Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 518904, 15 pages
doi:10.1155/2008/518904

Research Article

Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications

Andrea Acquaviva,[1] Andrea Alimonda,[2] Salvatore Carta,[2] and Michele Pittau[2]

[1] *Institute of Information Science and Technology, Electronics and Systems Design Group, University of Verona, 37100 Verona, Italy*
[2] *Department of Mathematics and Computer Science, University of Cagliari, 09100 Cagliari, Italy*

Correspondence should be addressed to Andrea Alimonda, alimonda@sc.unica.it

Received 4 April 2007; Revised 10 August 2007; Accepted 19 October 2007

Recommended by Alfons Crespo

Multiprocessor systems on chips (MPSoCs) are envisioned as the future of embedded platforms such as game-engines, smartphones and palmtop computers. One of the main challenge preventing the widespread diffusion of these systems is the efficient mapping of multitask multimedia applications on processing elements. Dynamic solutions based on task migration has been recently explored to perform run-time reallocation of task to maximize performance and optimize energy consumption. Even if task migration can provide high flexibility, its overhead must be carefully evaluated when applied to soft real-time applications. In fact, these applications impose deadlines that may be missed during the migration process. In this paper we first present a middleware infrastructure supporting dynamic task allocation for NUMA architectures. Then we perform an extensive characterization of its impact on multimedia soft real-time applications using a software FM Radio benchmark.

Copyright © 2008 Andrea Acquaviva et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

Multiprocessor systems on chip are of increasing and widespread use in many embedded applications ranging from multimedia to biological processing [1]. To satisfy application requirements by ensuring flexibility, many proposals were rising for architectural integration of system components and their interconnection. Currently, two main directions can be identified that lead to commercial products: master-slave architectures [2, 3], homogeneous architectures [4]. Due to the advances in integration and the increasing chip size, computational tiles are going to be resemble to elements of a cluster, with a nonuniform type of memory access (NUMA) and connected by networks on chip (NoC). This architecture addresses the scalability issues proper of symmetric multiprocessors that limit the number of integrable cores. It follows the structure envisioned for non-cache-coherent MPSoCs [5, 6]. These systems run tasks in private memories, like in distributed systems but similarly to symmetric multiprocessor SoCs, they exploit shared memories for communication and synchronization between processing nodes.

Even if MPSoCs are becoming the preferred target for multimedia embedded systems because they allow to fit a huge number of different applications, there are still fundamental challenges concerning the mapping of task into such a complex systems. The target is to balance the workload among processing units to minimize some metric such as overall execution time, power, or even temperature. Nevertheless, the software support implementing these strategies, that can be defined as the *resource manager*, must provide to the programmer a clean and standard interface for developing and running multimedia applications while hiding low-level details such as to which processor each task of the application is mapped. Given the large variety of possible use cases that these platforms must support and the resulting workload variability, offline approaches are no longer sufficient as application mapping paradigms, because they are limited in handling workload modifications due to variable quality of service (QoS) requirements. The extensive offline characterization on which they are based becomes unpractical in this context. Moreover, even when workload is constant, next generation processor will be characterized by variable performance, requiring online adaptation [7].

For this reason, dynamic mapping strategy have been recently proposed [8]. Run-time task allocation has been shown to be a promising technique to achieve load balancing, power minimization, temperature balancing, and reliability improvement [9]. To enable dynamic allocation, a task migration support must be provided. In distributed systems such as high-end multiprocessors and clusters of workstations (CoW) [10], task migration has been implemented and supported at the middleware level. Mosix is a well-known load-balancing support for CoW implemented at the kernel level that moves tasks in a fully transparent way. In the embedded multimedia system context however, task migration support has to trade off transparency with efficiency and lightweight implementation. User-level migration support may help in reducing its impact on performance [11]. However, it requires a nonnegligible effort by the programmer that has to explicitly define the context to be saved and restored in the new process life.

Multimedia applications are characterized by soft real-time requirements. As a consequence, migration overheads must be carefully evaluated to prevent deadline misses when moving processes between cores. These overheads depend on migration implementation which in turn depends on system architecture and communication paradigm. In distributed memory MPSoCs task migration involves task memory content transfer. As such, migration overhead depends both on the migration mechanism and on the task memory footprint. For this reason, there is the need of developing an efficient task migration strategy suitable for embedded multimedia MPSoC systems.

In this work, we address this need by presenting a middleware layer that implements task migration in MPSoCs and we assess its effectiveness when applied in the context of soft real-time multimedia applications. To achieve this target, we designed and implemented the proposed middleware on top of uClinux operating system running on a prototype multicore emulation platform [12]. We characterized its performance and energy overhead through extensive benchmarking and we finally assessed its effectiveness when applied to a multitask software FM Radio multitasking application. To test migration effectiveness, we impose in our experiments a variable frame rate to the FM Radio. Tasks are moved among processors as the frame rate changes to achieve the best configuration in terms of energy efficiency that provides the required performance level. Each configuration consists in a mapping of task to processor and the corresponding frequencies. Configurations are precomputed and stored for each frame rate.

Our experiments demonstrate that (i) migration at the middleware/OS level is feasible and improves energy efficiency of multimedia applications mapped on multiprocessor systems; (ii) migration overhead in terms of QoS can be hidden by the data reservoir present in interprocessor communication queues.

The rest of the paper is organized in the following way. Background work is covered in Section 2. Section 3 describes the architectural template we considered. Section 4 covers the organization of the software abstraction layer. Multimedia application mapping is discussed in Section 5 while results are discussed in Section 6.

2. BACKGROUND WORK

Due to the increasing complexity of these processing platforms, there is a large quantity and variety of resources that the software running on top of them has to manage. This may become a critical issue for embedded application developers, because resource allocation may strongly affect performance, energy efficiency, and reliability [13]. As a consequence, from one side there is need of efficiently exploit system resources, on the other side, being in an embedded market, fast and easy development of applications is a critical issue. For example, since multimedia applications are often made of several tasks, their mapping into processing elements has to be performed in a efficient way to exploit the available computational power and reducing energy consumption of the platform.

The problem of resource management in MPSoCs can be tackled from either a static or dynamic perspective. Static resource managers are based on the a priori knowledge of application workload. For instance, in [14] a static scheduling and allocation policy is presented for real-time applications, aimed at minimizing overall chip power consumption taking also into account interprocessor communication costs. Both worst case execution time and communication needs of each tasks are used as input of the minimization problem solved using integer linear programming (ILP) techniques. In this approach, authors first perform allocation of tasks to processors and memory requirement to storage devices, trying to minimize the communication cost. Then scheduling problem is solved, using the minimization of execution time as design objective.

Static resource allocation can have a large cost, especially when considering that each possible set of applications may lead to a different use case. The cost is due to run-time analysis of all use cases in isolation. In [15] a composition method is proposed to reduce the complexity of this analysis. An interesting semistatic approach that deals with scheduling in multiprocessor SoC environments for real-time systems is presented in [16]. The authors present a task decomposition/clustering method to design a scalable scheduling strategy. Both static and semistatic approaches have limitations in handling varying workload conditions due to data dependency or to changing application scenarios. As a consequence, dynamic resource management came into play.

Even if scheduling can be considered a dynamic resource allocation mechanism, in this paper we assume that a main feature of a dynamic resource manager in a multiprocessor system is the capability of moving tasks from processing elements at run time. This is referred to as task migration.

In the field of multiprocessor systems-on-chip, process migration can be effectively exploited to facilitate thermal chip management by moving tasks away from hot processing elements, to balance the workload of parallel processing elements and reduce power consumption by coupling dynamic voltage and frequency scaling [17–19]. However, the implementation of task migration, traditionally developed

for computer clusters or symmetric multiprocessor, cache-coherent machines, poses new challenges [11]. This is specially true for non-cache-coherent MPSoCs, where each core runs its own local copy of the operating system in private memory. A migration paradigm similar to the one implemented in computer clusters should be considered, with the addition of a shared memory support for interprocessor communication.

For instance, many embedded system architectures do not even provide support for virtual memory; therefore, many task migration optimization techniques applied to systems with remote paging support cannot be directly deployed, such as the eager dirty [10] or the copy-on-reference [20] strategies.

In general, migrating a task in a fully distributed system involves the transfer of processor state (registers), user level and kernel level context, and address space. A process address space usually accounts for a large fraction of the process state; therefore, process migration performance largely depends on the transfer efficiency of the address space. Although a number of techniques have been devised to alleviate this migration cost (e.g., lazy state transfer, precopying, residual dependencies [21]), a frequent number of migrations might seriously degrade application performance in an MPSoC scenario. As a consequence, assessing the impact of migration overhead is critical.

In the context of MPSoCs, in [8] a selective code/data migration strategy is proposed. Here authors use a compilation-level code profiling technique to evaluate the communication energy cost of transferring each function and procedure over the on-chip network. This information is used to decide whether it is worth migrating tasks on the same processor to reduce communication overhead or transferring data between them.

In [11], a feasibility study for the implementation of a lightweight migration mechanism is proposed. The user-managed migration scheme is based on code checkpointing and user-level middleware support. The user is responsible for determining the context to be migrated. To evaluate the practical viability of this scheme, authors propose a characterization methodology for task migration overhead, which is the minimum execution time following a task migration event during which the system configuration should be frozen to make up for the migration cost.

In this work, task migration for embedded systems is evaluated when applied to a real-world soft real-time application. Compared to [11], our migration strategy is implemented at the operating system and middleware level. Checkpoints are only used to determine migration points, while the context is automatically determined by the operating system.

3. TARGET ARCHITECTURE ORGANIZATION

The software infrastructure we present in this work is targeted to a wide range of multicore platforms having a number of homogeneous cores that can execute the same set of tasks, otherwise task migration is unfeasible. A typical target homogeneous architecture is the one shown in Figure 1(a). The architectural template we consider is based on a configurable number of 32-bit RISC processors without memory management unit (MMU) accessing cacheable private memories and a single noncacheable shared memory.

The template we are targeting is compliant with state-of-the-art MPSoC architectures. For instance, because of its synergistic processing elements with local storage and shared memory used as support for message-based stream processing is closely related to CELL [22].

In our target platform, each core runs in the logical private memory a single operating system instance. This is compliant with the state-of-the-art homogeneous multicore architectures such as the MP211 multicore by NEC [23].

As regards as task migration, the main architectural impact is related to the interconnect model. As we will describe in Section 6, our target platform uses a shared bus as interconnect. This is the worst case for task migration since it implies data transfers from the private memory of one core to the one of another core.

As far as this MPSoC model is concerned, processor cores execute tasks from their private memory and explicitly communicate with each others by means of the shared memory [24]. Synchronization and communication are supported by hardware semaphores and interrupt facilities: (i) each core can send interrupts to others using a memory-mapped interprocessor interrupt module; (ii) cores can synchronize between each other using a hardware *test-and-set* semaphore module that implements test-and-set operations. Additional dedicated hardware modules can be used to enhance interprocessor communication [25, 26]. In this paper, we consider a basic support to save the portability of the approach.

4. SOFTWARE INFRASTRUCTURE

Following the distributed NUMA architecture, each core runs its own instance of the uClinux operating system [27] in the private memory. The uClinux OS is a derivative of Linux 2.4 kernel intended for microcontrollers without MMU. Each task is represented using the process abstraction, having its own private address space. As a consequence, communication has to be explicitly carried on using a dedicated shared memory area on the same on-chip bus. The OS running on each core sees the shared area as an external memory space.

The software abstraction layer is described in Figure 1(b). Since uClinux is natively designed to run in a single-processor environment, we added the support for interprocessor communication at the middleware level. This organization is a natural choice for a loosely coupled distributed systems with no cache coherency, to enhance efficiency of parallel application without the need of a global synchronization, that would be required by a centralized OS. On top of local OSes we developed a layered software infrastructure to provide an efficient parallel programming model for MPSoC software developers enabled by an efficient task migration support layer.

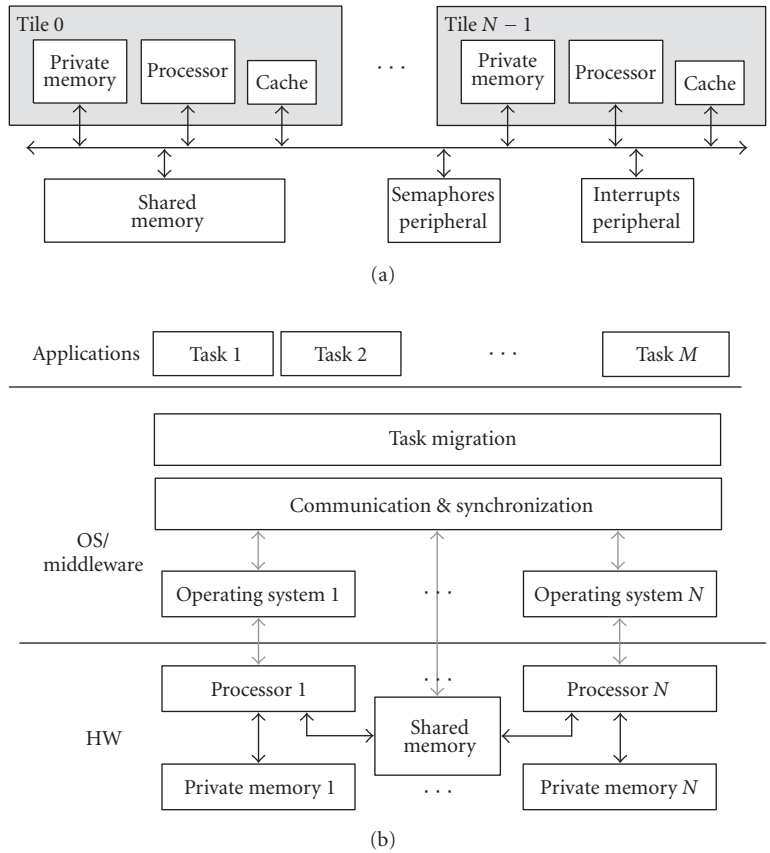


Figure 1: Hardware and software organizations: (a) target hardware architecture; (b) scheme of the software abstraction layer.

4.1. Communication and synchronization support

The communication library supports message passing through mailboxes. They are located either in the shared memory space or in smaller private scratch-pad memories, depending on their size and depending if the task owner of the queue is defined as migratable or not. The concept of migratable task will be explained later in this section. For each process a message queue is allocated in shared memory.

To use shared memory paradigm, two or more tasks are enabled to access a memory segment through a *shared malloc* function that returns a pointer to the shared memory area. The implementation of this additional system call is needed because by default the OS is not aware of the external shared memory. When one task writes into a shared memory location, all the other tasks update their internal data structure to account for this modification. Allocation in shared memory is implemented using a parallel version of the Kingsley allocator, commonly used in Linux kernels.

Task and OS synchronization is supported providing basic primitives like binary and counting semaphores. Both spinlock and blocking versions of semaphores are provided. Spinlock semaphores are based on hardware test-and-set memory-mapped peripherals, while nonblocking semaphores also exploit hardware interprocessor interrupts to signal waiting tasks.

4.2. Task migration support

To handle dynamic workload conditions and variable task and workload scenarios that are likely to arise in MPSoCs targeted to multimedia applications, we implemented a task migration strategy enabled by the middleware support. Migration policies can exploit this mechanism to achieve load balancing for performance and power reasons. In this section, we describe the middleware-level task migration support. In Section 6, we will show how task migration can be used to improve the efficiency of the system.

In our implementation, migration is allowed only at predefined checkpoints, that are provided to the user through a library of functions together with message passing primitives. A so-called *master daemon* runs in one of the cores and takes care of dispatching tasks on the processors. We implemented two kinds of migration mechanisms that differs in the way the memory is managed. A first version, based on a so-called "task-recreation" strategy, kills the process on the original processor and recreate it from scratch on the target processor. This support works only in operating systems supporting dynamic loading, such as uClinux. Task recreation is based on the execution of fork-exec system calls that take care of allocating the memory space required for the incoming task. To support task recreation on an architecture without MMU performing hardware address

translation, a position-independent type of code (called PIC) is required to prevent the generation of wrong references of pointers, since the starting address of the process memory space may change upon migration.

Unfortunately, PIC is not supported by the target processor we are using in our platform (microblazes) [28]. For this reason, we implemented an alternative migration strategy where a replica of each task is present in each local OS, called "task-replication." Only one processor at a time can run one replica of the task. While here the task is executed normally, in the other processors it is in a queue of suspended tasks. As such, a memory area is reserved for each replica in the local memory, while kernel-level task-related information is allocated by each OS in the process control block (PCB) (i.e., an array of pointers to the resources of the task). A second valid reason to implement this alternative technique is because deeply embedded operating systems are often not capable of dynamic loading and the application code is linked together with the OS code. Task replication is suitable for an operating system without dynamic loading because the absolute memory position of the process address space does not change upon migration, since it can be statically allocated at compile time. This is the case of deeply embedded operating systems such as RTEMS or eCos. This is compliant also with heterogeneous architectures, slave processors run a minimalist OS, that is, a library statically linked with the tasks to be run, that are known a priori. The master processor typically runs a general purpose OS such as Linux. Even if this technique leads to a waste of memory for migratable tasks, it has also the advantage of being faster, since it cuts down on memory allocation time with respect to a task recreation.

To further limit waste of memory, we defined both *migratable* and *nonmigratable* types of tasks. A migratable task is launched using a special system call, that enables the replication mechanism. Nonmigratable tasks are launched normally. As such, in the current implementation the user is responsible for distinguishing between the two types of tasks. However, in future implementation the middleware itself could be responsible of selecting migratable tasks depending on task characteristics.

The difference in terms of migration costs for the two strategies is shown in Figure 2. Cost is shown in terms of processor cycles needed to perform migrations as a function of the task size. In both cases, there is a contribution to migration overhead due to the amount of data transferred through the shared memory. Moreover, for task recreation technique, there is another overhead due to the additional time required to reload the program code from the file system. This explains the offset between the two curves. Moreover, the task recreation curve as a larger slope. This is due to the fact that a large amount of memory transfers also lead to an increasing contention on the bus, so that the contribution on the execution time increases more as the file size increases with respect to the task replication case.

In our system, the migration process is managed using two kinds of kernel daemons (part of the middleware layer), a master daemon running in a single processor, and slave daemons running in all the processors. The communication between master and slave daemons is implemented using dedicated, interrupt-based messages in shared memory. The master daemon takes care of implementing the run-time task allocation policy. Tasks can be migrated only corresponding to user-defined checkpoints. The code of the checkpoints is provided as a library to the programmer. When a new task or an application (i.e., a set of tasks) is launched by the user, the master daemon sends a message to each slave, that forks an instance of each task in the local processor. Depending on master's decision, tasks that have not to be executed on the local processor are placed in the suspended tasks queue, while the others are placed in the ready queue.

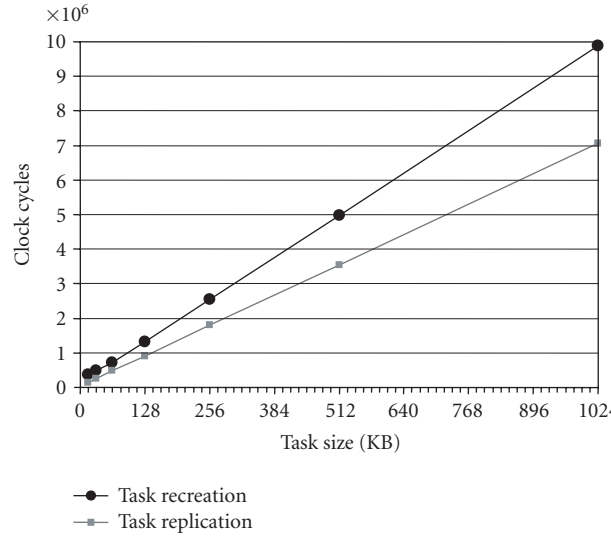


Figure 2: Migration cost as a function of task size for task replication and task recreation.

During execution, when a task reaches a user-defined checkpoint, it checks for migration requests performed by the master daemon. If the migration is taken, they suspend their execution waiting to be deallocated and restore to another processor from the migration middleware. When the master daemon wants to migrate a task, it signals to the slave daemons of the source processor that a task has to be migrated. A dedicated shared memory space is used as a buffer for task context transfer. To assist migration decision, each slave daemon writes in a shared data structure the statistics related to local task execution (e.g., processor utilization and memory occupation of each task) that are periodically read by the master daemon.

Migration mechanisms are outlined in Figure 3. Both execution and memory views are shown. With task replication (Figures 3(a) and 3(b)), the address space of all the tasks is present in all the private memories of processor 0,1, and 2. However, only a single instance of a task is running on processor 0, while others are sleeping on processors 1 and 2. It must be noted that master daemon (*M_daemon* in Figure 3) runs on processor 0 while slave daemons (*S_daemon* in Figure 3) run on all of the processors. However, any processor can run the master daemon.

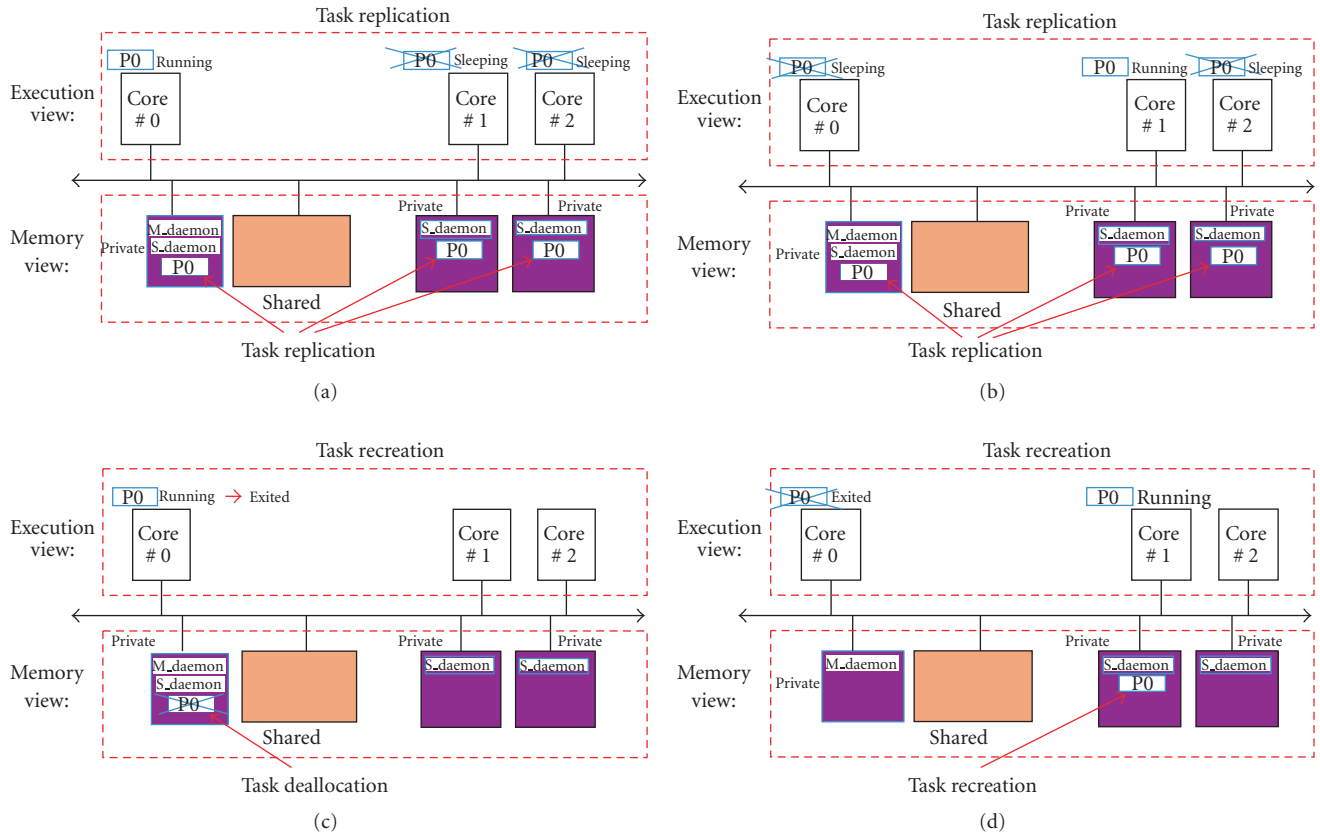


FIGURE 3: Migration mechanism: (a) task replication phase 1; (b) task replication phase 2; (c) task recreation phase 1; (d) task recreation phase 2.

Figures 3(c) and 3(d) shows task recreation mechanism. Before migration, process P0 runs on processor 0 and occupies memory space on the private memory of the same processor. Upon migration, P0 performs an exit system call and thus its memory space is deallocated. After migration (Figure 3(d)), memory space of P0 is reallocated on processor 1, where P0 runs.

Being based on a middleware-level implementation running on top of local operating systems, the proposed mechanism is suitable for heterogeneous architectures and its scalability is only limited by the centralized nature of the master-slave daemon implementation.

It must be noted that we have implemented a particular policy, where the master daemon keeps track of statistics and triggers the migration; however, based on the proposed infrastructure, a distributed load balancing policy can be implemented with slave daemons coordinating the migration without the need of a master daemon. Indeed, the distinction between master and slaves is not structural, but only related to the fact that the master is the one triggering the migration decision, because it keeps track of task allocation and loads. However, using an alternative scalable distributed policy (such as the Mosix algorithm used in computer clusters) this distinction is no longer needed and slave daemons can trigger migrations without the need of a centralized coordination.

5. MULTIMEDIA SOFT REAL-TIME APPLICATION MAPPING

Multimedia applications are typically composed by multiple tasks. In this paper, we consider each task as a process with its own private address space. This allows us to easily map these applications on a distributed memory platform like the one we are targeting in this work. The underlying framework supports task migration so that dynamic resource management policies can take care of run-time mapping of task to processors, to improve performance, power dissipation, thermal management, reliability. The programmer is not exposed to mapping issues, it is only responsible for the communication and synchronization as well as code checkpointing for migration. Indeed, task migration can occur only corresponding to checkpoints manually inserted in the code by the programmer.

In our migration framework, all the data structures describing the task in memory are replicated. Upon migration, the only kernel structure that is moved is the stack. As such, if a process has opened a local resource, this information is lost after migration. The programmer is responsible for carefully selecting migration points or eventually reopening resources left open in the previous task life.

An alternative, completely transparent approach, is the one implemented in Mosix for computer clusters [10], where

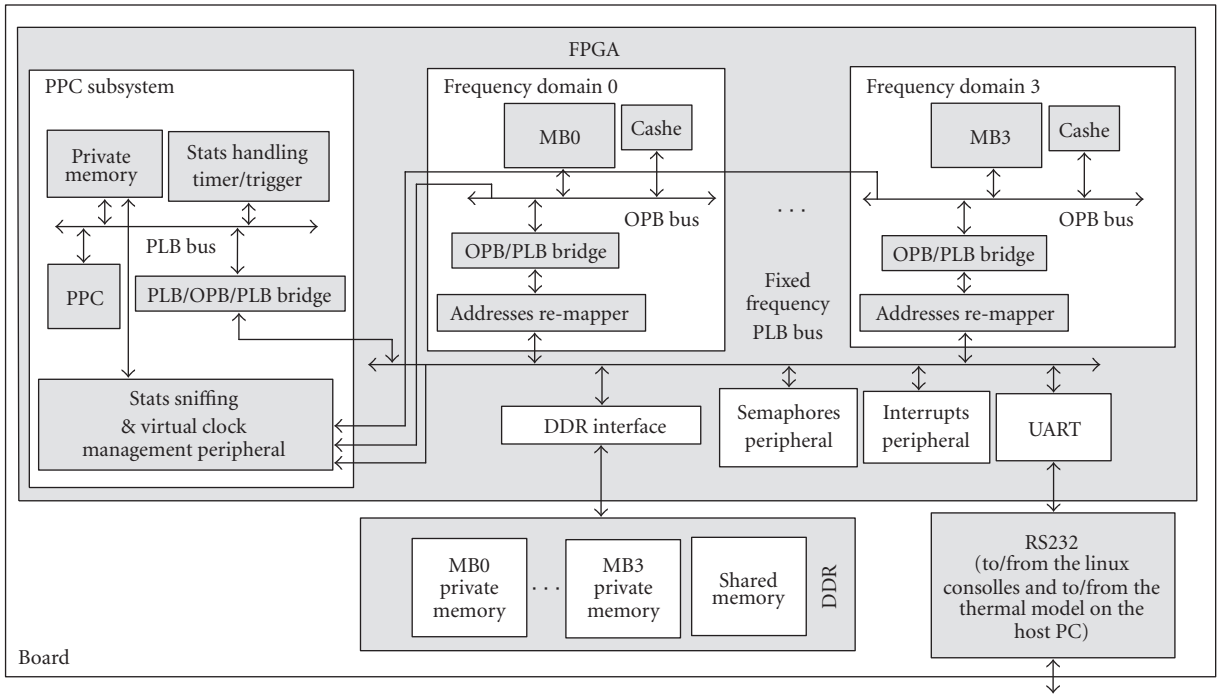


Figure 4: Overview HW architecture of emulated MPSoC platform.

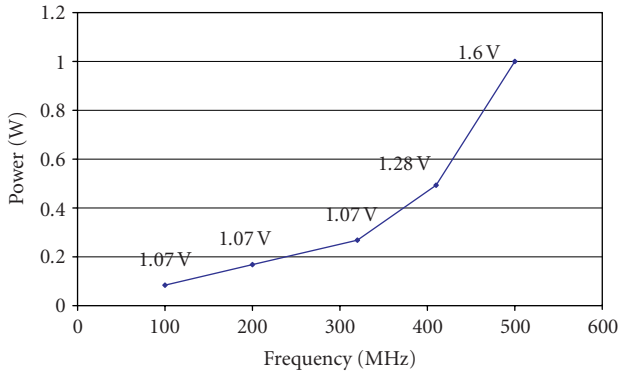


Figure 5: Power model: power as a function of the frequency.

a home processor defined for each process requires the implementation of a forwarding layer that takes care of forwarding system calls to the home node. In our system, we do not have the notion of home node. A more complex programming paradigm is thus traded off with efficiency and predictability of migration process. This approach is much more suitable to an embedded context, where controllability and predictability are key issues.

6. EXPERIMENTAL RESULTS

In this section, results of a deep experimental analysis of the multiprocessing middleware performed using an FPGA-based multiprocessor emulation platform are described. A first run of tests, based on synthetic applications, have been used to characterize the overhead of migration in terms of code checkpointing and run-time support. A second set of tests were performed on a soft real-time streaming application to highlight the effect of migration on a real-life multimedia application. Results show that a temporary QoS degradation due to migration can be hidden by exploiting data reservoir in interprocessor communication buffers and provide design guidelines concerning buffer size and migration times.

6.1. Emulation platform description and setup

For the simulation and performance evaluation of the proposed middleware, we used an FPGA-based, cycle accurate, MPSoC hardware emulator [12] built on top of the Xilinx XUP FPGA board [28], and described in Figure 4. Time and energy data are run-time stored using a nonintrusive, statistics subsystem, based on hardware sniffers which store frequencies, bus, and memory accesses. A PowerPC processor manages data and control communication with the host PC using a dedicated UART-based serial protocol. Run-time frequency scaling is also supported and power models running on the host PC allow to emulate voltage scaling mechanism. Frequency scaling is based on memory-mapped frequency dividers, which can be programmed both by microblazes or by PowerPC. The power associated to each processor is 1 Watt for the maximum frequency, and scales down almost cubically to 84 mW as voltage and frequency decrease. Power data refers to a commercial embedded RISC processor and is provided by an industrial partner. Power/frequency relationship is detailed in Figure 5. The emulation platform runs at 1/10 of the emulated frequency, enabling the experimentation of complex applications which may not be

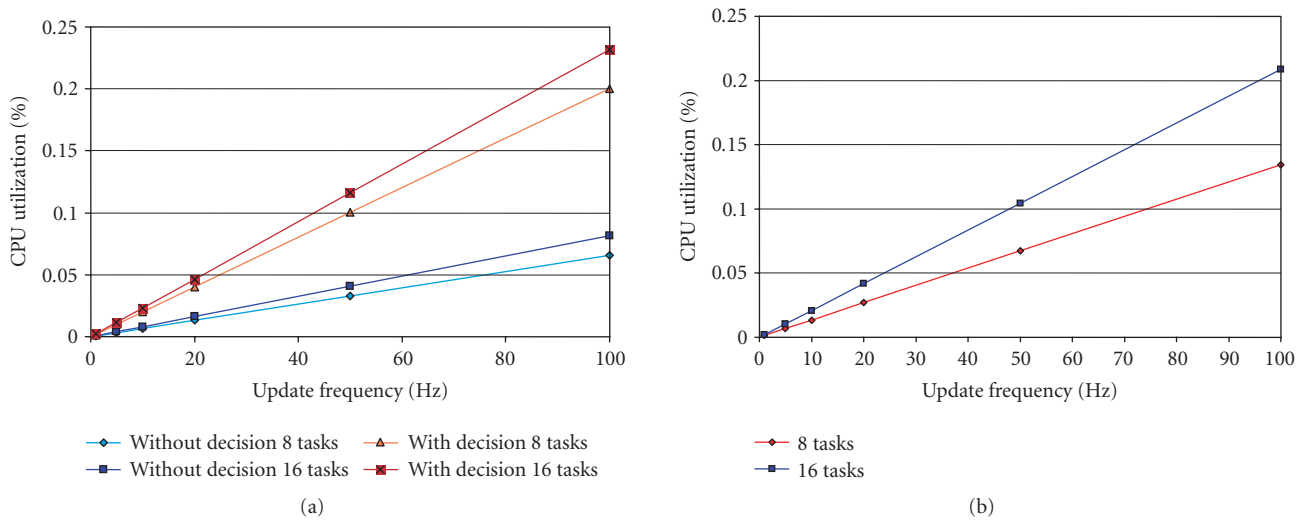


Figure 6: Migration daemon overhead in terms of processor utilization: (a) master daemon CPU load; (b) slave daemon CPU load.

experimented using software simulators with comparable accuracy.

In our platform, private memories are physically mapped into an external DDR memory. This is because the image of uClinux does not fit into the on-chip SRAM. This architectural solution is similar to NEC MP211 multicore platform [23].

Even in this particular implementation, we use an external DDR to map shared and private memories, the proposed approach (i.e., task replication version) can be also implemented in deeply embedded operating systems whose memory footprint would fit into a typical on-chip SRAM.

6.2. Migration support characterization

We present the results of experiments carried out to assess the overhead of the task migration infrastructure in terms of execution time. We exploited a controllable synthetic benchmark to this purpose, thanks to which we evaluated three components of the migration overhead: (i) the cost of the slave daemons running in background to check tasks runtime statistics; (ii) the cost of the master daemon running in background to trigger task shutoff and resume in the various processor cores; (iii) the number of cycles required to complete a migration, that is to shut off a task in the source core, transfer its context, and resume it on the destination core.

We evaluated the overhead of the master daemon by measuring its CPU utilization. Its job is to read the statistics about all the tasks in the system from the shared memory and look in a prestored lookup table whether or not a task should be migrated and where. As such, this overhead does not include the overhead due to (i) the pure cost of migration support and (ii) to the cycles spent to look in the table. We measured these two contributions separately to evaluate migration overhead independently from the particular migration policy. Figure 6(a) shows the CPU load for the two considered cases as a function of the frequency of daemon invocation and for two different quantities of task present in the system. We observed a negligible overhead, highlighting that there is a room for implementing more complex task allocation and migration policies in the master daemon without impacting system performance.

We evaluated the CPU utilization imposed by slave daemons which periodically write task information (currently the task load) in a dedicated data structure allocated in shared memory. This information is then exploited for migration decisions. In Figure 6(b), the CPU utilization of the daemon is shown as a function daemon period in a range of 1 to 100 Hz, two curves are plotted, for 8 and 16 tasks, referred to a core speed of 100 MHZ. It is worth to note that, although the processor speed is moderate, the overhead is negligible being lower than 0.25% in the worst case (update frequency of 100 Hz).

Finally, to quantitatively evaluate the cost of a single task migration, we exploited the capability of our platform to monitor the amount of processor cycles needed to perform a migration, from the checkpoint execution to the complete resume of the task on the new processor. Figure 7 shows migration cost as a function of the task size. Migration overhead is dominated by data transfer. Two main contributions affect the size of the task context to be transferred. The first is the kernel memory space reserved by uClinux to each process to allocate process data structures and the user memory space associated to the task to be migrated. The first contribution consists in our current implementation only of the kernel stack of the processor. This is because the other data structures are already replicated in all the kernels in the other cores (because of the replication strategy). Data, code sections, and heap belong to the second contribution. It must be taken into account that the minimum allocation of user space memory for each processor is 64 KB, even if the process is smaller. In our current implementation, we copy the whole memory areas because we exploit the presence of the memory pointers in the process control block (PCB). With the help of some additional information, optimized techniques can be implemented to reduce the amount of data to be copied, that will

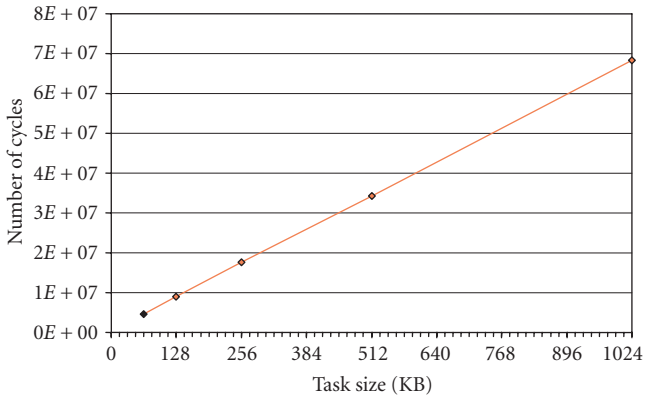


Figure 7: Task migration cost.

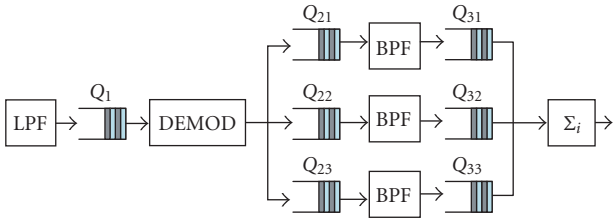


Figure 8: FM radio.

be an object of future work. The linear behavior of the cycle count shows that the only overhead added by the migration support is a constant amount of cycles that are needed to copy the task kernel context and to perform the other migration control operations as described in Section 4. This is visible as the intercept with the Y-axis is in the plot and its value is 104986 cycles.

Finally, it must be noted that a hardware support for data transfer, such as a DMA controller, that is currently not present in our emulation platform, could greatly improve migration efficiency.

6.3. Impact of migration on soft real-time streaming applications

To evaluate the effectiveness of the proposed support on a real test bed, we ported to our system a software FM radio benchmark, that is a representative of a large class of streaming multimedia applications following the split-join model [29] with soft real-time requirements. It allows to evaluate the tradeoff between the long-term performance improvement given by migration-enabled run-time task remapping and the short-term overhead and performance degradation associated to migration.

As shown in Figure 8, the application is composed by various tasks, graphically represented as blocks. Input data represent samples of the digitalized PCM radio signal which has to be processed in order to produce an equalized baseband audio signal. In the first step, the radio signal passes through a *lowpass filter* (LPF) to cut frequencies over the radio bandwidth. Then, it is demodulated by the *demodulator* (DEMOD) to shift the signal at the baseband and produce the audio signal. The audio signal is then equalized with a number of *bandpass filters* (BPF) implemented with a parallel split-join structure. Finally, the *consumer* (Σ) collects the data provided by each BPF and makes the sum with different weights (gains) in order to produce the final output.

To implement the communication between tasks, we adopted the message passing paradigm discussed in Section 4. Synchronization among the tasks is realized through message queues, so that each task reads data from its input queue and sends the processed results to the output queue to be read by the next task in the software pipeline. Since each BPF of the equalizer stage acts on the same data flow, the demodulator has to replicate its output data flow writing the same packet on every output queue.

Thanks to our platform, we could measure the workload profile of the various tasks. We verified that the most computational intensive stage is the demodulator, imposing a CPU utilization of 45% of the total, while for the other tasks we observed, respectively, 5% for the LPF, 13% for each BPF and 5% for the consumer. This information will be used by the implemented migration support to decide which task has to be migrated.

To assess migration advantages and costs, we refer to migration represented in Figure 9. We start from a single processor configuration providing a low frame rate. For each frame rate we stored the best configuration in terms of energy efficiency that provides the required performance level. Thus, as the frame rate imposed by the application increases, we move first to a two processors configuration and then to a three processors configuration. For each processor we also predetermined the appropriate frequency level.

In the rest of this subsection, we first show how the best configurations and the frame rate represent the crossing points between configurations. Second, to show what is the cost of transition from a configuration to another, we detail migration costs in terms of performance (frame misses) and energy.

Optimal task configurations

We use migration to perform run-time task remapping, driving the system from an unbalanced situation to a more balanced one. The purpose of the migration policy we used in this experiment is to split computational load between processing cores. Each processor automatically scales its speed depending on the value stored in a lookup table as a function of the frame rate.

It must be noted that minimum allowed core frequency is 100 MHz. As a consequence, there is no convenience in moving tasks if the cumulative processor utilization leads to a frequency lower than the minimum. Task load depends on the frame rate specified by the user, which in turn depends on the desired sound quality. After initial placement, migration is needed to balance workload variations generated by run-time user requests. The migration is automatically triggered by the master daemon integrated in our layer, following an offline computed lookup table which gives the best task/frequency mapping as a function of these run-time events. It is worth noting that in this paper, we are not

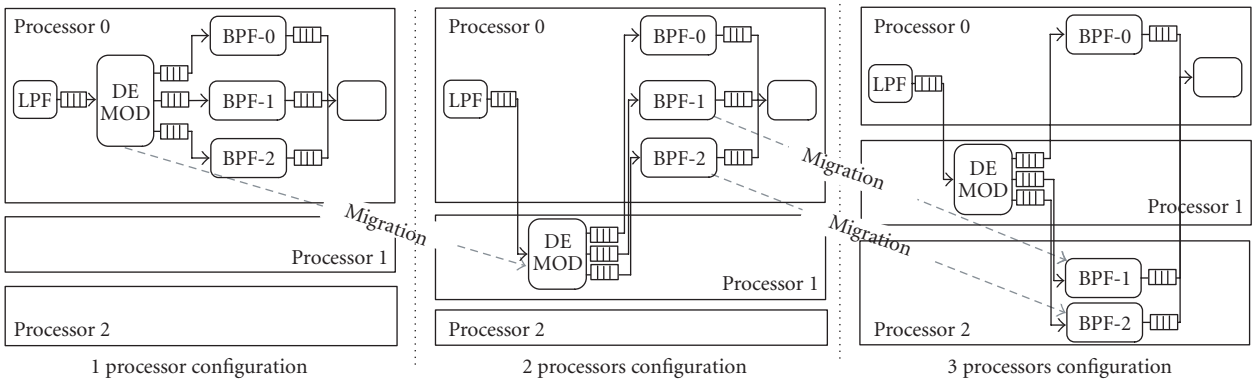


Figure 9: Task migrations performed to support the required frame rate.

interested in evaluating task/frequency allocation policies but the efficiency of the infrastructure. Rather, we are interested in demonstrating how migration cost is compensated by the execution time saved after migration when the system runs a more balanced configuration.

Task reallocation allows to improve the energy efficiency of the system at the various frame rates requested by the user. We show the impact of task reallocation on energy efficiency and we outline energetic cost of task migration. Energy-efficient configurations are computed starting from energy versus frame-rate curves. Each curve represents all the configurations that are obtained by keeping the same number of processors while increasing the frequency to match the desired frame rate. Then we take the Pareto configurations that minimize energy consumption for a given frame rate.

We considered two different energy behaviors of our system. In the first configuration, called *always on*, we do not use any built-in power-saving scheme, that is, the cores are always running and consume power depending on their frequency and voltage. In the second configuration, we emulate a *shutdown-when-idle* power-saving scheme in which the cores are provided with an ideal low-power state in which their power consumption is negligible.

In the *always on* case, the power consumed by the cores as a function of the frame rate is shown in Figure 10. On the left side of the figure, the plot shows three curves obtained by computing the power consumed when mapping the FM Radio tasks in one, two, or three cores. Following one of the curves, it can be noted that, due to the frequency discretization, power consumption is almost constant in frame rate intervals where the cores run at the same frequency/voltage. In each one of these intervals, the amount of idleness is maximum at the beginning and decreases until a frequency step is needed to support the next frame rate value. On the right side of the figure, the corresponding configurations are highlighted, as well as the task mapping and frequencies for each core. Being the demodulator task, the more computational intensive, it is the task that runs at the higher frequency than others.

In order to determine the best power configuration for each frame rate, we computed the Pareto points, as shown in Figure 11. Intuitively, when the frame rate increases, a higher number of cores are needed to get the desired QoS in an energy-efficient way. However, due to the frequency discretization, this is not true. There are cases in which energy efficiency is achieved using a lower number of cores for a higher frame rate. In practice, the "Pareto path" shown on the right side of Figure 11 is not monotonic on the number of cores. Migrations needed to go from one configuration to another are also highlighted as right-oriented arrows on the left side of the figure.

Conversely, in the *shutdown-when-idle* case, power consumption is no longer constant for a given frequency/voltage, because the cores are more active as the frame rate increases for a given frequency. This is shown in Figure 12, where the same curves presented before for the *always on* case are shown, representing static task mappings in one, two, and three cores.

Frequency discretization has less impact in this case. Indeed, by observing the Pareto curve shown in Figure 13, it is evident that the number of cores of the Pareto configurations is monotonically increasing as a function of the frame rate. In Figure 13 migration costs are also detailed. Following the Pareto curve from left to right, the various task mappings and corresponding core frequencies are shown. In particular, core configurations across migration points are shown as well as migration costs in terms of energy. The cost of migration has been evaluated in terms of cycles as described before. These cycles lead to an energy cost, depending on the frequency and voltage at which the core is executing the migration. In Figure 13, this cost is computed considering that migration is performed at the maximum frame rate sustainable at a given configuration, that is, 3000 fps for the 1-processor configuration and 4200 fps for the 2-processor configuration. In general, however, migration cost for transitioning from one configuration to another depends on the actual frame rate which determines the processor frequency. To detail the cost of migration at the various frequencies, we reported energy spent on transitioning from each configuration to another in Figure 14.

Migration energy cost depends on the frequency and voltage. The first set of bars shows the cost of migration from one-core configuration to two-cores configuration, which implies moving only the demodulator task. The second set

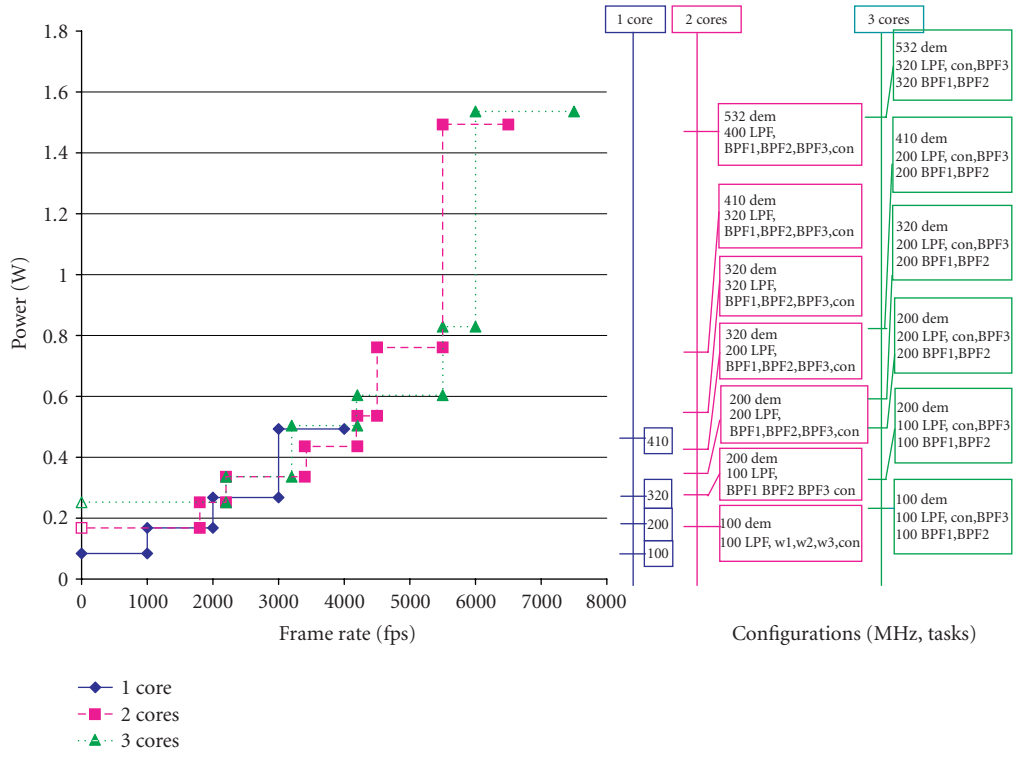


FIGURE 10: Energy cost of static task mappings in the *always on* case.

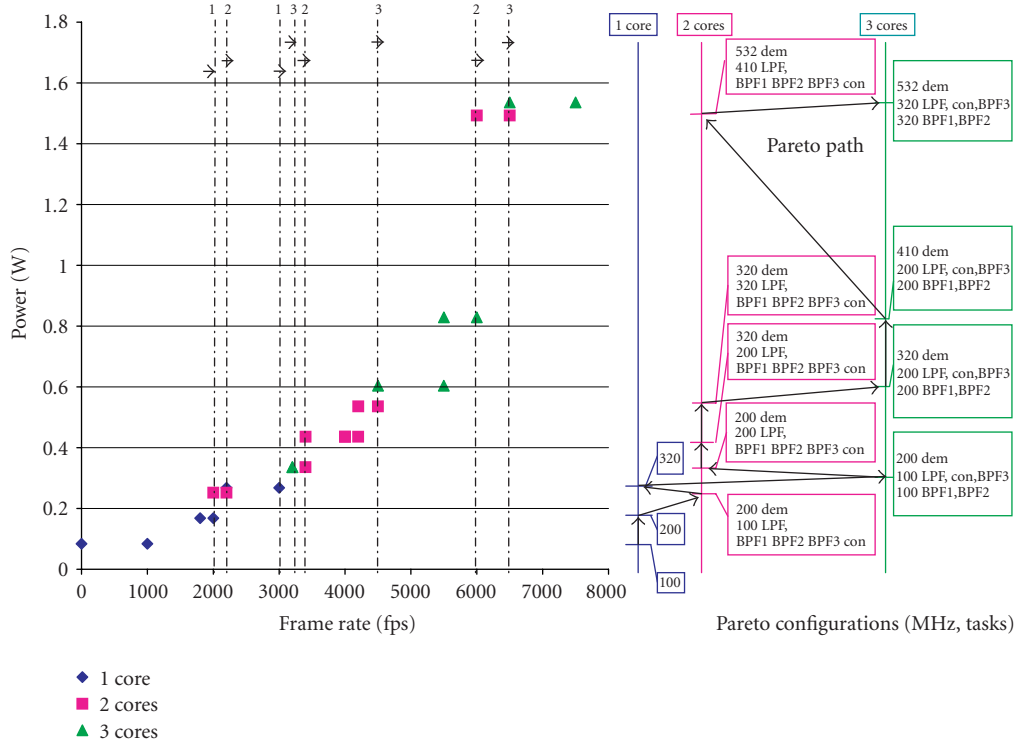


FIGURE 11: Pareto optimal configurations in the *always on* case.

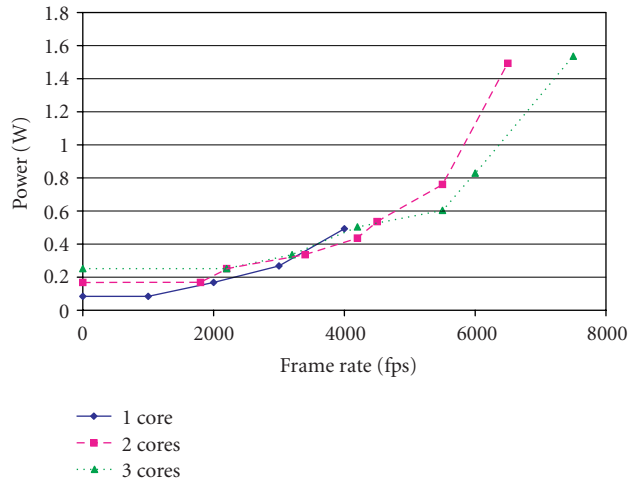


Figure 12: Energy cost of static task mappings in the *shutdown-when-idle* case.

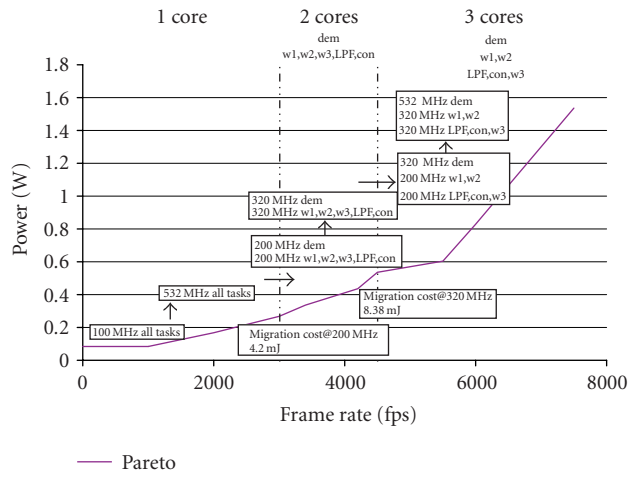


Figure 13: Pareto optimal configurations in the *shutdown-when-idle* case.

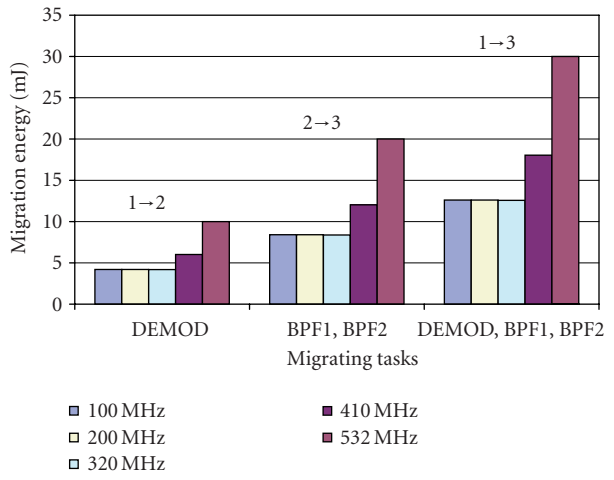


Figure 14: Task migration cost as a function of the frequency.

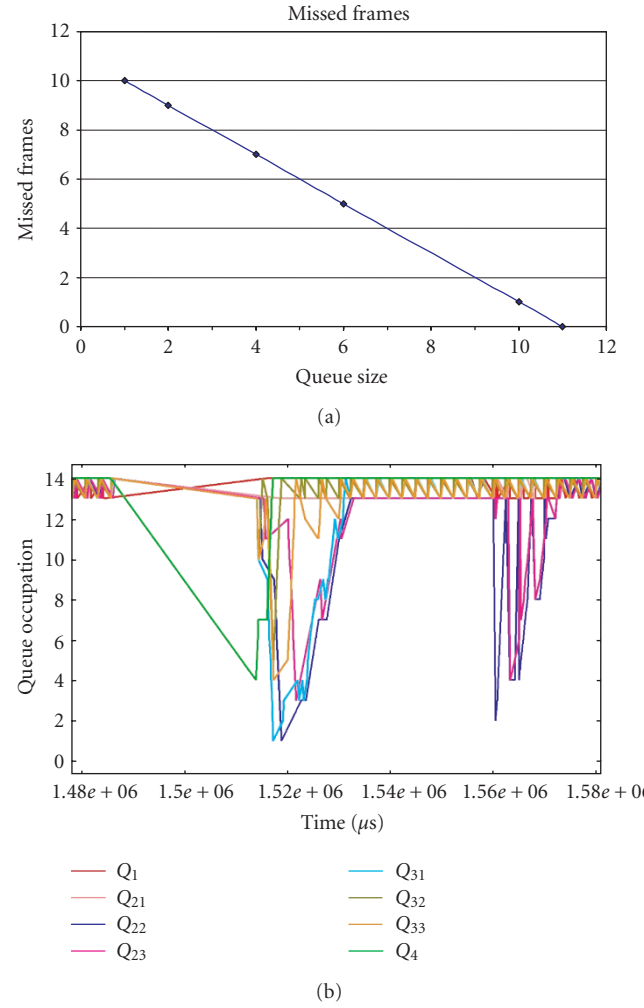


Figure 15: Impact of migration of communicating tasks: (a) Queue size versus deadline misses; (b) Queue occupancy during migration.

of bars shows the cost of migration from two cores to three cores, which implies moving the two BPF tasks from one core to another. The third set of bars shows the cost of migration from one core to three cores directly, which involves moving 4 tasks, the two BPFs, and the demodulator. It is worth noting that energy cost is almost equal from 100 MHz to 320 MHz. This is because in the power models we considered that voltage does not scale below 320 MHz, as reported in Figure 5. As such, dynamic power consumption of the processor linearly depends on the frequency and thus energy consumption is almost constant.

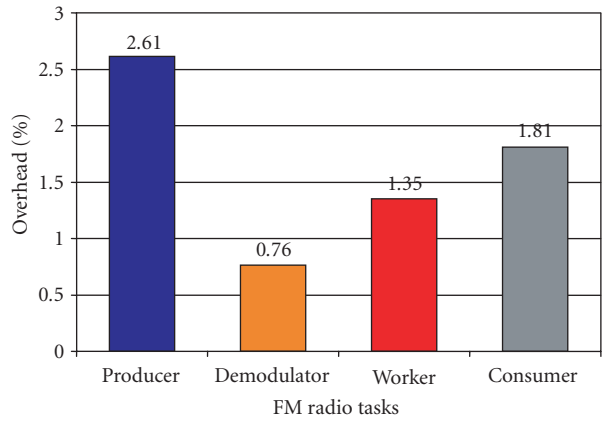
In the *shutdown-when-idle* case, migration implies an increase of the workload and thus less time in low-power state. In the *always on* case, migration cost is hidden if there is enough idleness. However, this depends on the relationship between frame rate (required workload), discrete core frequency and migration cost. In our experiment, migration overhead is completely hidden by the idleness due to frequency discretization. That is why migration costs are not shown in Figure 11.

```
while(1) {

   checkpoint();

   infifo.get (&elem);
   for(q=0; q < FM_FRAME_SIZE; q++)   {
       // demodulation
       temp = (elem.d[q] * lastElem);
      elem.d[q] = mGain *
              arctan(elem.d[q], 1.0/lastElem);
      lastElem = elem.d[q];
   }
   for(int w=0; w<n_workers; w++)   {
      outfifo[w]->put(&elem);
   }
}
```

(a)



(b)

Figure 16: Impact of migration of communicating tasks: (a) code of demodulator with checkpoints; (b) checkpointing overhead.

Migration overhead

Once energy-efficient configurations have been determined, these can be used to achieve the wanted frame rate in an energy-efficient way. However, when transitioning from a configuration to another, a migration cost must be paid. To assess the impact of this cost, in the following experiment we imposed a run-time variation of the required frame rate, leading to a variation in the optimal task/frequency mapping of the application. We considered to have an offline calculated lookup table giving the optimal frequencies/tasks mapping as a function of the frame rate of the application. The system starts with a frame rate of 450 fps. In this case, the system satisfies frame-rate requirements mapping all the tasks on a single processor running at the minimum frequency. As mentioned, scattering task on different processors is not profitable from an energy viewpoint in this case.

A frame rate of 900 fps is imposed at run time that almost doubles the overall workload. The corresponding optimal mapping consists of two processors each one running at the minimum frequency. Due to the workload distribution among tasks, the demodulator will be moved in another core where it can run alone while the other tasks will stay to ensure a balanced condition. Also in this case, further splitting tasks does not pay off because we are already running at the minimum frequency. As a consequence, the system reacts to the frame-rate variation by powering on a new processor and triggering the migration of the demodulator task from the first processor to the newcomer. During migration, the demodulator task is suspended, potentially causing a certain number of deadline misses, and hence, potentially leading to a quality of service degradation. This will be discussed later in this section. Further increasing the frame rate requires the migration of two workers on a third processor.

Whether or not the migration impacts QoS depends on interprocessor queue size. If the system is designed properly, queues should contain a data reservoir to handle sporadic workload variations. In fact, during normal operations, queue empty condition must be avoided and queue level must be maintained to a certain set point. In a real-life system this set point is hard to stabilize because of the discretization and the variability of producer and consumer rates, thus a practical condition is working with full queues (i.e., producer rate larger than consumer rate). When the demodulator task is suspended to be migrated, the queues between the demodulator and the workers start to deplete. Depending on the queue size, this may lead or not to an empty queue condition. This empty condition will propagate to the queue between the workers and the final consumer. If this queue becomes empty, deadline misses may occur.

In Figure 15(a), we show the results of the experiment we performed to evaluate the tradeoff between the queue size and the number of deadline misses due to task migration. The experiment has been carried on by measuring the deadline misses during the whole benchmark execution. The same measurement was performed by changing the size of the queues between the demodulator and the workers and the queue between the worker and the consumer (all having the same size). It can be noted that a queue size of 14 frames is sufficient to avoid deadline misses. Queue occupation during migration is illustrated in Figure 15(b). In this plot, Q_1 is the occupancy level of output queue of the first stage of the pipe(LPF), Q_{21} is the output queue of worker 1, and so on. It can be noted that the temporary depletion of intermediate queues does not cause frame misses. Indeed, the last queue of the pipeline (Q_4), responsible of the frame misses, never deplete.

In order to assess the overhead of migration support, we performed an additional test to quantify the cost of pure code checkpointing without migration. Since this overhead is paid when the system is in a stable, well-balanced configuration, its quantification is critical. It must be noted that checkpoint overhead depends on the granularity of checkpoints that in turn depends on the application code organization. For a software FM Radio application, we inserted a single checkpoint in each task, placed at the end of the processing of each frame. The results are shown in Figure 16(b). In Figure 16(a), we show as a reference the code of the part of the demodulator task with checkpoints. From the plot, we can see that the CPU overhead on the CPU utilization due to checkpoints in

case where no migrations are triggered is negligible, that is less than 1% also when for tasks that are checkpointed with a fine grain as for the producer or the consumer.

7. CONCLUSION

In this paper, we presented the assessment of the impact of task migration in embedded soft real-time multimedia applications. A software middleware/OS infrastructure was implemented to this purpose, allowing run-time allocation of tasks to face the dynamic variations of QoS requirements. Extensive characterization of migration costs have been performed both in terms of energy and deadline misses. Results show that the overhead of the migration infrastructure in terms of CPU utilization of master and slave daemons is negligible. Finally, by means of a software FM Radio streaming multimedia application, we demonstrated that the impact of migration overhead on the QoS can be hidden by properly selecting the size of interprocess communication buffers. This is because migration can be considered as a sporadic event performed to recover from an unbalanced task allocation due to the arrival of new tasks in the system or to the variation of workload and throughput requirements.

As future work, we plan to implement and test more complex task allocation policies, such as thermal and leakage aware. Moreover, we will test them with other multimedia and interactive benchmarks we are currently porting such as an H.264 encoder-decoder and an interactive game. Finally, from a research perspective we are interested in evaluating the scalability of the proposed approach by extending the current FPGA platform to allow the emulation of more processors and finally to implement a heterogeneous architecture exploiting the on-board PowerPC as master.

REFERENCES

[1] K. Eshraghian, "SoC emerging technologies," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1197–1212, 2006.

[2] Cradle Technologies, "Multi-core DSPs for IP network surveillance," www.cradle.com.

[3] "STMicroelectronics Multimedia Processors," www.st.com/nomadik.

[4] ARM Ltd, "ARM11 MPCore," www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[5] L. Friebe, H.-J. Stolberg, M. Berekovic, et al., "HiBRID-SoC: a system-on-chip architecture with two multimedia DSPs and a RISC core," in *Proceedings of the IEEE International Systems-on-Chip (SOC '03)*, pp. 85–88, September 2003.

[6] P. D. Van Wolf, E. De Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: An interface-centric approach," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis, CODES+ISSS*, pp. 206–217, Stockholm, Sweden, September 2004.

[7] C. Sanz, M. Prieto, A. Papanikolaou, M. Miranda, and F. Cathoor, "System-level process variability compensation on memory organizations of dynamic applications: a case study," in *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED '06)*, p. 7, March 2006.

[8] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy, "Selective code/data migration for reducing communication energy in embedded MpSoC architectures," in *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, vol. 2006, pp. 386–391, 2006.

[9] F. Bellosa, S. Kellner, M. Waitz, and A. Weissel, "Event-driven energy accounting for dynamic thermal management," in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP '03)*, New Orleans, La, USA, September 2003.

[10] A. Barak, O. La'adan, and A. Shiloh, "Scalable cluster computing with MOSIX for linux," in *Proceedings of the 5th Annual Linux (Expo '99)*, pp. 95–100, Raleigh, NC, USA, May 1999.

[11] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, 2006.

[12] S. Carta, M. Acquaviva, P. G. Del Valle, et al., "Multi-processor operating system emulation framework with thermal feedback for systems-on-chip," in *Proceedings of the 17th Great Lakes Symposium on Great Lakes Symposium on VLSI (GLSVLSI '07)*, pp. 311–316, Stresa-Lago Maggiore, Italy, 2007.

[13] A. A. Jerraya, H. Tenhunen, and W. Wolf, "Guest Editors' introduction: multiprocessor systems-on-chips," *IEEE Computer*, vol. 38, no. 7, pp. 36–40, 2005.

[14] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini, "Application-specific power-aware workload allocation for voltage scalable MPSoC platforms," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, vol. 2005, pp. 87–93, San Jose, Calif, USA, October 2005.

[15] A. Kumar, B. Mesman, H. Corporaal, J. Van Meerbergen, and Y. Ha, "Global analysis of resource arbitration for MPSoC," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 71–78, 2006.

[16] Z. Ma and F. Catthoor, "Scalable performance-energy tradeoff exploration of embedded real-time systems on multiprocessor platforms," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, March 2006.

[17] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware allocation and scheduling for systems-on-a-chip design," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 898–899, March 2005.

[18] F. Li and M. Kandemir, "Locality-conscious workload assignment for array-based computations in MPSOC architectures," in *Proceedings of the 42nd Annual Conference on Design Automation*, pp. 95–100, 2005.

[19] M. Kandemir and G. Chen, "Locality-aware process scheduling for embedded MPSoCs," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. II, pp. 870–875, 2005.

[20] E. Zayas, "Attacking the process migration bottleneck," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 13–24, 1987.

[21] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration Survey," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[22] D. Pham, "The design and implementation of a first generation CELL processor," in *IEEE/ACM ISSCC*, pp. 184–186, July 2003.

[23] J. Sakai, INOUE, and H. M. Edahiro, "Towards scalable and secure execution platform for embedded systems," in *Proceedings of the Design Automation Conference (DAC '07)*, pp. 350–354, Yokohama, Japan, January 2007.

[24] P. Francesco, P. Antonio, and P. Marchal, "Flexible hardware/software support for message passing on a distributed shared memory architecture," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. II, pp. 736–741, 2005.

[25] S.-I. Han, A. Baghdadi, M. Bonaciu, S.-I. Chae, and A. A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory," in *Proceedings of the Design Automation Conference (DAC '04)*, pp. 250–255, San Diego, Calif, USA, June 2004.

[26] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Power/performance hardware optimization for synchronization intensive applications in MPSoCs," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, March 2006.

[27] uClinux, "Embedded Linux Microcontroller Project," 2007, www.uclinux.org/.

[28] Xilinx Inc, "Xilinx XUP Virtex II Pro Development System," http://www.xilinx.com/univ/xupv2p.html.

[29] W. Thies, M. I. Gordon, M. Karczmarek, et al., "Language and compiler design for streaming applications," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 18, pp. 2815–2822, 2004.