around an application processor (AP) running an microkernel, connected to a number of application specific ISP cores such as DSPs and ASIPs, each of which is devoted to a given functionality such as baseband processing, video decoding, etc. Such heterogeneous system architecture is motivated by concerns of cost and energy efficiency. Because of the heterogeneous nature of such systems, inter-processor communications take place by means of exchanged messages, mostly through an external shared memory, and often orchestrated by the AP operating system. Such system template poses severe scalability problems that may be summarized as follows:

1) Reliability and ageing of silicon in below $22nm$ nodes motivate research in the area of resilient architecture design [1]. Most explored solutions rely on the use of spare/redundant processing resources, for either instant fault detection/correction (critical systems) or remapping of functionality on fault detection. Because of being highly heterogeneous the typical MPSoC template hardly enables devising such techniques.
2) As performance and power efficiency scaling cannot solely rely on technology downscaling, online optimization is of utmost importance, for better exploiting available resources thereby avoiding design oversizing. Though typical embedded MPSoCs employ advanced techniques such as semi-distributed power-gating, voltage and frequency scaling, the scope of foreseeable techniques remains limited mostly because of the heterogeneous and centralized nature of such systems.
3) The resulting system complexity induces a number of difficulties from a software point of view, with typically hundreds of IPs for which drivers are either closed-source, or provide bare minimum support for the default given purpose of any given IP. Because of this, the system often remains underexploited: user applications are intended to be executed on the AP and not the application-specific ISPs that are not seen as programmable from a end-

- *L. Ost, R. Garibotti, G. Sassatelli, R. Busseuil, A. Butko and M. Robert are with Laboratory of Informatics, Robotics and Microelectronic of Montpellier (LIRMM)*
34095, Montpellier, France.
E-mail: {ost, garibotti, sassatelli, busseuil, butko, robert}@lirmm.fr

- *G. Marchesan Almeida and Jürgen Becker are with Karlsruhe Institute of Technology (KIT)*
76131 Karlsruhe, Germany.
E-mail: {gabriel.almeida,juergen.becker}@kit.edu

user/application developer perspective[1].

This paper therefore proposes an alternative system organization that offers online optimization opportunities beyond classical techniques, thereby furthering potential benefits. We regard the heterogeneous and centralized nature of MPSoCs as being the main limiting factors towards achieving better scalability. Though the use of an application processor is not questionable, we argue that a subset of domain-specific ISPs can be advantageously replaced by a fully decentralized homogeneous multiprocessor core designed for scalability and adaptation, similar to the major shift observed in GPU architectures that have become homogeneous for achieving a better utilization of processing resources.

The multiprocessor core used in this study and previously proposed by the authors [2], is available under the GNU General Public License at [3]. This core is based on an array of loosely coupled independent processors that communicate through exchanged messages routed in a 2D-mesh NoC. Each processor is equipped with a local memory that stores both application code and a compact microkernel that provides a number of services among which online optimization functionalities, from now on referred to as adaptation[2], which are either executed in a native distributed fashion or in a semi-centralized mode.

- first, based on the proposed architecture template, we demonstrate the benefits of local distributed control through an implementation of feedback control loops for adjusting frequency at processor level.
- second, we propose a novel approach to load balancing named remote execution, which makes for a tradeoff between reactivity and performance compared to task migration.
- third, though message passing is the default programming model for such a distributed memory architecture, we describe a distributed shared-memory (DSM) approach that enables the architecture to operate in multithreading mode using a thread API similar to POSIX threads. This approach allows for exploiting load balancing at intra-task level thereby giving another degree of freedom in the adaptation process; and further paves the way for semi general-purpose application acceleration.

The rest of this paper is organized as follows: Section 2 presents and discusses relevant approaches in the literature, and put these in perspective with the current work. Section 3 then describes the used multiprocessor core hardware and software architecture as well as the programming models. Section 4 is devoted to the first 2 contributions, use of feedback control-loops for distributed control and remote execution. Section 5 presents the proposed multithreading approach, and provides benchmarking results for scalability assessment. Section 6 is a wrap-up of all proposed techniques that are compared from an adaptation perspective on a case-study. Section 7 draws conclusions and proposes future work directions.

1. GPUs are a notable exception, with the growing interest in General-Purpose computing on GPUs (GPGPU) using APIs such as OpenCL. As these architectures provides significant speedups for very regular code only, they are not discussed further here

2. The concept of adaptation (sometimes referred to as self-adaptation in the literature) used throughout this paper is similar to that used in biology: it refers to the process of adapting to time-changing situations, driven by the entity that undergoes the process itself.

2 State of the Art

Adaptation refers to online optimization mechanisms that may take place at several abstraction levels: (i) architecture (*low-level*); and (ii) system (*high-level*). In the first case, because of being dealt with by the hardware itself, quasi-instant decisions may be made at the price of somewhat limited capability of analysis. Typically, decisions are made on threshold crossing and often relate to tuning of architecture parameters such as voltage or frequency. Phenomena that take place at a larger time scale with therefore slower dynamics are handled at system-level, often by the software: in this latter case, finer analysis is foreseeable (i.e. statistical information gathering), and possible adaptation strategies cover techniques such as task migration, route reconfigurations, computation/communication rescheduling etc.

2.1 Adaptation at Architecture Level

Dynamic Voltage and Frequency Scaling (DVFS) techniques have long been used in portable embedded systems in order to reduce the energy consumption and temperature of such systems. Most existing DVFS are defined at design time, where they are typically based on pre-defined profiling analysis (e.g. application data) that attempts to define an optimal DVFS configuration [4][5]. Puschini et al. [6] propose a scalable multi-objective approach based on game theory, which adjusts at run-time the frequency of each processing element (PE) while reducing tile temperature and maintaining the synchronization between application tasks. Due to the dynamic variations in the workload of these systems and the impact on energy consumption, other adaptation techniques such as Proportional-Integral-Derivative (PID) based control have been used to dynamically scale voltage and the frequency of processors [7][8], and recently, of Network-on-chip (NoC) [9][10]. These techniques differ in terms of monitored parameters (e.g. task deadline, temperature) and response times (period necessary to stabilize a new voltage/frequency).

Ogras et al. [9] propose an adaptive control technique for a multiple clock domain NoC, which considers the dynamic workload variation aiming at decreasing the power consumption. Besides, the proposed control technique ensures the operation speed and the frequency limits of each island (clock domain, defined according to a given temperature). The effectiveness of the proposed adaptive control technique was evaluated considering different scenarios (e.g. comparison with a

PID controller) for a MPEG-2 encoder. In [10], a PID controller is used to provide a pre-determined throughput for multiple voltage-frequency/voltage-clock NoC architectures. The proposed PID-based controller sets the voltage and frequency of each NoC island by reserving virtual channels weights (primary control parameter) in order to provide the necessary throughput for different application communications, while saving energy.

In [11] authors show a technique for minimizing the total power consumption of a chip multiprocessor while maintaining a target average throughput. The proposed solution relies on a hierarchical framework, which employs core consolidation, coarse-grain DVFS. The problem of optimally assigning dependent tasks in multiprocessor system has not been addressed in their paper. Adaptability has been explored under a number of aspects in embedded systems, ranging from adaptive modulation used in 3GPP-LTE (3^{rd} Generation Partnership Project Long Term Evolution) standard SDR (Software Defined Radio) [12] to adaptive cores instantiation in dynamically reconfigurable FPGAs [13].

2.2 Adaptation at System Level

Adaptation at system level, here understood as a means for dynamically assigning tasks to nodes so as to optimize system performance, can be done in two ways: (i) by using dynamic task mapping heuristics; and (ii) task migration (TM) for enabling load balancing. Dynamic mapping heuristics are intended to efficiently map at run-time application tasks from a repository to the available processors in the architecture. Several dynamic mapping heuristics have been proposed [14][15][16][17]. Such heuristics aim to satisfy QoS requirements (e.g. [15]), to optimize resources usage (e.g. network contention [18][16]), and to minimize energy consumption (e.g. [18][19][17]). Each of those has its own parameters and cost functions, which can have numerous variations (defined as new heuristics with different properties and strengths). Thus, choosing the optimal dynamic heuristic for a set of applications is not trivial; hence evaluating different performance metrics is time-consuming. Alternatively, TM may employ mapping heuristics that operate on monitoring information (e.g. processor workload, communication latency) in order to better fit current execution scenario. Streichert et al. [20] employ TM to ensure computation correctness of an application by migrating executing tasks from a faulty node to a healthy one. Authors propose an analytical methodology that uses binding to reduce the overhead of the task migration. However, neither architecture details nor task mechanisms are presented in this work. A similar analytical-based approach is presented in [21], which proposes five task migration heuristics. These heuristics are employed to remapping tasks onto remaining non-faulty nodes after the identification and isolation of a given fault.

Shen et al. [22] discuss the software and hardware architecture features that are necessary to support TM in heterogeneous MPSoCs. In [23] authors propose a policy that exploits run-time temperature as well as workload information of streaming applications to define suitable run-time thermal migration patterns. In [24], authors present a migration case study for MPSoCs that relies on the μClinux operating system and a check pointing mechanism, which defines possible migration points in the application code. This work was extended in [25], where an OS and middleware infrastructure for task migration for their MPSoC platform is described.

Barcelos et al. [26] explore the use of TM in a NoC-based system (high level model) that comprises both shared and distributed memory organizations, aiming to decrease the energy consumption when transferring the task code. The distance between the nodes involved in the code migration is used to define, at runtime, the memory organization that will be used. In sum, this section presented related work on adaptive techniques at both architecture and system level: (i) adaptation at architecture level (dynamic voltage and frequency scaling), (ii) adaptation at system level (a) dynamic task mapping heuristics; b) task migration in both shared memory and distributed memory architectures. Our approach differs from the existing in the following aspects:

- Several contributions have been presented in these two levels but none of them propose an adaptive multiprocessor combining all three features. Our approach employs task mapping mechanism at two levels: by means of using dynamic mapping heuristics in order to provide a better initial state to the system, and once tasks are mapped into the platform task migration mechanism enables achieving load balancing among different PEs in the architecture.
- To the best of our knowledge the used embedded multiprocessor core is the only distributed memory system, with RTL implementation, that supports task migration without any help of shared memory; which makes the template scalable.

This paper puts focus on actuation mechanisms rather than decision making algorithms and extends from previous work in three key directions: (1) a novel purely distributed adaptation technique based on PID controllers is proposed and benchmarked in term of performance and power efficiency; (2) novel load-balancing schemes that feature faster reactivity for load balancing compared to task migration are proposed and compared with the previous work; and (3) finally, a distributed shared memory implementation that allows for finer grain load balancing is described and benchmarked.

3 Proposed Architecture Template

This section describes the architecture of OpenScale [2], i.e. an open-source RTL multiprocessor core designed for scalability and adaptation. In order to enable scaling this architecture to any arbitrary size, it makes use of distributed memories so as to avoid usual shared memory bottlenecks. Advanced adaptation features such as task

migration are enabled thanks to a custom microkernel capable of handling complex procedures such as code migrations across distributed physical memories, etc.

3.1 Hardware Architecture

The adopted architecture is a homogeneous message-passing NoC-based multiprocessor core with distributed memory. 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 [27]. These components are interconnected via a Wishbone v4 open-source bus. The communication between the node and the NoC router is implemented in the NI (network interface) which performs low-level packetization/depacketization.

The adopted NoC relies on a mesh of tiny routers based on HERMES infrastructure [28]. The NoC employs packet switching of wormhole type: the incoming and outgoing ports used to route a packet are locked during the entire packets transfer. The routing algorithm is of XY type (Hamiltonian routing) that allows deterministic routing. Each router has one incoming FIFO buffer per port. The depth of FIFOs can be chosen according to the desired communication performance. Figure 1 depicts the platform along with the 2 supported programming models that are described in 3.3, message passing and multithreading through run-time vSMP (virtual Symmetric Multi Processor) cluster definition.

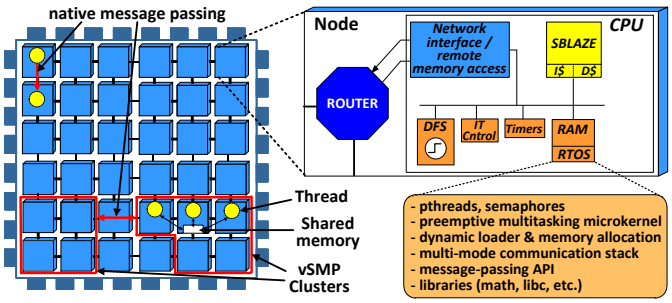


Fig. 1. OpenScale Platform

3.2 Software Architecture

The microkernel used in our platform was originally developed by [29] and extensively modified in order to implement and provide new services needed for proposing new adaptive mechanisms for homogeneous MPSoC architectures. This choice was mostly motivated by the small memory footprint (around 150KB) of this microkernel given the provided features.

Applications are modelled as tasks that access hardware resources through an API. A function call then generates a software exception, handled by the operating system *Exception Manager*. Additionally, the *Scheduler* communicates with the memory and task management modules, and exchanges messages with the decision-making mechanisms. The *Routing Table Manager* keeps track of tasks' location storing the information in a data structure called routing table. Whenever a task has to communicate with another, the operating system performs a lookup in the local routing table. Besides, *Adaptation Mechanisms* are implemented in order to provide load balancing as well as optimize platforms resources utilization, namely by scaling processor frequency according to application requirements.

One of the objectives of this work is to enable dynamic load balancing which implies the capability to migrate running tasks from processor to processor. Migrating tasks usually implies: (1) to dynamically load corresponding code in memory and schedule a new process; (2) to restore the context of the task that has been migrated. A possible approach relies on resolving all references of a given code at load-time; such a feature is partly supported in the ELF (Executable and Linkable Format) [30] which lists the dynamic symbols of the code and enables the operating system loader to link the code for the newly decided memory location. Such mechanisms heavily rely on the presence of a hardware MMU (Memory Management Unit) for address virtualization and are memory consuming which clearly puts this solution out of the scope of the approach.

Another solution for enabling the loading of processes without such mechanisms relies on a feature that is partly supported by the GCC compiler that enables to emit relocatable code (PIC - Position Independent Code). This feature generally used for shared libraries generates only relative jumps and accesses data locations and functions using a Global Offset Table (GOT) that is embedded into the generated ELF file. A specific post-processing tool which operates on this format was used for reconstructing a completely relocatable executable. Experiments performed in [31] show that both memory and performance overheads remain under 5% for this solution which is clearly acceptable.

3.3 Programming Model

3.3.1 Message Passing Interface

Programming takes place using a message-passing API. Hence, tasks are hosted on nodes which provide through their operating system communication primitives that enable data transfers between communicating tasks. The proposed model uses two P2P communication primitives, $MPI\_Send()$ and $MPI\_Receive()$, based on synchronous message passing interface (MPI). $MPI\_Receive()$ blocks the task until the data is available while $MPI\_Send()$ is non-blocking, unless no buffer space is available.

Each communicating task may have one or multiple software FIFOs. In case a FIFO runs full, the operating system resizes it according to the available memory. This process is handled by a flow control mechanism

implemented as a service in the OS. This strategy allows saving resources once memory is dynamically allocated/deallocated according to communication needs.

3.3.2 Shared Memory/Multithreading

The work presented above relies on the assumption of economically viability of numerous on-chip distributed memories. While this may prove doable in the near future because of advances in the area of dense non-volatile emerging technology memories [32] such as Magnetic RAMs (MRAMs) and Phase-Change Memories (PCM), devising an approach that makes it possible to share memory would further provide the distinct advantage of opening the way for multithreading, which is by far the most popular parallel programming model in the area of general purpose computing. We here present a low-overhead hardware/software distributed shared memory approach that makes such distributed-memory architecture multithreading-capable. The proposed solution has been implemented into the multiprocessor core through developing a POSIX-like thread API. This approach efficiently draws strengths from the on-chip distributed memory nature of the original architecture template that therefore retains its intrinsic power-efficiency, and further opens the way to exposing the multithreading capabilities of that component as a general purpose accelerator.

3.4 Benchmarking Application Kernels

Four application kernels were employed for benchmarking both adaptation techniques and multithreading extensions. These application kernels are: MJPEG (Motion JPEG) video decoder, Smith Waterman (used to find similar regions between two DNA sequences [33]), LU (matrix factorization) and FFT (Fast Fourrier Transform). For each of these compute kernels, both a message passing (making use of the above-mentioned primitives) and a multithreaded version were developed. Contrary to MPI implementation in which communications are explicit, Pthread-based implementations rely on threads concurrently accessing shared variables residing in a shared memory assumed coherent. For instance, MJPEG threads process entire images, whereas Smith-Waterman implementation relies on worker threads that run sequence alignment algorithm on different data sets. Similarly, LU factorizes a matrix as the product of a lower and an upper triangular matrix and each thread processes different data sets. In turn, FFT is made of a number of threads, each of which performs a number of butterfly operations for minimizing data transfers.

Table 1 gives benchmark-specific figures expressed in terms of single-thread processing time, computation/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 over 13 millions clock cycles ($26ms$ at 500MHz). The second important factor 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. This figure gives an overview of the importance of instruction loading with respects to data loading. Hence, an application having a high computation to data loading ratio would be compute-dominated, therefore with proportionally less data accesses. Note that Smith Waterman has the higher computation to data ratio, when compared to the others.

TABLE 1
Performance Information of Adopted Applications

Application Name	Single Thread Execution Time (16KB of Cache)	Computation to Data Ratio	Thread Size (Instructions)
MJPEG	$5,3$ Mega cycles	5.34	52kB
SmithWaterman	$13,2$ Mega cycles	7.08	3.8kB
LU	$15,5$ Mega cycles	2.48	2.4kB
FFT	$1,9$ Mega cycles	4.18	5kB

The fourth column in Table 1 shows the thread code size of each application. A thread with large code size would potentially lead to more instruction cache misses, since it would likely not fit entirely inside the cache.

4 Novel Adaptive Techniques

Adaptation draws strengths from the ability to handle unpredictable scenarios: given the large variety of possible use cases that typical MPSoC platforms must support and the resulting workload variability, offline approaches are no longer sufficient because they do not allow coping with time changing workloads. This section presents three adaptive techniques implemented in the OpenScale platform, previously introduced in Section 3. We regard the system as having a number of steady states (i.e. task mappings), going from one such state to another occurs whenever a system-level adaptation technique (such as a task migration) takes place, i.e. a transient phase between two steady states. The two levels at which adaptation is implemented are:

- architecture-level: adaptation at this level addresses optimization when the system operates in a steady state: we here assume soft real-time applications therefore a target performance-level has to be met while minimizing power.
- system-level: adaptation at this level triggers remappings in the architecture, resulting into switching from one steady state to another. In that precise case the system operates in best effort mode, attempting at reaching another steady state in the shortest possible time.

4.1 Architecture Level Optimization

As previously mentioned, OpenScale runs an microkernel that may generate multiple perturbations due to the complexity of managing all the provided services such as dynamic task loading, message passing protocol, task migration, frequency scaling, among others. This is made further worse by the data dependent nature of some algorithms for which workload may greatly vary depending on input data, such as video decoding applications. The distributed frequency scaling (DFS) mechanism must be efficient enough for coping with such scenarios so that soft-real time application requirements are satisfied. Next section presents a smart DFS strategy that makes it possible to meet real-time application requirements in the presence of perturbations and fluctuating workload.

4.1.1 Dynamic and Distributed Frequency Scaling

A PID controller is a generic control loop feedback mechanism widely used in industrial control systems. It calculates an *error* value as the difference between a measured process variable and a desired *setpoint*. The controller attempts to minimize the error by adjusting the process control inputs. However, for best performance, the PID parameters used must be tuned according to the nature of the process to be regulated. We here propose the use of PID controllers at task-level as pictured in Figure 2: an offline profiling permits extracting reference performance values (such as throughput, inter-arrival time, jitter etc) that are then used as setpoints for each application task PID controller that then continuously sets the frequency accordingly. This approach aims at setting the lowest possible frequency that satisfies timing constraints, so that minimum energy is wasted in idle cycles in each processor. Once our platform is fully parameterizable, it is possible, by executing different experiments, to extract the most appropriate period for executing the DVFS control. This functionality is implemented as a service in the microkernel and is triggered accordingly to the chosen period by means of interrupts.

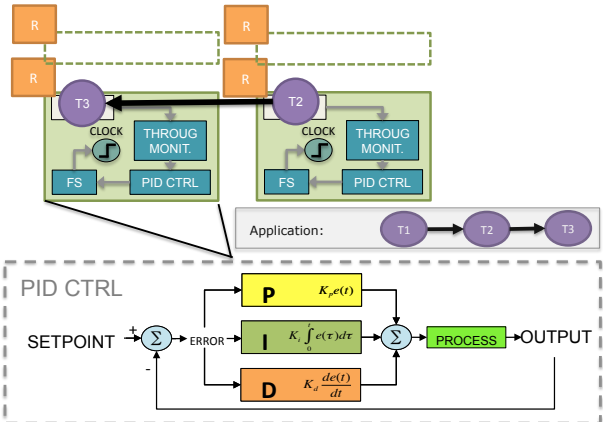


Fig. 2. Distributed PID Controllers

The proportional, integral, and derivative terms are summed to calculate the output of the PID controller. Defining $u(t)$ as the controller output, the final form of the PID algorithm is:

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

1) *Proportional gain, K_p*: larger values typically mean faster response since the larger the error, the larger the proportional term compensation.
2) *Integral gain, K_i:* larger values imply steady state errors are eliminated more quickly.
3) *Derivative gain, K_d:* larger values decrease overshoot, but slow down transient response and may lead to instability due to signal noise amplification in the differentiation of the error.

Each running application is composed of one or multiple tasks monitored by a system thread responsible for calculating applications performance in a *non-intrusive mode*. This information is passed on to the PID controller which scales frequency up or down whenever a deviation is observed from the setpoint value. The strategy consists in deciding controllers parameters on a task basis. To this purpose, a simulation of the MPSoC system is executed in order to obtain the system step response, here defined as the obtained application throughput under frequency change. Based on this high-level model, a number of different configurations of controllers can be explored, each of which exhibits different features such as speed, overshoot, static error, etc. In this scenario, the values of P, I and D have been chosen aiming to increase the reactivity of the system. The system is modeled in Matlab and the different values for P, I, and D are set.

4.1.2 Experimental Protocol and Results

The following scenario is based on an audio/video decoding application which includes ADPCM (Adaptive Differential Pulse-code Modulation) decoder, MJPEG and FIR (Finite Impulse Response). MJPEG video decoder application is composed of five tasks running on a processor and a different application (a synthetic task ($P2$) implemented in such way that it requires a considerable amount of processing power) is migrated to the processor, disturbing the MJPEG application performance. Three different perturbation scenarios ($P1, P2$ and $P3$) are created based on different perturbation factors defined respectively as follows: (i) low, (ii) medium and (iii) high, represented in Table 2. ATBP represents the Average Throughput Before Perturbation while ATAP represents the Average Throughput After Perturbation.

The following scenario depicts the system response under perturbation condition ($P3$). Application performance is analyzed for two approaches: (i) no adaptation is done and processor frequency is constant and (ii) processor frequency is controlled by a PID. The perturbation is configured as follows: (1) perturbation start time:

TABLE 2

Different Perturbation Scenarios

Case	Perturb. Level	Proc. Freq. (*MHz*)	ATBP (*KB/s*)	ATAP (*KB/s*)	Impact Factor
$P1$	low	425	1,050	610	42%
$P2$	$medium$	425	1,050	540	49%
$P3$	$high$	425	1,050	440	58%

$300ms$; (2) perturbation period: $700ms$; (3) application performance impact factor: 58%.

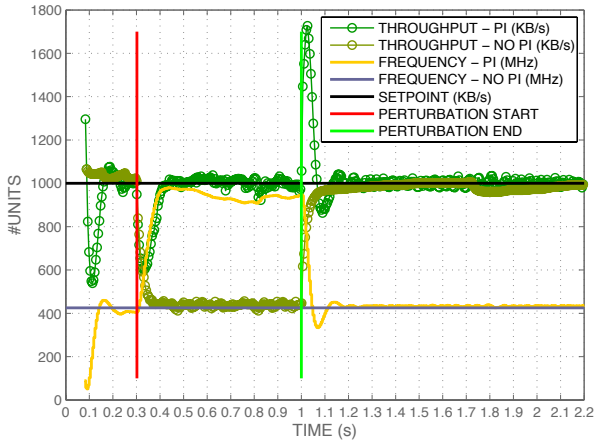


Fig. 3. Perturbation Representation - PI *vs* No PI

Figure 3 shows throughput evolution for both fixed-frequency and PI-driven adaptation. Without PI controller, task throughput is reduced to approximately 440kB/s during the perturbation period. Considering a soft real-time video decoding application where performance constraints must be met so as to ensure constant frame rate, this scenario could result is freezes/frame skips. When using a PI controller, it takes only $100ms$ to restore initial *setpoint*, that represents the minimal performance requirements of a given application. Shortly after perturbation end a peak in task throughput is observed due to the fact the processor is running at high frequency in order to compensate the perturbation occuring this far. The PI controller gradually lowers the frequency, throughput goes back down to *setpoint* which results in power savings.

Table 3 presents the power and energy consumption for three different perturbation scenarios. It can be observed that power and energy consumption are significantly reduced when using PI- and PID-based controllers. In $S1$, the energy consumption is reduced by 32% while power consumption is reduced by 42% in $S3$. It is important to observe that the energy/power consumption saving can be much higher in scenarios with longer perturbation periods and where impact factor of perturbations over applications is higher. It is further import to note that such an approach that promotes continous and frequent frequency scaling may not be optimal depending on the targeted platform, because of

TABLE 3

Node Power and Energy Consumption Representation for Three Different Scenarios Using $65nm$ Technology

Case	Version	Dynamic Power (mW)	Total Power (mW)	Energy (mJ)	Time (s)
$S1$	No PID	295.50	295.64	802.10	2.56
	PI	206.90	207.04	545.10	
	PID	207.10	207.24	545.70	
$S2$	No PID	333.00	333.14	903.80	2.56
	PI	217.70	217.84	573.70	
	PID	217.90	218.04	574.10	
$S3$	No PID	407.90	408.04	1,107.10	2.56
	PI	236.00	236.14	620.70	
	PID	242.90	243.04	638.70	

constraints on selectables voltage / frequency couples for instance. As being implemented in form of a service in the microkernel, easy tuning of actuation frequency, frequency range, number of selectable frequencies or even use of hysteresis thresholds is possible to best fit the platform / application / technology specifics.

4.2 System Level Optimizations

As mentioned previously, system-level adaptation techniques are seen as transient phases between steady states. Because of often incurring penalty on application performance, the platform runs those technique in best effort mode so as to resume normal application execution in the shortest possible time. Benchmarking results for the two techniques proposed in this Section are presented and discussed in Section 6.

4.2.1 Task Migration with Redirection (TMR)

Task migration (TM) mechanisms are implemented by means of tight hardware/software interaction that handle code migration from source node local memory to destination node local memory. The basic mechanism relies on interrupting data transmission between migrating task and all of its predecessors that buffer produced data until completion of migration procedure: handshaking messages are exchanged between predecessor(s) node(s), source node and destination node. Once migration of code across physical memories of source and destination nodes is complete, data transmission is resumed starting with buffered packets.

The migration time very much depends on node processor workload, NoC traffic; and task size. However the latency is typically in the order of $10ms$ to $20ms$ at 500MHz frequency. This therefore gives an assessment of the migration cost; overall migrations shall not be triggered more that 10 times per second at that frequency otherwise performance penalty may become significant.

Task migration with redirection (TMR) mechanism aims at speeding up migration and operates local buffering on the source node rather than the predecessor node,

this allows for i) asynchronous update of predecessor node(s) routing table(s) resulting in data buffered on both the source node and destination node and ii) immediate redirection of buffered packets right after code has been tranfered.

4.2.2 Remote Execution (RE)

Different from TM and TMR techniques, in the RE mechanism the task is remotely executed in the destination node, using a remote memory access protocol implemented in the NoC. The remote execution protocol is illustrated in Figure 4. First processor in node A interrupts $t1$ execution (step 1) and then sends an execution request with the current task state (step 2). Once the chosen node B granted its request, $t1$ is executed in processor (node B) by fetching its code from the remote processor in node A (step 3). To improve execution efficiency, instructions fetched remotely are cached in the $L1$ instruction cache of processor (node B).

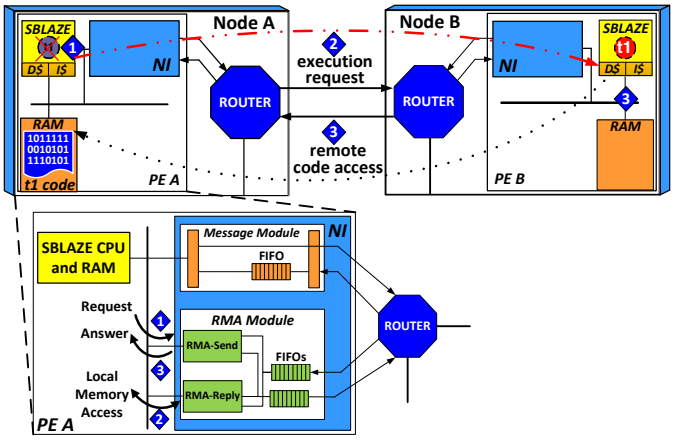


Fig. 4. Remote Task Execution Protocol

It is important to note that the programming philosophy of the platform here remains message passing and not shared memory. Hence, only few data (static data) are fetched from remote memory during a remote execution, with the main data flow coming from messages. In this configuration, data remain uncached during a remote execution. This strategy was taken considering that we have on-chip memory, with memory access latency few orders of magnitude less than standard shared memory architecture using off-chip memory. Furthermore, using uncached data avoids the use of complex cache coherency mechanisms between data cache and remote memories. Instructions, however, are entirely cached.

In order to support this protocol, a hardware module called remote memory access (RMA) was implemented. The RMA is composed of an RMA-Send and an RMA-Reply, which enable distant memory access through the NoC. Two asynchronous FIFOs are employed to guarantee the communication between both modules and the NoC, as shown Figure 4. The purpose of the RMA-Send is to handle memory requests from the local CPU (step 1 and step 3). In case of write request, the RMA-Send component sends the data that must be stored in task node (e.g. in Figure 4 node B transfers the resulting data of execution of $t1$ to the node A). In case of a read request, the RMA-Send component requests the desired data to the remote node (e.g. in Figure 4 node B requesting instruction code to the node A). In turn, the RMA-Reply module answers the request coming from the RMA-Send (NoC side - step 2). In case the incoming request is granted the data is stored in the memory. In a read request, the data is read from the memory, packetized and sent to the requesting node.

The overall memory access performance depends on two main factors. First, as our architecture provides only $L1$ cache, the cache-miss rate is particularly important, as it determines the number of remote memory accesses, hence the traffic. Therefore remote memory access performance depends on: (i) cache size, (ii) cache policy, and (iii) number of remote memory accesses resulting from application.

5 Simultaneous Multithreading

OpenScale being natively a distributed memory system, message passing is the default programming model, and task migration the natural means for achieving load balancing. In order to both offer an alternative programming model and enhance adaptive load balancing, we developed a distributed shared memory approach (DSM). In this context, the in-house microkernel described in Section 3.2 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 5.

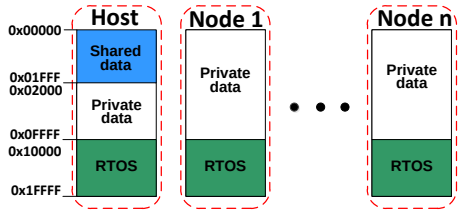


Fig. 5. Distributed Shared Memory node organization

Contrary to message passing in which communications are explicit, shared memory / thread-based parallelism relies on threads concurrently accessing shared variables residing in a shared memory assumed coherent. Synchronizations are handled by means of deciding a specific memory consistency model and an API such as the POSIX threads standard that offers a range of primitives (mutex, semaphores, fork/join, etc). Implementing a coherent memory architecture proves challenging in case several physical memories are used: in such configurations complex cache coherence mechanisms are required, machines of this type are usually referred

to as ccNUMA (cache-coherent Non-Uniform Memory Architecture).

The proposed approach offers the opportunity to decide shared-memory clusters of arbitrary size and geometry at run-time: nodes belonging to the clusters go into bonding mode and share part of the host node memory (one per cluster). A POSIX-like thread API implementation along with DSM hardware support is presented in this section and benchmarked.

5.1 Thread API and memory consistency model

The used thread API is similar to POSIX and has functions belonging to 3 categories: thread creation, mutexes and barriers. This implementation relies on a relaxed consistency memory model in which memory is guaranteed coherent at synchronization points, i.e. whenever an API function call is made. In order to further lower performance overhead related to memory consistency, the API requires shared data be explicitly flagged as such, which accounts for the only difference in function prototype with POSIX thread API, all other being otherwise exactly similar. A multithreaded task in our implementation uses the message passing API for communication with other tasks, but creates worker threads on remote processor nodes. During thread creation, data caches are flushed on the caller side, and invalidated on the executer side. During mutex lock, cache lines that possibly contain shared data that can be accessed between the locking and unlocking are invalidated. During mutex unlock, those same lines are flushed. During barrier calls, the shared memory is also flushed and invalidated. This memory consistency model limits the cache coherence mechanism to the bare minimum thereby simplifying software and hardware support, and features limited performance overhead: invalidation and flush of a given cache line occurs only if cache line tag corresponds to the address specified by the instruction. This condition avoids unnecessary cache flushes/invalidations of cache lines containing unrelated data.

5.2 Multithreading Hardware Support

The RMA has been purposely modified compared to that used in remote execution, so as to better support specific memory consistency operations. It is composed of an RMA-Send and an RMA-Reply, which enable distant memory access through the NoC. Two asynchronous FIFOs are employed to guarantee the 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 the execution of the thread as soon as the cache line is received (iii). This protocol has a latency of 182 clock cycles at zero NoC load, from the cache line request to the remote thread execution, as illustrated in Figure 6.

The maximum end-to-end bandwidth of 90MB/s at 500MHz in this implementation. Note that the host RMA is kept busy during a fraction of that time (64 clock cycles), a maximum theoretical bandwidth of 250MB/s exists when several requests are aliased.

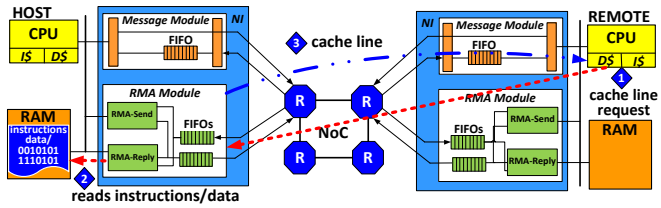


Fig. 6. Cache Miss RMA Protocol

5.3 Experiments and Results

The platform is configured as follows:

- 3×3 processor array, NoC with 4 positions input buffers and 32 bits channel width.
- processor cache size was set to 4, 8 and 16kB, 8 words per lines.
- 500MHz frequency for both processor nodes and NoC routers.
- CPU: hardware multiplier, divider and barrel shifter.

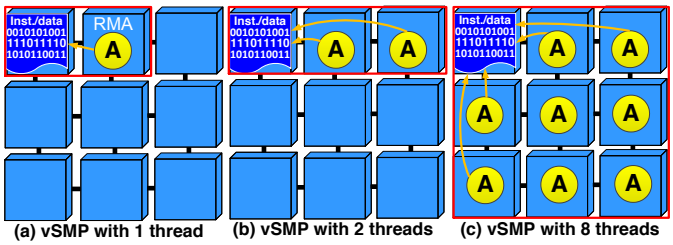


Fig. 7. Adopted Scenario

Figure 7 shows the thread mapping used for experiments. In those, shared data reside in the top-left processor node, and are accessed by threads running in other nodes. This mapping results in less bandwidth available, as only two NoC ports are available. This scenario was used on purpose so as to enable identifying possible limitations/bottlenecks of the proposed system. For the sake of simplicity, Figure 7 (c) does not illustrate all remote memory accesses. All results were gathered on a synthesizable RTL VHDL description of this accelerator. 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 limitations).

5.3.1 Reference Platforms

The GEM5 Simulator framework[34] was used to produce two shared-memory reference platforms: bus- and

NoC-based platforms. GEM5 was chosen because of providing a good tradeoff between simulation speed and accuracy, while modeling a real Realview Platform Baseboard Explore for Cortex-A9 (ARMv7 A-profile ISA).

To create our first reference platform (Figure 8 (a)) it was necessary to modify the CPU model and Linux bootloader supported in GEM5 to enable configurations with more than 4 cores. The bus-based platform, illustrated in Figure 8 (a), is configured as follows: (i) up to 8 ARM Cortex-A9 cores, (ii) CPU running at 500MHz, (iii) Linux Kernel 2.6.38, (iv) 16kB private L1 data and instruction caches, (v) 32bits channel width and (vi) DDR physical memory running at 400MHz.

Figure 8 (b) shows the main architecture of our second reference platform, which comprises: (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) 256MB unified L2 cache and (vii) an interconnection network. Such a large L2 cache size was decided for avoinding any traffic with external DRAM memory during benchmark execution, so as to perform fair comparisons with the proposed architecture template that uses on-chip memories only. The Cache Coherence Protocol that was used is MOESI Hammer (AMD's Hammer protocol used in AMD's Hammer chip [35]). Three interconnection network topologies were use as reference NoC-based models: (i) crossbar - each controller (L1/L2/Directory) is connected to every other controllers via one router (modeling the crossbar), (ii) mesh and (iii) torus. In both Mesh and Torus topologies, each router is connected to a L1 and a L2.

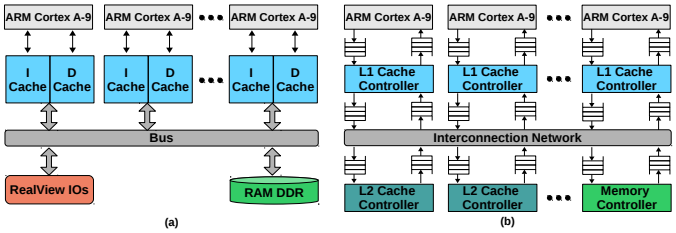


Fig. 8. Reference Platforms

5.3.2 Speedup

The first experiment evaluates the platform scalability considering MJPEG, Smith Waterman, FFT and LU during their execution. In order to maximize speedup, one thread per node was used in all experiments, thereby avoiding performance penalties resulting from context switching. Following results are CPU normalized so as to assess scalability of the template and abstract ISA/microarchitecture specifics: hence, single core performance is shown as same for all architectures. Figure 9 (a) shows how MJPEG performance scales versus the number of cores. Near-linear speedup is observed for the proposed architecture template when cache sizes of 8 and 16kB are employed. Reference NoC-based platforms show better performance when compared to the reference shared bus-based system, which reaches a plateau from 5 cores due to the bus saturation. The same behavior is also observed with the proposed architecture template when using cache sizes of 4kB.

In turn, Figure 9 (b) shows near-perfect linear speedup for the Smith-Waterman application, for both proposed and reference platforms. In turn, a maximum speedup of 6.3 is achieved in the reference shared-memory mesh platform. As being very compute oriented, few data communication occur and thread code fit in the local caches for all tested configurations, resulting in limited cache miss rate. Different from the previous benchmarks the FFT application was employed to explore the shared data protection when multiple threads trigger concurrent accesses to the same shared data/variable (e.g. during a read or write operation). In this situation, synchronization primitives like Mutexes must be used to allow exclusive thread data access and Barrier to coordinate the parallel execution of threads, during their synchronization. Thus, a Mutex variable is used to allow each thread to use a single variable in shared memory to identify them before starting their execution process.

Figure 9 (c) shows the performance scalability of FFT application, regarding the adopted scenario (Figure 7). Due to the sequential synchronizations, the parallelization capability of FFT is low. Thus, using our platform with a 16kB cache configuration, the application can only achieve a speedup of 4.6. However, such results show the better performance when compared to the other reference systems, which achieve a plateau from 5 cores. The LU scenario presented the worst scalability results, as shown in Figure 9 (d). Excluding the proposed distributed memory architecture with cache size of 16kB that does not reach the saturation (application speedup of 4.1), others architectures reached a plateau from 7 cores.

To further platform explorations, we generated an 4x4 scenario, varying the host position. Three scenarios were explored:

- Host placed at 0x0: which comprises, in cartesian addressing top-bottom corner positions (00, 03, 30 and 33). In this case, in the worst case 3 threads will communicate using the east port and the other 12 threads will communicate through the south port, making this the bottleneck.
- Host placed at 1x1: comprising central mesh positions (11, 12, 21 and 22), which reduce the south port communication from 12 to 8 threads, whille decrising NPU-Host distance to 4 hops.
- Host placed at 1x0: which comprises the remaining positions. Due to the adopted XY routing algorithm, 12 threads will communicate through the south port, while one and two threads through west and east ports, respectively.

As displayed in Figure 9 (e), the total gain using 15 threads is about 3% by changing only the host position,

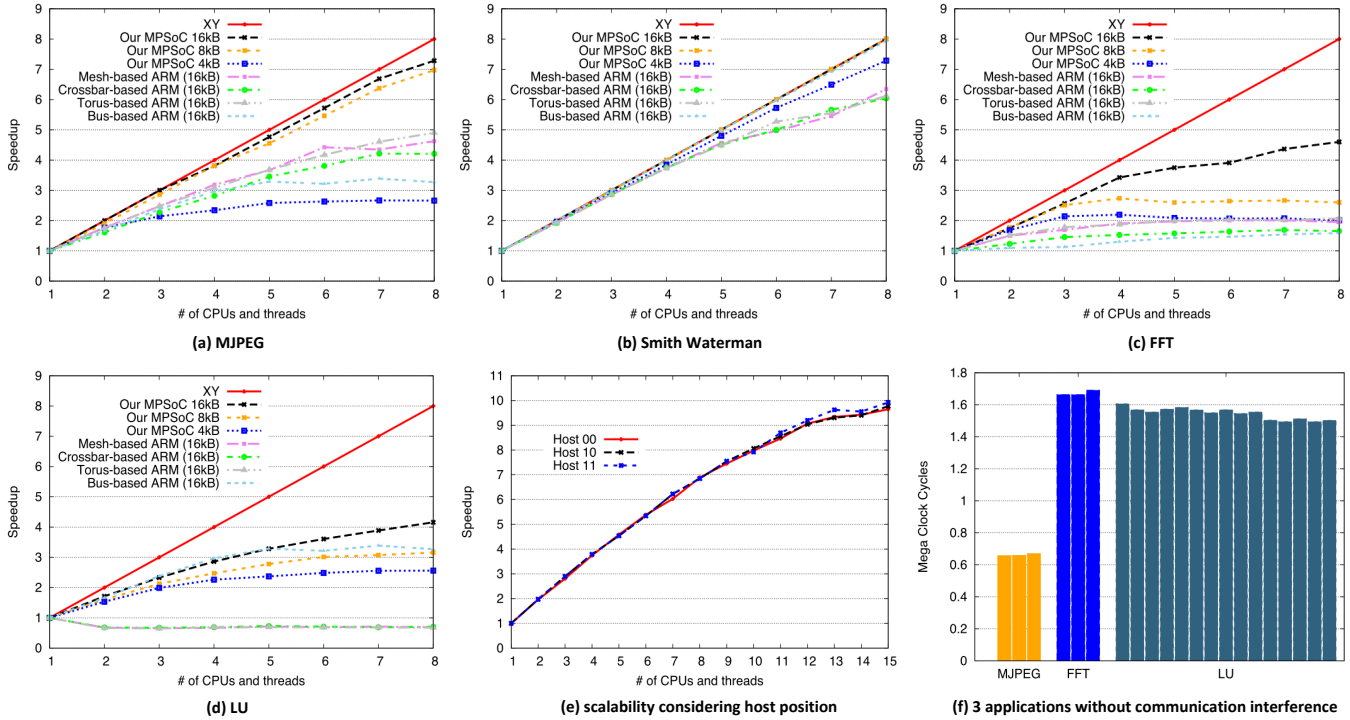


Fig. 9. Architecture scalability, for cache sizes of 4kB, 8kB and 16 kB (a)(b)(c)(d); scalability scenario regarding host position in a 4 x 4 configuration (e); cluster exploration considering MJPEG (3 threads), FFT (3 threads) and LU (15 threads), executing simultaneously (f), where each bar represents the execution time of one thread

this can be explained by the fact that the MJPEG application is scalable and the NoC was not saturated when a 16kB cache size is employed, as will be detailed in Section 5.3.3. The next experiment (Figure 9 (f)) explores the use of multiple vSMP clusters. This scenarios comprises a 6x4 platform organized in 3 virtual clusters, where: two 2x2 clusters are used to execute FFT and MJPEG while a 4x4 cluster executes the LU application. Jitter in thread execution time remains below 5

5.3.3 RMA and NoC Throughput

Figure 10 shows the average bandwidth usage 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 a role in the NoC usage, as explained in Section 2.2. It can be observed that south link is unused for 1 and 2 threads, because of the decided mapping. Most data flows through the east host NoC link, due to the used mapping. For 8 and 16kB cache sizes the bandwidth grows almost linearly versus the number of threads. 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 9 (a) because of the RMA saturation.

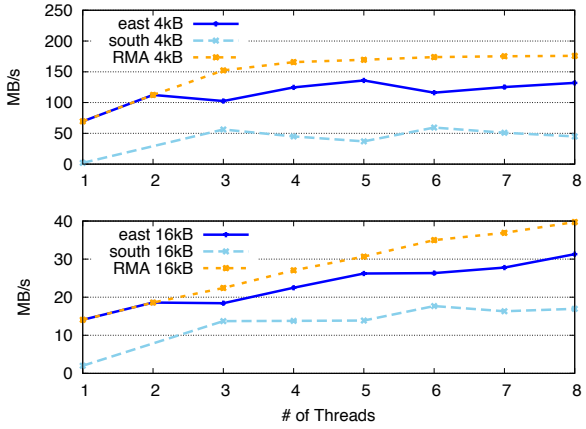


Fig. 10. Average bandwidth usage for MJPEG

6 Comparison of Proposed Techniques

6.1 Performance Comparison

Multithreading as presented in Section 5 allows for thread-level parallelism, hence at a finer grain compared to task-level parallelism. Beyond the obvious gains in programmability, this permits lifting some performance bottlenecks should a given task be critical in an application processing pipeline. In order to fairly assess the performance and drawbacks of all proposed techniques, a case study 3x3 processor system capable of running

any of the proposed techniques has been built . This setup was experimented on a MJPEG processing pipeline comprising 4 tasks, with a defavorable initial mapping in which IVLC and IQuant, the 2 most compute intensive task are executed on a single processor node in a time-sliced mode. Six simulation runs, depicted in Figure 11, are presented accounting for TM, TMR, RE and multithreaded execution from 1 to 3 worker threads, alongside a main thread which handles synchronizations and acts as a wrapper.

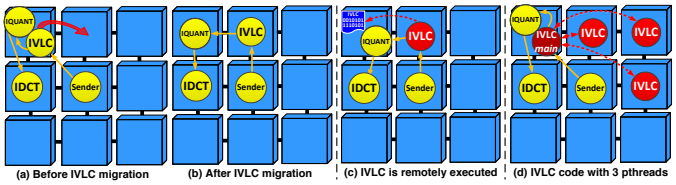


Fig. 11. Adopted Scenarios

Figure 12 (a) shows the throughput obtained for each technique triggered at the same time in all 6 runs. The throughput in all scenarios is maintained up until the migration trigger point. TM incurs a significant penalty due to the required time to transfer task code, interrupting and later restoring data transfers. During this 10ms period TM throughput drops to about half of initial value; while TMR shows a greater performance penalty lasting much less, therefore completing migration in about 4ms thanks to the data redirection strategy. RE negates observed performance penalty and instantly triggers execution on the remote node. Post-migration performance is however less than TM due to the latency incurred by remote instruction fetching. Concerning multithreaded implementations, single thread performance is similar to RE while 2 and 3 thread implementations outperform all other techniques, at the expense of respectively one and two more processor nodes used. Figure 12 (b) plots the buffer usage when the MJPEG pipeline is connected to a video DAC that consumes packets at regular time intervals. Buffer occupation drops during migration as a function of the used technique. While TM requires over 20 buffered packets for avoiding starvation, much less is required for all other techniques. Buffer occupation then increases after migration as higher throughput is achieved, plot crossing give an assessment of the break-even point and therefore the payback time.

Multithreading is here regarded as a complementary technique to TM, TMR and RE: in the example above should a higher level of performance be required for the video decoding, multithreading proposes an alternative to repartitionning of the entire application into tasks which would otherwise be needed. As multithreaded execution performance is sensitive to available bandwidth on the NoC as presented in Figure 10, several measures may be taken to limit performance penalty, such as controlling the amount of through-traffic in the vSMP cluster.

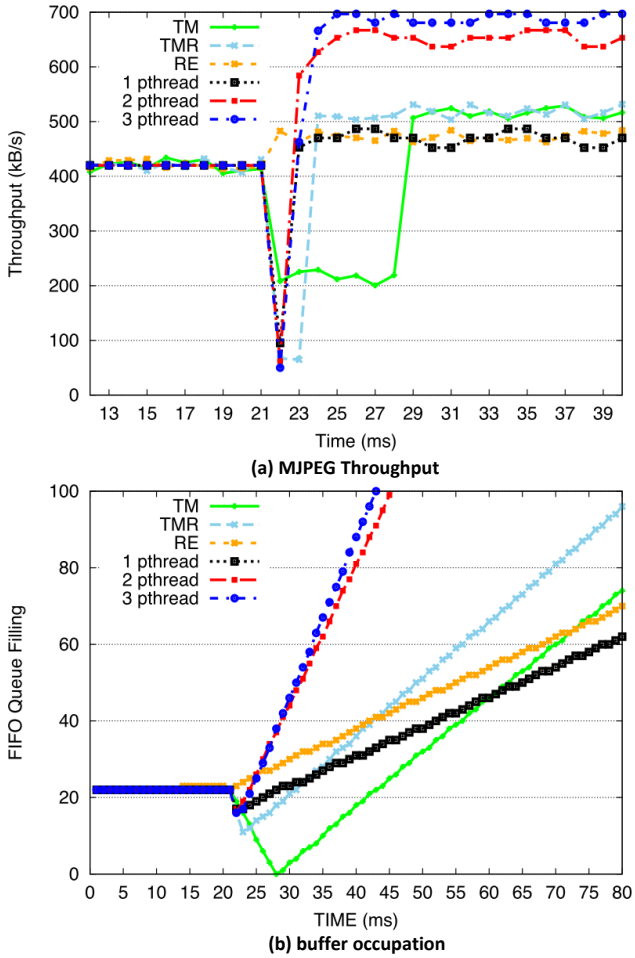


Fig. 12. Performance Comparison for the 4 techniques

6.2 Proposed Architecture Area Overhead

Area results were obtained for both FPGA and ASIC targets targeting 40nm CMOS process, with the following memory sizes: (i) 64kB, (ii) 128kB and (iii) 256kB. Target processor architecture has FPU, optional IO (e.g. debug, UART) were removed from synthesis.

TABLE 4
PE area evaluated at 40nm CMOS Technology

Memory Size	Low-Power core with FP	Low-Power core with FP + RMA	Area Overhead
64kB	$0,2803mm^2$	$0,3240mm^2$	$15,59\%$
128kB	$0,4392mm^2$	$0,4829mm^2$	$9,94\%$
256kB	$0,7650mm^2$	$0,8088mm^2$	$5,72\%$

Table 4 shows area overhead related to multithreading range from $5,72\%$ to $15,59\%$, using respectively 256kB and 64kB of RAM. An additional analysis was performed using a Xilinx Spartan-3 FPGA, RMA implementation incurs an resource overhead ranging from 7% to 15% depending on the size of shared memory.

7 Conclusion

While most platform SoCs are today heterogeneous, our approach follows a different route and advocates the design of adaptive and scalable multiprocessor cores, sitting next to a traditional application processor, that could compete performance and power-wise with ISPs to their ability to adapt to time-changing scenarios. With the underlying motivation of architecture scalability, this paper demonstrated on the basis of a scalable and synthesizable HW/SW multiprocessor template, several novel purely distributed adaptation techniques that leverage the existing techniques for such architectures:

i) distributed PID-based control, experimented on frequency scaling has proven effective both in the handling of transient perturbations and power-wise. This approach could be extended both in term of algorithm (use of Kalman filters) or control scheme, that could be designed hierarchical such as those used in most complex control automation processes.
ii) as an alternative to task migration in distributed memory systems, two techniques are presented: task migration with redirection and remote execution, the latter can be regarded as a technique that makes load balancing in distributed memory systems viable, because of incurring negligible latencies, similar to those in shared-memory multicore systems.
iii) in order to widen the scope of load-balancing opportunities a low area overhead technique for multithreading supports based on distributed shared memory is proposed: it efficiently draws strengths from the scalable, distributed memory and onchip features of our architecture: the low memory access latencies make it possible to literally dynamically grow a large number or threads for achieving maximum performance.

Summarizing the above discussions, we believe that all these techniques combined together make for a flexible template with a panel of techniques that permits exploiting different application profiles and use cases. Our current work lies in exploring the opportunities offered by this technique for harnessing the processing power of such dense multiprocessor cores from the application processor for quasi-general purpose application acceleration contrary to typical heterogeneous platform SoCs. APIs such as OpenMP and OpenCL are promising alternatives that will certainly render such solutions more attractive in the future.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10 – 16, nov.-dec. 2005.

[2] R. Busseuil and et al., "Open-scale: A scalable, open-source noc-based mpsoc for design space exploration," *Reconfigurable Computing and FPGAs, International Conference on*, pp. 357–362, 2011.

[3] LIRMM, "Computing, adaptation & related hardware/software," 2012. [Online]. Available: http://www.lirmm.fr/ADAC

[4] G. Magklis and et al., "Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor," *SIGARCH Comput. Archit. News*, vol. 31, pp. 14–27, May 2003.

[5] F. Xie and et al., "Compile-time dynamic voltage scaling settings: opportunities and limits," *SIGPLAN*, vol. 38, pp. 49–62, May 2003.

[6] D. Puschini and et al., "A game-theoretic approach for run-time distributed optimization on mp-soc," *International Journal of Reconfigurable Computing*, 2008.

[7] Q. Wu and et al., "Formal online methods for voltage/frequency control in multiple clock domain microprocessors," *SIGARCH Comput. Archit. News*, vol. 32, pp. 248–259, October 2004.

[8] Y. Zhu and F. Mueller, "Feedback edf scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling," *SIGPLAN Not.*, vol. 40, pp. 203–212, June 2005.

[9] U. Y. Ogras and et al., "Variation-adaptive feedback control for networks-on-chip with multiple clock domains," *Proceedings of the 45th Annual Design Automation Conference*, pp. 614–619, 2008.

[10] A. Sharifi and et al., "Feedback control for providing qos in noc based multicores," *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 1384–1389, March 2010.

[11] M. Ghasemazar and et al., "Minimizing the power consumption of a chip multiprocessor under an average throughput constraint," in *International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2010.

[12] F. Clermidy and et al., "An open and reconfigurable platform for 4g telecommunication: Concepts and application," in *DSD'09: Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 449–456.

[13] D. Puschini and et al., "Dynamic and Distributed Frequency Assignment for Energy and Latency Constrained MP-SoC," in *DATE'09: Design Automation and Test in Europe*, Nice, France, 04 2009, pp. 1564–1567.

[14] M. A. Al Faruque and et al., "Adam: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the 45th annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 760–765.

[15] L. T. Smit and et al., "Run-time mapping of applications to a heterogeneous soc," in *Proceedings of the International Symposium on System-on-Chip, SoC 2005*. IEEE, 2005, pp. 78–81.

[16] E. L. de Souza Carvalho and et al., "Dynamic task mapping for mpsocs," *IEEE Design & Test of Computers*, vol. 27, pp. 26–35, 2010.

[17] M. Mandelli and et al., "Energy-aware dynamic task mapping for NoC-based MPSoCs," in *IEEE International Symposium on Circuits and Systems*, 2011, pp. 1676–1679.

[18] A. K. Singh and et al., "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," *J. Syst. Archit.*, vol. 56, pp. 242–255, July 2010.

[19] S. Wildermann and et al., "Run time mapping of adaptive applications onto homogeneous noc-based reconfigurable architectures," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, dec. 2009, pp. 514 –517.

[20] T. Streichert and et al., "Dynamic task binding for hard-

ware/software reconfigurable networks," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*. New York, NY, USA: Acm, 2006, pp. 38–43.

[21] O. Derin and et al., "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, ser. NOCS '11. New York, NY, USA: ACM, 2011, pp. 129–136.

[22] H. Shen and F. Petrot, "Novel task migration framework on configurable heterogeneous mpsoc platforms," in *Design Automation Conference ASP-DAC. Asia and South Pacific*, 2009, pp. 733–738.

[23] F. Mulas and et al., "Thermal balancing policy for multiprocessor stream computing platforms," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 12, pp. 1870–1882, 2009.

[24] S. Bertozzi and et al., "Supporting task migration in multi-processor systems-on-chip: A feasibility study," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, 2006, pp. 1–6.

[25] M. Pittau and et al., "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation." in *ESTImedia*. IEEE, 2007, pp. 59–64.

[26] D. Barcelos and et al., "A hybrid memory organization to enhance task migration and dynamic task allocation in noc-based mpsocs," in *SBCCI'07:*. New York, NY, USA: ACM, 2007, pp. 282–287.

[27] L. Barthe and et al., "The secretblaze: A configurable and cost-effective open-source soft-core processor," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 310 –313.

[28] F. Moraes and et al., "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69–93, 2004.

[29] S. Rhoads, "Plasma - most mips i(tm)." [Online]. Available: (http://www.opencores.org/project,plasma)

[30] J. R. Levine, *Linkers and Loaders*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[31] N. Saint-Jean, "Study and design of self-adaptive multiprocessor systems for embedded systems," *PhD. Thesis, University of Montpellier 2, France*, 2008.

[32] C. Muller and et al., "Design challenges for prototypical and emerging memory concepts relying on resistance switching," in *Custom Integrated Circuits Conference (CICC)*, sept. 2011, pp. 1 –7.

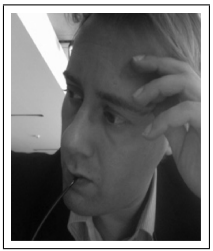
[33] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

[34] N. Binkert and et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[35] *AMD Opteron Shared Memory MP Systems*, Aug. 2002.

Dr. Luciano Ost is Associate Professor at the University of Montpellier II and member of microelectronic group at LIRMM, which is a cross-faculty research entity of the UM2 and the National Center for Scientific Research (CNRS). Ost got his PhD degree in Computer Science from PUCRS, Brazil in 2010. From 2004 to 2006 he worked as Research Assistant in this same University.

PhDc. Rafael Garibotti is currently a PhD candidate at LIRMM, France. He received his MSc. Degree in Microelectronics from EMSE, France and his BSc. Degree in Computer Engineer from PUCRS, Brazil. Moreover, he did a MBA in Project Management at FGV, Brazil. He worked two years and a half as Digital ASIC Designer at CEITEC S.A, besides having done an internship at NXP Semiconductors, France.



Dr. Gilles Sassatelli occupies a senior scientist position at LIRMM. He is currently the leader of the Adaptive Computing Group composed of 12 permanent staff researchers and over 20 Ph.D. students. He has published more than 150 publications in a number of renowned international conferences and journals, and regularly serves as track or topic chair in the major conferences in the area of reconfigurable computing.

Dr. Gabriel Marchesan Almeida is currently a senior research scientist at Karlsruhe Institute of Technology (KIT), in Germany. He received his BSc. Degree in Computer Science from URI, Brazil in 2004 and his MSc. Degree in Computer Science in 2007. In 2008 he moved to Montpellier, France where he got, in 2011, his PhD. in Microelectronics from University of Montpellier II.

Dr. Rémi Busseuil got in 2012 his PhD in microelectronics from LIRMM, Laboratory of Informatics, Robotics and Microelectronics of Montpellier, France. He received his Masters degree in microelectronics in 2009 from the ENS Cachan school, Paris, France and he currently holds a teaching position in France.

PhDc. Anastasiia Butko is currently a PhD candidate at LIRMM, France. Butko works in the area of adaptive multiprocessor architectures for embedded systems in the Adaptive computing group. She received her MSc. Degree in Microelectronics from UM2, France and MSc and BSc Degrees in Design of Electronic Digital Equipment from NTUU "KPI", Ukraine.

Prof.Dr. Michel Robert obtained respectively his MSc and PhD degree in 1980 and 1987 from the University of Montpellier. From 1980 to 1984 he was in the semiconductor division of Texas Instruments as R&D engineer. From 1985 to 1990, he was assistant professor at the University of Montpellier 2, and LIRMM laboratory. Since 2012 he is the president of the University of Sciences of Montpellier (France).

Prof. Dr.-Ing. Jürgen Becker received the Diploma degree in 1992, and his Ph.D. in 1997, both at Kaiserslautern University, Germany. His research work focused on application development environments for reconfigurable accelerators, including hw/sw codesign, parallelizing compilers and high-level synthesis. He is author of more than 250 scientific papers, published in peer-reviewed international journals and conferences.