Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study

Stefano Bertozzi[1], Andrea Acquaviva[1], Davide Bertozzi[2], and Antonio Poggiali[3]

[1]ISTI, University of Urbino, 61029 Urbino, Italy

[2]DIE, University of Ferrara, 44100 Ferrara, Italy

[3]DEIS, University of Bologna, 40136 Bologna, Italy

Abstract

With the advent of multi-processor systems-on-chip, the interest in process migration is again on the rise both in research and in product development. New challenges associated with the new scenario include increased sensitivity to implementation complexity, tight power budgets, requirements on execution predictability, the lack of virtual memory support in many low-end MPSoCs. As a consequence, effectiveness and applicability of traditional transparent migration mechanisms are put in discussion in this context. Our paper proposes a task management software infrastructure that is well suited for the constraints of single chip multiprocessors with distributed operating systems. Load balancing in the system is maintained by means of intelligent initial placement and task migration. We propose a user-managed migration scheme based on code checkpointing and user-level middleware support as an effective solution for many MPSoC application domains. In order to prove the practical viability of this scheme, we also propose a characterization methodology for task migration overhead. We derive the minimum execution time following a task migration event during which the system configuration should be frozen to make up for the migration cost.

1. Introduction and Motivation

Interest in process migration was historically raised by the massive deployment of distributed systems (and distributed operating systems in particular) in the parallel computing domain. Run-time transfer of processes between different machines has always been viewed as a way to perform dynamic load distribution, ensure fault resilience, facilitate system administration and enhance data access locality [15].

With the advent of multi-processor systems-on-chip, the interest in process migration is again on the rise both in research and in product development. Beyond traditional objectives, in this new domain process migration can be effectively deployed to maximize energy savings of dynamic voltage and frequency scaling techniques, to facilitate thermal chip management by moving tasks away from hot processing elements and to balance the workload of parallel processing elements [7, 9]. However, the distinctive features of single-chip multiprocessors pose new challenges to the implementation of task migration mechanisms, and this explains why they have not caught on yet in spite of their potential benefits. This is specially true for no-cache-coherent MPSoCs, where each core runs its own local copy of the operating system in private memory [12, 17]. A migration paradigm similar to the one implemented in computer clusters should be considered in this case, with the addition of a shared memory support for interprocessor communication.

MPSoCs are resource-constrained systems with tight power budgets. Therefore, the constraints on implementation complexity already faced in the parallel computing domain are even more stringent here. Moreover, many embedded applications feature real-time constraints, and the performance overhead and unpredictability introduced by transparent task migration might lead to timing violations. Instability conditions related, for instance, to processes migrating back and forth between nodes may further jeopardize execution predictability. For these reasons, less transparent but more controllable migration techniques may be deliberately required in this context, such as user-level or even application-specific migration implementation. Organizing the migration support in reusable run-time libraries can alleviate the drawback of this approach. Lower-level migration might involve intrusive operating system changes and hardware extensions, and research efforts to understand whether this approach pays off are still in the early stage.

In addition, many new generation MPSoCs are designed based on the symmetric multi-processing paradigm [3], where computation nodes are homogeneous and resources are shared, therefore conventional techniques leveraging home dependencies [4] might not simplify task migration. Finally, 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 [1] or the copy-on-reference [18] 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[15]), a frequent number of migration events might seriously degrade application performance in an MPSoC scenario.

In this work we present a lightweight migration mechanism based on code checkpointing and user-level middleware, providing support for a number of load information management strategies and migration activation policies. Our approach to task migration is aware of the potentially high sensitivity of application-perceived performance to task migration overhead and tries to overcome the lack of consolidated architectural support and optimization techniques to alleviate it in the context of MPSoCs. Leveraging our implementation, we characterize the overhead incurred by a user-driven migration strategy and determine the break-even points for the migration to bring benefits to system performance. Finally, we show in a practical case study that a user-driven approach can pay off in terms of reduced state transfers and low-overhead migration.

Our work is structured as follows. Section 2 introduces our hardware/software architecture, while migration methodologies in multiprocessor systems are described in Section 3. Section 4 presents our task migration support and 5 presents experimental results.

2. Hardware/Software Architecture

Depending on the coupling of processors and memory, multiprocessors may be broadly divided into two major categories.

Shared Memory Multiprocessors. In a shared memory multiprocessors, all main memory is accessible to and shared by all processors. The cost for accessing shared memory is the same for all processors, therefore these systems are usually called UMA (Uniform Memory Access) systems. A common communication medium links several memory modules to computational modules consisting of a cache and one or more processor elements, and I/O devices are attached directly to it. In tightly coupled shared memory SMP (Symmetric Multi-Processor) systems, which belong to this category, all the processors run a single copy of an operating system that coordinates global activities [5]. Synchronization is maintained through a cache-coherent low-latency shared memory. A particular category of shared memory multiprocessors is Non-Uniform Memory Access (NUMA). In a NUMA architecture, all physical memory in the system is partitioned into modules, each of which is local to and associated with a specific processor. As a result, access time to local memory is lower than that to non-local memory. NUMA machines may use an interconnection network to connect all processors to memory units, or use cache-based algorithms and a hierarchical set of buses for connecting processors to memory units. In both kinds of machines, I/O devices can be attached to individual processor modules or can be shared.

Distributed Multi-Processors. The individual processing units reside as separate nodes. Each processor runs its own operating system and synchronizes with other processors using messages or semaphores over an interconnect. From a memory access viewpoint, each processor has its own local memory that is not shared by other processors in the system (NO Remote Memory Access - NORMA - Multiprocessors). Computer clusters (CC) are examples of non-shared memory multiprocessors. Workstation clusters usually do not offer specialized hardware for low-latency inter-machine communication and also for implementation of selected global operations like global synchronization or broadcast. In general, NORMA machines do not support cache or main memory consistency on different processors' memory modules. Such consistency is guaranteed only for local memory and caches (i.e., for non-shared memory), or it must be explicitly enforced for shared memory by user- or compiler-generated code.

Embedded MPSoCs. Modern Multiprocessor Systems-on-Chip are usually equipped with local and shared memory, so the access is non-uniform (NUMA). In this work, we target MPSoCs where cache-coherency is guaranteed on private memories where a local copy of the operating system runs for each core (distributed operating system). The system runs a distributed version of the uClinux operating system [16]. On the same on-chip bus, a non-coherent shared memory can be accessed, thus providing support for interprocessor communication. Hardware semaphores can be exploited to implement synchronized accesses to shared resources.

The contributions of this paper can be summarized as follows:
(i) we propose a combined approach to load balancing leveraging intelligent process initial placement and task migration.
(ii) We show the practical viability of a user-managed migration mechanism in MPSoCs, where the lack of transparency may be a desired feature instead of a drawback.
(iii) the overhead of our infrastructure for migration support is accurately assessed via functional simulation, and the break-even points determined so that parameters (e.g., time-outs, thresholds) of different task migration policies can be set to amortize the cost of task migration. To our knowledge, this work is the first to present a detailed implementation of migration mechanisms in the MPSoC domain, going beyond the traditional description of migration proof-of-concept setups.

3. Migration Methodology

Migration has been traditionally studied in both distributed and shared memory multiprocessor systems with the purpose of addressing load sharing problems. When the operating system runs in shared memory, the implicit cost of migrating a task is much lower because process address space is not to be moved. Costs of migration are related to the efficiency of the load sharing policy with respect to resource contention, which is critical in this kind of systems [14]. Migration decisions are taken by looking at OS-maintained process queues in shared memory. Efficient multiprocessor scheduling has been implemented in general purpose OS like Linux 2.6 [8] and has been investigated also for embedded MPSoCs [9, 7, 13].

In distributed systems without shared memory support, like computer clusters, migration policy must be coordinated using messages among nodes. In addition, the implicit cost of migration is larger due to the need of moving the process context over a slower (w.r.t. on-chip or on-board links) communication medium. Finally, the resource management problem is much more complex because resources available in the original node may not be available on the remote node. Examples of such resources are local files and peripherals, that are accessed by processes through system calls. The MOSIX solution [1] is a well known load balancing infrastructure for CCs. From a migration perspective, a MOSIX process has two contexts. A user context (called the *remote*) contains program code, stack, data, memory maps and registers of the process. A system context (called *deputy*) contains description of resources opened by the process. Only the remote is migrated while the deputy represents the so called *home node - UHN*. When a request for a home resource is made by the migrated process, a link layer in the remote node takes care of forwarding that request to the UHN and returns the corresponding results. This abstraction layer is required to provide transparency of migration from the programmer's viewpoint. This approach is impractical for embedded MPSoCs where predictability and controllability are critical issues.

For the embedded MPSoC architecture we target in this work, a new migration paradigm should be explored. In fact, compared to CCs, we can exploit the advantage of the shared memory support to maintain global state information on the processes running on each core by implementing a *process migration table*. However, compared to shared SMP multiprocessors, we pay a higher cost for migration, since process context must be moved. This cost could be critical in an embedded MPSoC system, specially if it is not predictable. For this reason, we propose a lightweight migration support that is more suitable to embedded systems. By following a common programming methodology used in embedded systems, we reduce abstraction layers to increase predictability and reduce migration overheads.

The system is organized with a master processor and an arbitrary number of slave processors. The master core performs admission control and initial process placement on the different slave processors, aiming at an equal distribution of the workload among the slave cores. In our view, this centralized scheduling scheme is viable for on-chip multiprocessors, allows a globally coordinated and hence more efficient placement mechanism and is well suited for many MPSoC architectures where one processor core plays the role of global controller. We developed a user level middleware on top of the uClinux operating system [16], composed by a daemon running on each core and a messaging library providing task termination and remote invocation capability. Processes communicate with the underlaying middleware by means of a library of functions. The migration library allows to implement migration points through code checkpointing. The programmer explicitly specifies the state information (context sensitive data) needed to restart the process on another machine. The middleware library allows to selectively store the logical task state and to restore it at the destination processor. On one hand, only those variables needed to resume execution at the remote site have to be saved and transferred. On the other hand, the user has to explicitly manage the migration process, and must identify the minimum task state to be saved at each migration point.

In this work, we make use of a standard migration initiation mechanism. We assume that when tasks reach candidate migration points, they check whether a migration request has been triggered by the master processor. In practice, a proper field in shared memory is checked. The time between the migration request assertion and the migration point determines a large fraction of the so-called reaction time, which is the migration overhead from the performance viewpoint. Of course, as pointed out in [11], this mechanism gives also rise to a run-time overhead when there is no pending migration request (normal execution) due to the need to check for pending requests. This overhead could be especially noticeable since a task may require frequent migration points in order to reduce the reaction time. However, a number of techniques have been proposed to relieve this overhead, like for instance using debug registers present on most modern processing elements. As proved in the Experimental Results section, in our implementation the run-time checkpointing overhead is negligible because it involves just a few accesses to shared memory.

4. Task Migration Support

In this section we describe the migration support layer that can be used to implement load balancing and task allocation strategies. The support relies upon two main components: i) message passing interface (MPI) and ii) process context save-restore mechanism. Message passing is used to initiate migration requests and to transfer process context, which is saved and restored by means of user level functions. Being completely developed by means of a user level library, the message passing support we developed shows a negligible overhead, thus allowing an efficient migration mechanism. However, any kind of standard MPI support can be used to implement our migration mechanism.

Migration is achieved through a dedicated middleware. An admission control and task allocation daemon (master daemon, M_daemon) is running on the master core. It is responsible of handling task creation requests. The daemon also keeps track of processes born as children of already admitted processes. Master daemon implements also the load balancing strategy and controls migration events in a centralized fashion. It stores the current allocation of processes to cores into a process migration table in shared memory. The load balancing algorithm looks at this table to make migration decisions.

A slave migration daemon (S_daemon) is running on each slave processor. It is responsible to create processes on the local processor upon creation requests generated by the master daemon through messages. Migration is the result of collaboration between master and slave daemons, as shown in Figure 1. When a migration is to be done, the master daemon sets a bit in the migration table corresponding to the selected process entry. When the selected process reaches a migration point, it acknowledges the migration request by clearing the bit. As a consequence, the master daemon triggers the creation of the new process by sending a message to the slave daemon of the destination core. In the experimental results section, we prove the effectiveness of this technique in reducing migration overhead.

In our distributed system architecture, a process always remains within its address space; communication among processes happens through message transfer via a communication channel. When a process in one address space requests a service from another address space, it creates a message describing its requirements, and sends it to the target address space. A process in the target address space receives the message, interprets it and services the request. We implemented a lightweight message passing scheme that exploits shared memory space to implement ingoing mailboxes for each processor

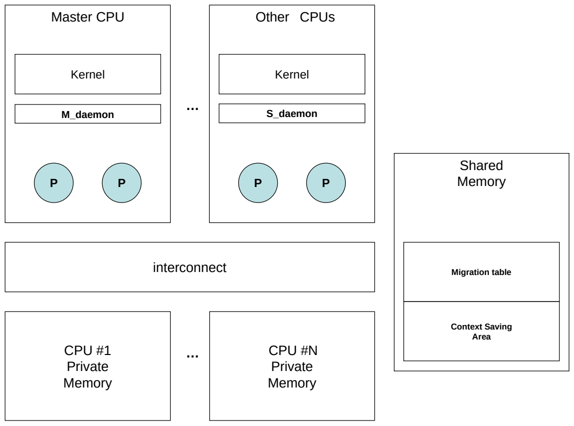


Figure 1. Illustration of the software/hardware organization.

core. We defined a user-level library of functions to be included in the main program that each process can use to perform blocking write ($MPI\_sendmsg(dst\_port, buf, size)$) and read ($MPI\_recvmsg(src\_port, buf, size)$) of data buffers ($buf$) on the mailbox. We defined a mailbox for each core and not for each process to avoid allocation/deallocation of mailboxes depending on process lifetime. It must be noted that the library manages the shared memory completely at the user level. This is obtained through a direct mapping of the shared memory to the process address space.

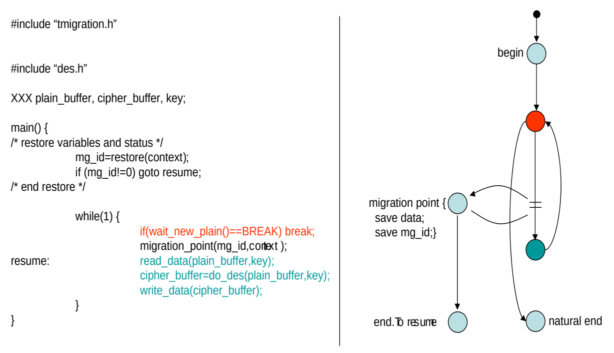
4.1 Checkpointing-Based Migration

As previously mentioned, migration techniques involve saving and restoring the context of a process so that it can be safely executed on a new core. Both in computer cluster and shared memory environments only the user context is migrated. System context is kept either on the home node or in shared memory. We follow the same approach in our migration framework, but in our case system context is destroyed in the starting core (the core where the process was running before migration) and rebuilt on the new core. This allows to avoid the implementation of a link layer (like in Mosix) that impacts predictability and performance of the migrated process, which in our system does not have the notion of home node (UHN [1]).

This is possible only adopting a suitable checkpointing strategy. In fact, by destroying all system context information, also the information concerning opened resources (such as I/O peripherals) in the starting node are lost. As a consequence, the programmer must take care of this by carefully selecting migration points or eventually re-opening resources left open in the previous process life. In this case, a more complex programming paradigm is traded-off with efficiency and predictability of the migration process, as will be showd in the results section. This approach is much more suitable for an embedded context, where controllability and predictability are key issues.

Given that, process migration involves the following steps: i) saving migrating process user context; ii) killing the process; iii) transferring the context to the target processor; iv) launching a replica of the process in the new processor and v) restoring the context.

Checkpointing-based migration technique relies upon modifications of the user program to explicitly define migration and restore points, the advantage being i) the absence of operating system modification and thus a complete portability; ii) predictability and controllability of the migration process. User level checkpointing and restore for migration has been studied in the past for computer clusters [2]. Migration support functions have been im-



```
#include "tmigration.h"


#include "des.h"

XXX plain_buffer, cipher_buffer, key;

main() {
/* restore variables and status */
            mg_id=restore(context);
            if (mg_id!=0) goto resume;
/* end restore */

            while(1) {
                        if(wait_new_plain()==BREAK) break;
                        migration_point(mg_id,context );
resume:                 read_data(plain_buffer,key);
                        cipher_buffer=do_des(plain_buffer,key);
                        write_data(cipher_buffer);
            }
}
```

Figure 2. Example usage of the migration library.

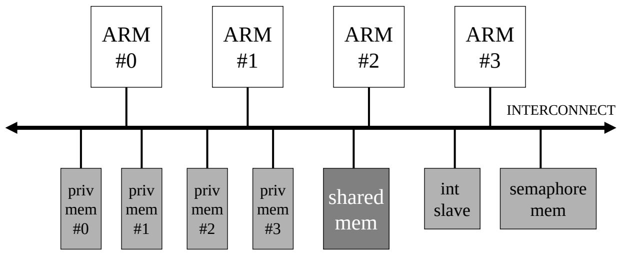


Figure 3. MPSoC platform.

checkpoint frequency	task slow down
1 (1 call every ms)	0,2%
0,2 (1 call every 5 ms)	0,04%
0,1 (1 call every 10 ms)	0,02%
0,05 (1 call every 20 ms)	0,01%

Table 1. Checkpoint overhead

plemented in a *migration library*. An example of utilization of these functions is shown in Figure 2.

When a process reaches a migration point it checks in the process migration table into the shared memory whether there is a migration request issued by the master daemon for the current process. If so, the process stores context sensitive variables, acknowledges the migration by resetting the migration request bit in the shared memory and kills itself. The user explicitly manages a data buffer by taking into account data type and size for a correct restore. The example showed in Figure 2 shows the checkpointed code of a DES (Data Encryption System) application. The application reads `plaintext` and `key` from shared memory, copying them into local buffers, performs encryption in a local memory buffer and finally writes ciphered text back to shared memory.

As the process begins its execution, a restore ($TM\_restore()$) function is called. This function checks if the current process comes from a migration point or not. If yes, it returns the migration point identifier (mg_id) so that the user process can jump to the corresponding checkpoint. The checkpoint identifier has been saved at the migration point in the previous process life. Clearly, it is the responsibility of the user to restore saved data before the jump. This may include also data allocated in the heap. A dedicated function of the migration library can be used to perform allocation and copy of saved data in the newly allocated memory space.

This approach allows the user to migrate only context sensitive variables, instead of migrating the whole context. Although this programming paradigm requires modification of user programs as shown in Figure 2, it may reduce migration overhead and improve controllability and predictability, which are desirable features for embedded time-constrained systems.

5. Experimental Results

We carried out our analysis within the framework of the SystemC-based MPARM simulation platform [10]. Figure 3 shows a pictorial overview of the simulated architecture. It consists of a configurable number of 32-bit ARMv7 processors. Each processor core has its own private memory, and a shared memory is used for inter-processor communication. Synchronization among the cores is provided by hardware semaphores implementing the *test-and-set* operation. The system interconnect is a shared bus.

The software environment consists of the uClinux operating system that we ported to the multiprocessor virtual platform. The operating system image is the Linux 2.6 kernel, patched for no-mmu architectures and a rom filesystem. Each system image (one for each processor) is loaded before the starting of the simulation in the private memory. This way each processor has its own filesystem. Although the $romfs$ should be installed in a non-volatile shared memory, we decided to have a distributed file system for simplicity of porting. By replicating user-level programs in each private memory we can avoid to transfer the code during migrations. In a shared file system implementation, program code would not have to be moved as well. Since uClinux is natively designed to run in a single-processor environment, we added the support for interprocessor communication through the message passing library. We also implemented the support for interprocessor interrupts.

In order to measure performance of our system and characterize migration costs we performed several tests. The first test aimed at evaluating the perturbing impact of the support for migration on the system running processes. Results are shown in Table 1.

Moving a task in our migration infrastructure requires to modify its code adding checkpoint API invocations. In order to evaluate how checkpoint API calls affect task speed in normal execution, we experimentally measured the execution time of a synthetic task with and without checkpoints. Obviously, we assumed there were no pending migration requests to account for the run-time checkpointing overhead. We measured about 2 μs of CPU time to process a checkpoint, and the relative impact on execution speed of the task was calculated based on the checkpoint frequency. We can observe that even with very high and not realistic checkpoint frequencies, the overhead is negligible.

In our framework, there is a daemon process running on each CPU, which periodically wakes up and tests if there are migration requests of tasks to its local processor core. In this case, the daemon forks and generates a new instance of the migrating process. In order to evaluate the leakage of CPU time caused by the daemon execution we experimentally measured the non-forking execution time of the daemon, that is the time used by the daemon to wake up, check that there are no tasks to move and go back to sleep. Non-forking daemons waste about 10 μs of CPU time. The percentage of wasted CPU time with respect to the daemon activation frequency is reported in Table 2, where we assume a 20 ms timeslice under the uClinux kernel.

Again, the impact of daemon execution on system performance is negligible. Referring to Figure 4, we call $T\_migration$ the time elapsed between the call of a taken checkpoint and the end of the process restore at its destination site. $T\_migration$ consists of the following components: (i) the time used by the checkpoint API to check for migration request, to save the context and

Daemon activation frequency	wasted CPU time
1 (1 call every 1 timeslice)	0,05%
0,2 (1 call every 5 timeslices)	0,01%
0,1 (1 call every 10 timeslices)	0,005%
0,05 (1 call every 20 timeslices)	0,0025%

Table 2. Migration daemon overhead

state size (Kb)	time (ms)	time/size (ms/Kb)
1	0,0565	0,0565
8	0,4471	0,0559
16	0,9231	0,0577
32	1,8131	0,0567
64	3,6136	0,0565

Table 3. Migration time overhead

exit ($T\_shutdown$); (ii) the time elapsed waiting for the daemon wakeup ($T\_activation$); (iii) the time needed to rebuild the migrated task, T_reboot.

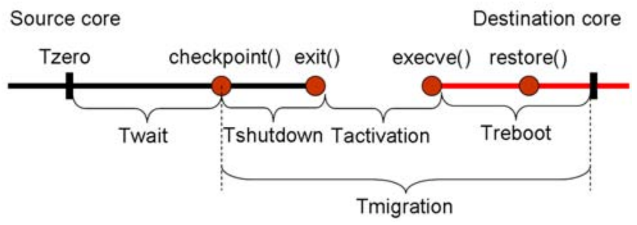


Figure 4. Timeline of the migration mechanism.

$T\_activation$ depends on daemon activation frequency and scheduler behavior, and is therefore not precisely predictable, although it can be assumed to typically vary between 0 and the daemon activation period. $T\_shutdown$ and T_reboot more directly measure the efficiency of our migration implementation, therefore several tests to characterize them have been performed, assuming different state sizes. Results are reported in Table 3, where the sum of these two times is reported.

Experimental results shows that there is a good linear correlation between times. This means that the time needed to move task state heavily dominates the time needed to fork, exec, and restore the migrated process. Please note that the transfer time of state information is spent partly during $T\_shutdown$ (writes to shared memory) and partly during T_reboot (reads from shared memory).

Finally, we built a set of experiments to characterize the effectiveness of migration. We took a characterization approach similar to [14]. The objective is to determine the time after which the migration cost is compensated by a speed-up in system performance. We considered a migration taking place between two processors (CPU A and B). The system starts in an unbalanced state since CPU A executes three processes while CPU B only one. Processes are obtained as instances of the same synthetic program and are therefore independent. At time zero of our experiment the master processor, which is in charge of detecting load unbalances and of managing migration events to balance the system, triggers the transfer of one process running on CPU A to CPU B.

Since load balancing policies are not the focus of this work, we implemented a load balancing policy that quantifies CPU load in terms of number of processes running on a certain CPU and migrates processes away from the overloaded node to a less busy one. More in general, any kind of load balancing policy could be used, such as CPU and memory utilization, open resources, etc., and our infrastructure is in principle able to support all of them. In our experiments, we set the daemon activation period to 1 ms, the checkpoint period to 15 ms and varied the task state size from 0 to 512 kByte.

In Figure 5 we represent the time needed by a single task to get a certain amount of CPU time in four scenarios: i) the task runs standalone on a CPU; ii) the task shares the CPU with another task; iii) the task shares the CPU with other two tasks; iv) the task moves from condition iii) to condition ii), which is the migration case. x-axys values indicate the effective CPU time obtained by

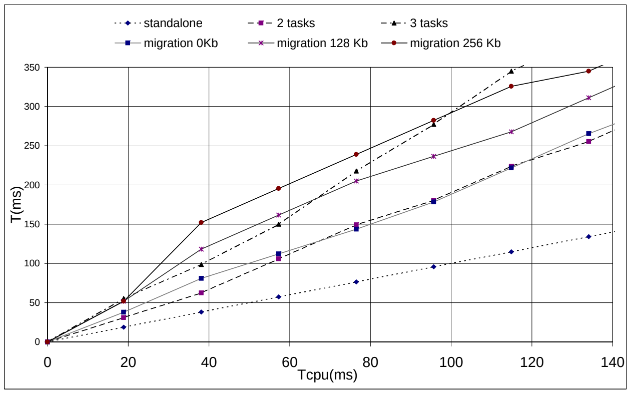


Figure 5. Execution time as a function of CPU time for a single task.

a task, while the y-axys represents the elapsed time needed to get that effective CPU time.

Let us follow the curve referred to migration with zero state. Before migrating, the task executes up to the first checkpoint in an unbalanced context, and might take up to 55 ms to realize that it has to migrate. Then the task is moved to CPU B with a negligible migration overhead, and starts executing in a balanced scenario. Therefore, the relative curve in the plot starts with a slope which is intermediate with respect to the (3,1) and (2,2) curves. Then, between 55 and 100 ms (on the y-axys) the curve has a slope change and tends to almost overlap with the balanced (2,2) curve. Actually, the two curves should not be completely overlapped, because the migration-related curve has to pay the price for starting in an unbalanced scenario. The state to transfer being null, the offset between the curves is minimal, and in the plot it is below the measurement noise.

This is not the case for the migration curves with non-negligible state, which requires a certain amount of time to be transferred to the destination node. These curves end up being parallel, and not overlapped, with the fully balanced (2,2) curve. Moreover, a break-even point does exist between the migration curves and the unbalanced curve where the task runs with other two tasks without migrating. Should the migrating task complete before the break-even time, the migration cost would not be amortized and it would have completed in less time by staying in the original unbalanced configuration. With a 256 kByte state size, the break-even time is about 100 ms (5 uClinux time slices), while with zero state the benefit for task performance is immediate. Such information could be considered by a smart load balancing policy, which could compare the break-even time with the remaining execution time of the task to migrate.

In Figure 6 we repeated the same experiment but we considered the average time needed by all of the four tasks in the system to get a certain amount of CPU time in three cases: i) leaving the system unbalanced (one CPU with 3 tasks and one CPU with one task); ii) leaving the system balanced (2 tasks on each CPU); iii) using migration to balance the system. The interest for this plot stems from the fact that when a migration event takes place, two non-migrating processes start immediately speeding-up, while another one, which was running alone, starts slowing down although it had made more progress. We want to assess how these facts combine together to determine average system performance.

Looking at Figure 6, it is possible to see that migration allows the system to change the slope of its curve, moving to the one of the balanced system. More interestingly, this slope tends to diverge with respect to the curve of the unbalanced system, indicating that migration benefits are more and more significant as the system remains in the new configuration. Although the curves confirm the trend already observed in Figure 5, there is one main difference: for a given state to transfer, the system break-even time (actual CPU time on the x-axys) is lower than the break-even point

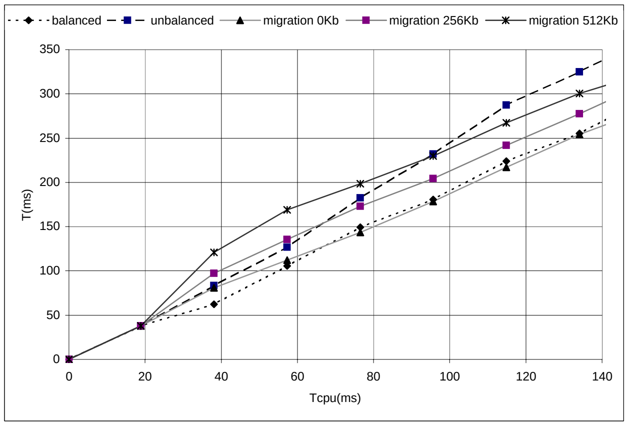


Figure 6. Average execution time as a function of average CPU time for all 4 tasks.

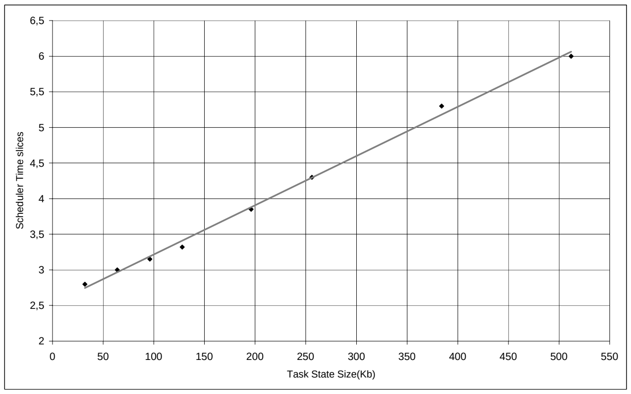


Figure 7. Time needed to amortize the migration overhead.

of the migrating task. This a system level effect, due to the fact that when a task migrates in the considered scenario, two tasks can run immediately at a higher speed, while another task slows down to the same speed featured by tasks in a balanced (2,2) scenario. On average, the system is able to amortize the migration cost more rapidly than the specific migrating task does, because it can rely on the compensating effect of multiple tasks, from a performance viewpoint.

Analyzing data collected by means of multiple simulation runs, an analytical model of the migration-capable system has been developed. The analytical model uses three input parameter: the frequency of the chekpoints in the task code, the daemon activation frequency and the size of the task state. By a point-to-point comparison of the migration curves obtained with the analytical model with those obtained through simulation in Figure 6, we derived for each curve an average error always lower than 2.3% and a maximum standard deviation of 2.44, thus proving the accuracy of the derived model.

We deployed the analytical model to determine the *system configuration freezing time*. Freezing time is the time needed by the new configuration (following a migration event) to start providing performance benefits with respect to the unbalanced configuration. This time depends on the implementation efficiency of the migration infrastructure, since it is tightly related to the migration cost. The freezing time could be used to set the parameters of a load balancing policy, which could prevent changes in the system configuration or new migration requests before the cost for the previous migration event has been amortized. We report in Figure 7 the freezing time for different state sizes, analytically derived for the usual system which moves from a (3,1) to a (2,2) task mapping configuration. The freezing time is expressed as number of scheduler time slices and corresponds to the break-even point between theoretically derived migration curves and unbalanced system curve. The plot points out a linear increase of the freezing time as the state size increases. The distribution of the break-even points around the imaginary straight line is due to the discretization effect of the scheduler, which schedules processes on a time-slice basis.

Finally, we tested our migration system on a real application. We considered a task that performs a DES ECB encryption and modified it in order to be able to use our migration support (as shown in Figure 2). As the task receives plain data and sends cipher data to a collector task, our checkpointing policy allows to migrate the task with an empty task state at the beginning or at the end of its execution. This is definitely less than the data section of the DES task binary executable, that amounts to about 5 KByte. The zero task state is due to the fact that before and after execution the only state the task needs to keep consists of the pointers to the shared memory queues. However, such queues have been statically allocated, therefore their location for the task is hardwired in the code. Migration during task computation does not seem a good option here, since the computation is very short and this would imply to transfer some state variables, which are however of small size although not always easy to determine. This is a typical situation where a user-level technique allows a process to migrate by moving to the destination CPU a lower amount of data with respect to kernel-level techniques.

6. Conclusions

In this work we presented a lightweight migration mechanism based on code checkpointing and user-level middleware. We characterized the overhead of this approach via functional simulation and proved its viability in the context of distributed OS MPSoCs.

7. REFERENCES

[1] Barak A., La'adan O. and Shiloh A., "'Scalable Cluster Computing with MOSIX for Linux,"' Proc. Linux Expo '99, pp. 95-100, 1999.

[2] P. E. Chung, Y. Huang, S. Yajnik, G. Fowler, K. P. Vo, and Y. M. Wang, Checkpointing in CosMiC: a User-level Process Migration Environment," *Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, 1997.

[3] D. Pham et al. "The design and implementation of a first generation CELL processor". *IEEE/ACM ISSCC*, pp.184–186, 2005. July 2003.

[4] F. Douglis and J. Ousterhout, "'Transparent Process Migration: Design Alternatives and the Sprite Implementation,"' *Software-Practice and Experience*, 21(8):757-785, August 1991.

[5] S. Dharmasanam, "'Multiprocessing with real-time operating systems,"' http://www.embedded.com/story/OEG20030512S0080

[6] Intel, "'MultiProcessor Specification,"' http://www.intel.com/design/pentium/datashts/242016.htm

[7] F. Li and M. Kandemir, "'Locality-conscious workload assignment for array-based computations in MPSOC architectures,"' *Proceedings of the 42nd annual conference on Design automation*, pp. 95–100, 2005.

[8] ARM Limited, "'MPCore Linux 2.6 SMP kernel and tools,"' www.arm.com/products/CPUs/linux2_6_smp.html

[9] M.T. Kandemir, G. Chen, "'Locality-Aware Process Scheduling for Embedded MPSoCs,"' *Proceedings of DATE*, pp. 870–875, 2005.

[10] MPARM, http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

[11] V. Nollet, P. Avasare, J. Mignolet, D. Verkest, "'Low Cost Task Migration Initiation in a Heterogeneous MP-SoC,"' DATE, pp. 252–253, 2005.

[12] L. Friebe, H.-J. Stolberg, M. Berekovic, S. Moch, M. B. Kulaczewski, A. Dehnhardt, P. Pirsch, "HiBRID-SoC: A System-on-Chip Architecture with Two Multimedia DSPs and a RISC Core," IEEE International SOC Conference, September 2003, pp. 85-88.

[13] P. Schaumont, Bo. Lai, W. Qin, I. Verbauwhede, "'Cooperative multithreading on 3mbedded multiprocessor architectures enables energy-scalable design,"' DAC, pp. 27–30, 2005.

[14] R. D. Nelson, M. S. Squillante, "'Modeling and Analysis of Task Migration in Shared-Memory Computer Systems,"', MASCOTS, pp. 261–266, 1996.

[15] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, Process Migration Survey, ACM Computing Surveys, September 2000.

[16] uClinux, "'Embedded Linux Microcontroller Project,"' www.uclinux.org/

[17] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, G. Essink, "'Design and programming of embedded multiprocessors: an interface-centric approach,"' CODES+ISSS, pp. 206–217, 2004.

[18] E. Zayas, "'Attacking the process migration bottleneck,"' *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pp. 13–24, 1987.