2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Task Migration for Dynamic Power and Performance Characteristics on Many-Core Distributed Operating Systems

Simon Holmbacka, Wictor Lund, Sébastien Lafond, Johan Lilius
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 FIN-20520 Turku
Email: firstname.lastname@abo.fi

***Abstract*—Spatial locality of task execution will become more important on future hardware platforms since the number of cores are steadily increasing. The large amount of cores requires more intelligent power management due to the notion of spatial locality, and the high chip density requires an increased thermal awareness in order to avoid thermal hotspots on the chip. At the same time, high performance of the CPU is only achieved by parallelizing tasks over the chip in order to fully utilize the hardware. This paper presents a task migration mechanism for distributed operating systems running on many-core platforms. In this work, we evaluate the performance and energy efficiency of an implemented task migration mechanism. This is shown by parallelizing tasks as the performance of a single core is not sufficient, and by collecting tasks to as few cores as possible as CPU load is low. The task migration mechanism is implemented as a library for FreeRTOS using 1300 lines of code, and introduced a total task migration overhead of 100 ms on a shared memory platform. With the presented task migration mechanism, we intend to improve the dynamism of power and performance characteristics in distributed many-core operating systems.**

***Keywords*-Task Migration, Distributed Operating Systems, Many-Core Systems, ARM Cortex-A9**

I. Introduction

Spatial locality of resources provides a measurement of the distance between executing tasks and their resources. The value is proportional to the communication delay introduced between the communicating tasks due to spatial separation. In a many-core Network-on-Chip (NoC) processor, this overhead is clearly noticed as the messages need to propagate along the routing network of the chip. In order to get as small as possible communication overhead when using inter-core communication, the communicating tasks should be placed as close as possible to each other. An optimal mapping of tasks can in a static system be done at compile time, but in a general purpose computer with dynamic task creation, execution times, suspension etc. the tasks should migrate on the chip during runtime to obtain the smallest communication overhead.

High system performance is usually improved by mapping tasks in parallel applications on multiple cores in order to improve the hardware utilization, since multiple processing elements are then capable of executing separate parts of the application in parallel. On the other hand, performance improvements are usually achieved with the sacrifice of energy. In contrast to parallelizing tasks, collecting them to only a few cores allows for sleep state based power management to shut down idle cores and create a more energy efficient system. In both cases, tasks must be movable during runtime in order for the system to be able to optimize for energy vs. performance schemes.

Another important issue caused by the locality of task execution is the thermal balance inside the chip [1], [2]. By changing the location of task execution on the chip, it is possible to avoid thermal hotspots which can gradually wear out the chip. Work has previously been done in terms of task scheduling and heat distribution on the chip. Figure 1 shows an example of how the mapping of tasks affects the thermal gradient of the CPU. The left part of the figure shows a highly parallelized mapping in which the temperature is more evenly balanced, while the right part shows a mapping which concentrates tasks to only a few CPU cores and forms a red hotspot. From the figure it is clear that task mapping

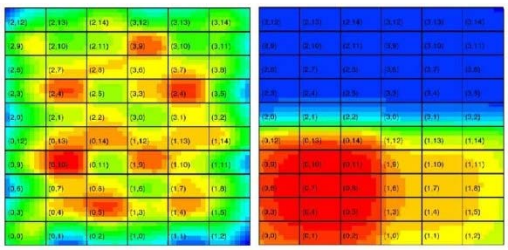

Figure 1. Thermal gradients of many-core chip (Red means hot). The labels (`g`,`m`) illustrate core group `g` and core number `m` [3]

on many-core systems affects the temperature and hotspots on the chip based on the spatial locations of the tasks. This effect will show even more clearly in 3D chips [4] since heat producing elements will spread out in three dimensions.

Task migration is the required technique to re-map tasks on a CPU, and thus enable the aforementioned dynamism. In this paper we present the implementation of a task migration mechanism using checkpoints for homogeneous many-core systems with shared memory. We show how task migration can be used to improve performance and create a more energy efficient system. Furthermore, the overhead of

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.52

executing the task migration is measured and presented in the concluding results.

II. Related Work

Load balancing in conventional Linux [5] kernels on SMP systems has existed for decades. The Linux CFS (Completely Fair Scheduler) aims towards balancing the work as evenly as possible over all processing elements in the system [6]. The Linux load balancer inserts the task into the run queue on a selected core while keeping all references to kernel resources unmodified. We have created a task migration mechanism for, in contrast to the monolithic single kernel Linux environment, a distributed OS [7]–[9] consisting of multiple kernels. The difference between our notion of task migration and conventional load balancing is mainly the transfer of task context. Migrating a task between OS kernels require more context transfer since the kernels are working independently of each other.

Many task migration techniques have been investigated [2], [10]–[12], and the choice is usually dependent on what hardware configuration and what kind of OS is in use. Task migration between physically separate memories require a transfer of both data and program memory area to the new memory location [13]. Heterogeneous task migration techniques has also been considered in previous work, in which program code is modified to support the destination architecture [14]. This introduces several other challenges such as memory alignment, endianness and different instructions.

In contrast to previous work, our task migration mechanism is intended for a distributed OS with multiple kernels running on a homogeneous shared memory architecture with MMU. Because of this architecture, only a pointer to the task handle needs to be physically moved. This means that in contrast to [10] and [2], task size does not influence the migration overhead.

Different notions of task migration and strategies how to initiate task migration have been previously presented. [12], [15]–[19]. Notably in [17], a replication mechanism is used to migrate tasks between cores in a multi-core system. As a task is created on one core, there is also a replica of the same task created on the other cores on which the task is migratable to. The replica tasks are suspended while the original is put in the running state. When migrating the task, the replica task receives its starting point and state and starts running exactly from the point at which the original task was suspended. A task migration mechanism based on re-creation was presented in [12]. The re-creation strategy involves creating a task only on the native core initially. At the execution of task migration, the task is completely copied to the other core's memory and started from the point at which it suspended. After the migration is completed, the original task is suspended and deleted on the first core.

The migration of a task, in our work, is based on only migrating the memory references a task is using. This migration strategy is possible since our target platform includes a shared memory between all cores. No replica of the task is needed, nor any transfer of program memory or the stack. Furthermore, task migration is often implemented in simulator environments. The presented mechanism, in this work, is implemented on a real platform and run on top of a multi-core real-time OS.

III. Platform

For our work, we are considering a many-core platform consisting of homogeneous CPU cores, shared memory and a MMU. These characteristics have been obtained in recent NoC-based many-core platforms [20]–[22], and is therefore a relevant choice.

When using these kind of platforms, the monolithic kernel architecture used in Linux starts to suffer from scalability problems [23]. The reason is mainly because inter-core locking of data structures in the kernel is required [24]. These locks are used to protect data structures from being accessed simultaneously by several cores, but becomes a bottleneck as the number of CPU cores increase. Linux uses, for example, a per-process kernel mutex which serializes calls to `mmap` and `munmap` [23].

Instead we focus on using a distributed operating system as our target platform. This OS structure has been adopted by several research operating systems such as the multikernel structure in Barrelfish [7] or the satellite kernels in Helios [8]. When using a multikernel OS tasks can use core-local kernel calls instead of sharing one big kernel. No core-to-core communication or inter-core spinlocks are therefore required for kernel calls. This OS design scales therefore better for kernel intense applications [9], [25].

Our task migration mechanism has been created explicitly for shared memory distributed operating systems. The platform is assumed to use shared memory for task stack allocation, dynamic memory allocation, program code and for inter-core message passing between tasks. The kernels can run either in shared memory or in private core memory. Figure 2 shows the structure of our system using one OS per core, a certain number of tasks and one task migration mechanism (TM) per core. The kernel of each OS is located, in this case, in separate parts of the shared memory.

A multi-core port of FreeRTOS [26] for the ARM Cortex-A9 MPCore was created as demonstration platform for the task migration mechanism and is freely available [27]. FreeRTOS is a small real-time kernel ported to many popular architectures. The kernel supports a real-time scheduler on top of which applications can be scheduled with hard real-time requirements. This RTOS was chosen due to its simplicity, small overhead and portability.

The platform used was the Versatile Express [28] board equipped with an ARM Cortex-A9 based CA9 NEC [29]

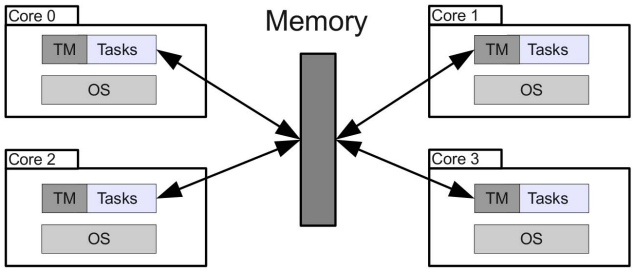

Figure 2. Structure of the platform with shared memory for both OS kernels and core-to-core communication

CoreTile 9x4 quad-core chip running at 400 MHz with 1 GB of DDR2 main memory. One instance of FreeRTOS was mapped on each core as seen in Figure 2, and each instance schedules tasks only on the local core. This is an Asymmetric Multi-Processing (AMP) OS view, which means that each OS instance (with scheduler) is running independent of the others and tasks on different cores do not share the same OS view as in the SMP case.

IV. Task Migration Methodology

The procedure for performing task migration on distributed-kernel operating systems is to safely suspend a selected task and transfer its state and references to another kernel on another CPU core. After the transfer, the task should be able to continue executing from the same point and with the same state it had before it was suspended. Moreover, the task should be attached to the task list of the target kernel and detached from the task list of source kernel. The task should also keep the same priority, name, stack pointer and stack size as it is transferred to the target kernel.

A. Notion of task migration

Because our notion of task migration covers migration between completely independent OS kernels, the task must be ensured a safe state in order to keep the notion of consistency [30] in case I/O or kernel functionality is used. A safe state is a defined state in which the task is guaranteed to not be influenced by any external actors disturbing the transfer of the task state. Arbitrary transfer of a task might issue abrupt terminations of core dependent resources such as I/O communication, which could lead to lost data or unwanted timeout errors. Any usage of resources, kernel functionality, intra-core communication or other non-preemptive functionality must therefore complete or safely be aborted before a migration can occur. Because of this uncertainty in computer programs, a checkpointing mechanism is used to depict points in the program at which is it safe to migrate the task. Checkpointing also decreases the complexity of the task migration mechanism since all migrations are done at completely predictable points.

To make a program migratable, the programmer sets the checkpoints as the program is created. In our model, a checkpoint is set by a simple function call `TASK_IN_SAFE_STATE()`. This point is the dedicated place at which a task can migrate to another core.

The initiation of task migration is up to the system or another task e.g. a power manager. Our task migration mechanism uses an observer task which recognizes scenarios for task migration. The observer can investigate the setup of other cores and make decisions where to move a task from which source core. Optimization algorithms and optimal decision for task migration is, however, not part of this work.

Migration requests are signaled by a request hook in each migratable task, which sets a migration request flag initiated by the observer. This flag is regularly checked by the task in order to reach the safe state if a request is issued. This procedure should be followed in order to achieved task migration in our model:

1) An observer actor in the system requests the migration of *Task 1* to *Core n*
2) The request hook is called in *Task 1* and the `migration_request` flag is set
3) *Task 1* checks the `migration_request` flag in the application, which now is set, and enters the safe state though the function `TASK_IN_SAFE_STATE()`
4) The task migrator is called and *Task 1* is migrated to *Core n*

B. Use-case

Since the checkpoints are placed by the programmer, the system should be analyzed beforehand in order to determine an eventual request lag. The request lag is the time between a migration request has been issued by the observer task until the task reaches the safe state. This time is minimized by placing checkpoints more frequently in the program. Polling the request flag uses only three instructions with the `-O3` flag on the Cortex-A9 CPU, but since a more frequent occurrence of checkpoints slightly increases the overhead, the frequency of the checkpoint placement should be taken into account.

```
void looptask()
{
    while (1) {
        for(i=0 ; i<MAX ; i++){
            foo();      /*Call to function*/
            if (migration_requested)
            /* Go into safe state and suspend*/
                TASK_IN_SAFE_STATE();
        }
    }
}
```

Listing 1. Checkpoint example

Listing 1 shows a simple loop incrementing numbers and calling a function `foo()`. A checkpoint is set after each loop iteration, meaning that the task containing the loop can

be migrated after each loop iteration. The task checks the `migration_request` flag at each loop iteration, which means that in worst case the migration lag is the time of one loop iteration. The responsibility of the programmer is ultimately to set the checkpoints wisely to achieve the least request lag while keeping the checkpoint overhead low.

V. Implementation

A. Overview

The task migration mechanism has been implemented in the C-language specifically for FreeRTOS on 1300 lines of code in total. It consists of a migrator task mapped on each core, which handles the physical transfer and inter-core communication. The FreeRTOS kernel was modified to support the dynamic attachment and detachment of tasks from the task list while keeping the tasks' state consistent. The complete modification to FreeRTOS was implemented using 110 lines of C-code. This section describes the most important part of the implementation, namely how the memory is used between the kernels and how the task state is transferred across cores.

B. Virtual memory mapping

Virtual memory is used to replace the physical memory layout from the system, and replace it with a virtual representation which is easier to operate against.

In our model, each core C contains one kernel K. Each kernel uses the same virtual memory space, which means that each kernel has the same memory view; this is seen in the left part of Figure 3. This mirrored view abstracts away the fact that the kernels actually execute in separate memory locations in the physical memory (right part of Figure 3). With this setup, all tasks can issue kernel calls with the same address of reference. This increases the OS scalability since no inter-core kernel locks are required, but tasks always call the local kernel. An example is later shown in Section V-C.

Figure 3 shows the memory layout for the quad-core CPU earlier mentioned. The tasks running on a kernel are dealt a specific memory location GVM (Globally Visible Memory). This location depends on at what core (in which kernel) the task is created. For example a task created on core $C1$ will allocate its stack space in $GVM1$. The GVMs do not, on the other hand, provide the same virtual memory view. This is because tasks should be able to switch kernel to be scheduled on. The state of the task must always refer to the absolute memory address space to be kept consistent independent on what core the task is running on. For example consider a task $T1$ with stack memory $GVM1$ migrating from $C1$ to $C2$. In order for the state to be kept consistent, $T1$ must still keep the references to $GVM1$ even though it is moved to $C2$. If the stack pointer of $T1$ should now point to $GVM2$ instead, the content of the stack would not be kept consistent without physically moving the whole stack to $GVM2$. Since we assume a shared memory

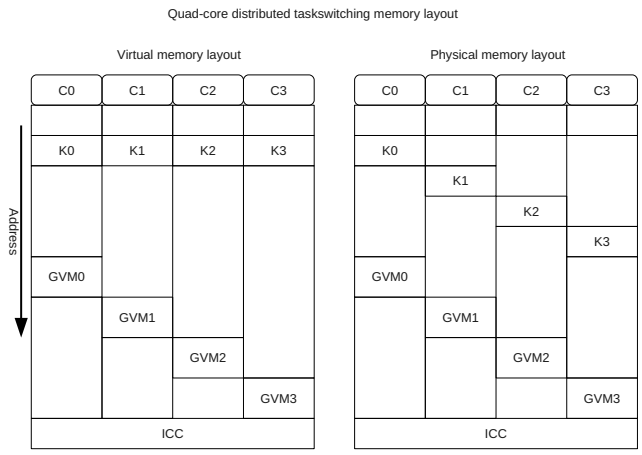

Figure 3. Memory layout for a Quad-Core CPU

architecture, it is possible to only pass references to the corresponding GVM instead of a complete transfer. The migration overhead will also be much smaller since less information is moved.

After $T1$ has migrated from $C1$ to $C2$ the pointer to $GVM1$ is passed to $K2$, which updates the local task list. If $T1$ is deleted on $K2$, $K2$ sends a message to $K1$ in order to free the allocated memory $T1$ was using in $GVM1$. Generally if a core issues a delete command on a task created on a non-local core, the delete request propagates back to the origin of the task in order to free the memory allocated by the task.

The memory reserved for core-to-core communication (ICC) is a statically allocated area in the highest part of the memory and is used to pass messages between cores. Communication with shared variables will not be affected by the task migration since the address of the shared variable is located in the globally visible memory part and can thus be accessed by any task independently of what core the task is scheduled on.

C. Context transfer

The state transfer in a task migration is more complex than on Linux SMP systems since the task is moved to a new OS instance while keeping its state constant. The state of a task is any entity stored in the task that determines the execution of the task, which during runtime is modifiable. Besides the name and function pointer to the task itself, the following context is transferred during a task migration:

1) Stack state: The stack is initially created in a certain GVM depending on which core a task is created on. Upon task migration the location pointer to the stack is transferred to the target core. The stack itself is not physically moved since we assume that task stacks are located in Globally Visible Memory.

2) Heap state: All dynamically allocated variables are stored in the heap. Similarly to the stack, the heap variables are stored in the GVM on the core the task using the variables was created on. As the task is migrated, all dynamically allocated variables pass their reference pointers to the new core, which means that no data is physically moved other than the pointers similarly to the stack state.

3) Function references: The motivation behind using the distributed kernel is to create a scalable OS for many-core architectures. An important functionality in this architecture is to enable core-local kernel calls. All tasks should therefore only use the local kernel for kernel calls (provided that the kernel in question can provide the requested functionality).

Consider the system shown in Figure 4: a task T1 is created on core C1 and uses kernel K1 for kernel calls. After the task migration to C2, T1 should update its kernel reference to K2 in order to use the core-local kernel calls.

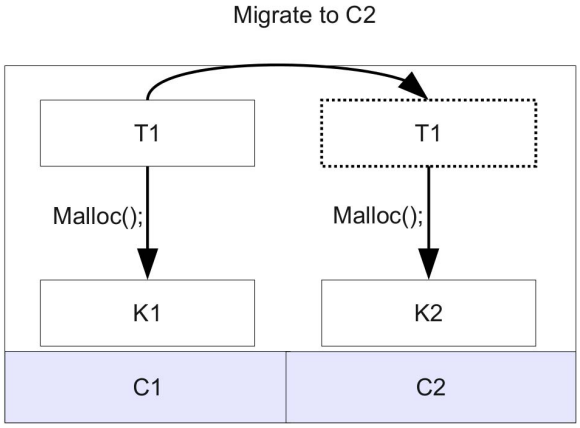

Figure 4. Update of kernel reference after task migration

To obtain this functionality, we have implemented relinkable elf binaries for FreeRTOS. All tasks are compiled to distinct elf binaries and are linked together with the kernel on a core. During the task migration, the link between the task and the kernel is broken and re-linked with the kernel on the target core. The memory references to the kernel do not change, since the virtual memory ensures the same memory outlook of all kernels (as has been shown in Figure 3). In this way, the tasks do not need to keep track on what core they are mapped on – making the programming completely location transparent for the programmer.

4) Inter-task communication: Tasks communicating with shared memory will retain the memory location used for the communication without any modifications. This is possible because all tasks can access any GVM at any time. The migrated task will, after the migration, keep the address to the shared memory at which the communication was taking place.

Communication with message passing between tasks is a part of future work. This functionality is non trivial since the message passing mechanism is dependent on local or non-local communication. Non-local communication requires explicit core-to-core communication because the communicating tasks are located on different cores, while local communication should only use the message queue mechanism in order to not introduce unnecessary overhead.

VI. Evaluation

The evaluation setup consists of four identical video playing tasks mapped on the ARM quad-core platform described in Section III. Each task plays a video with a certain resolution. The frame rate (fps) is measured with a regular interval in order to evaluate the performance of the video. We evaluated the system for both performance and energy efficiency in order to demonstrate the improved dynamism of the system. The overhead introduced by the task migration mechanism, was also measured to give the final conclusions.

Since the evaluated platform (described in Section III) is designed as a low power platform and does not per se suffer from thermal hotspots, and because it does not provide per-core temperature measurement, we do not include temperature measurements in our experiments.

A. Performance evaluation

The first test was run to show how the performance of the video tasks is boosted by parallelization; namely by migrating tasks to all available cores. Our goal for this test is to obtain a stable video playback (25 fps) for all four videos. Initially, four large resolution videos were mapped on Core0 on the ARM platform. After measuring a low frame rate, three of the video tasks were migrated to other cores, resulting in a system with one video task per core. The video task included one safe state point per frame, which resulted in 11 additional lines of source code in the application.

Figure 5 shows the execution of the test. At the beginning of the test (time=[1 to 3]), all videos play with a frame rate between 6 and 14 fps, which is too low for user satisfaction. At time=3, the first video task (Video1) migrated to another core, which results in increased frame rate for Video1 at time=5, and also higher frame rate for the remaining video tasks on Core0. Similarly at time=6, Video2 is migrated and thus achieves satisfactory frame rate at time=9. Finally Video3 migrates at time=13 and is fully stable at time=14. The high peak of the non-migrated task (Video4) is due to the frame dropping mechanism used to compensate for low frame rate in the beginning of the test.

All videos are mapped on their own dedicated core at time=14, and all videos tasks are executed completely in parallel. By parallelizing tasks and better utilizing the hardware, all videos increase the frame rates to roughly 25 fps and keep a stable playback after time=18.

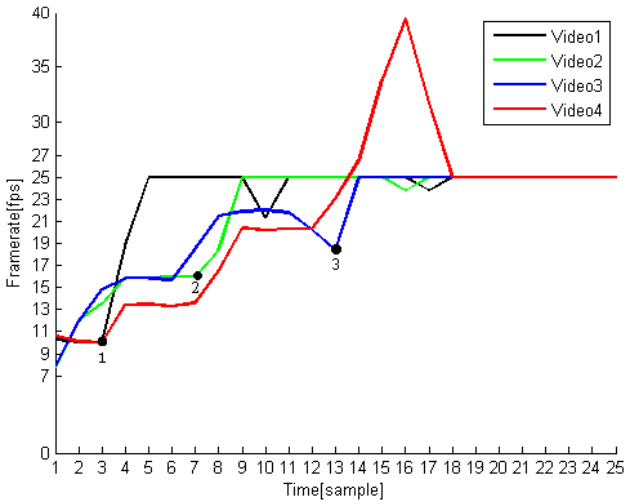

Figure 5. Frame rates for large resolution videos migrated to four cores. Task migration initiated at time=3. Points 1,2 and 3 shows migration points in time

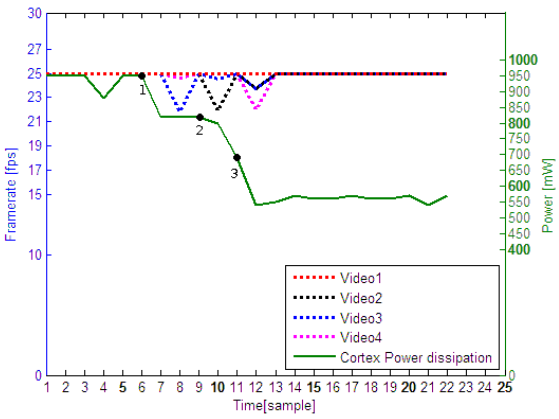

Figure 6. Frame rates for small resolution videos migrated to one core and Cortex-A9 power output. Task migration initiated at time=7. Points 1,2 and 3 shows migration points in time

B. Power evaluation

The second test was used to show the energy efficient potentials task migration can give rise to. In this setup, we started by having four small resolution videos mapped one on each core on the same quad-core ARM platform. The power output was measured directly from an internal register in the Cortex-A9 MPCore chip. Our goal with this test is to minimize the power dissipation as much as possible while keeping the frame rate stable. From the starting point of having four parallel videos, all video tasks were migrated to one core (Core0). The remaining idle cores (Core1, Core2, Core3) were shut down since no tasks were mapped on them after the migration.

Figure 6 shows the frame rate and related power output for the test. The curves related to videos are plotted against the frame rate axis and the power dissipation curve against the power axis. The test starts with having video playbacks with frame rates of 25.0 fps which is sufficient to the user, however the small resolution of the video format would allow collecting all videos to a single core. At time=7 in Figure 6, the migration mechanism starts to collect one video task at the time to Core0, and completes this operation at time=12.

The figure shows how the power dissipation initially starts at about 900 mW and decreases to roughly 550 mW after the tasks have migrated to Core0. This clearly affects the energy consumption of the platform since the power output is reduced with 40 %. Figure 6 also shows that Core0 alone is able to keep a sufficient frame rate of 25 fps for all four videos during normal playback. At the time of migration, the migrated video tasks occasionally measure a slight frame rate drop, but the overall quality is still kept sufficient.

C. Migration overhead

The migration of tasks introduces a slight overhead due to the moving of data and detachment and attachment of tasks to the OS scheduler. A simple evaluation to measure the total overhead was run in order to demonstrate the feasibility of migrating a streaming task such as a video player without noticeable interruption. Our defined overhead of a task migration is measured as the time between *the suspension of the task on the sender core and the resume of the task on the target core*. This overhead was measured in three parts:

1) Time of task detachment and the activation of inter-core communication
2) Time for moving data over the inter-core channel
3) Time between the arrival of inter-core data and the attachment of the task

Part 1 and 3 was measured with a simple tic-toc timer which counts the elapsed time between tic and toc in OS ticks, which is easily converted into Milli seconds. Part 2 (inter-core communication delay) was measured with a provided inter-core communications library for FreeRTOS. This library uses inter-core interrupts to synchronize the time between two communicating inter-core channels, since the clocks of different cores are not identical. Measurements were run several times and the deviation of the results were zero for all three parts. The total overhead is presented in Table I. The table shows that Part 3, which consisted of attaching the task to the new OS scheduler, introduced the

Table I
OVERHEAD MEASUREMENTS

	Part 1)	Part 2)	Part 3)	Total
Time	17 ms	38 ms	45 ms	100 ms

largest overhead when migrating one of our video tasks. A reason for this is due to increased L1 cache misses as the task is moved to a new core with a cold L1 cache. To reduce this overhead, the system could be set to warm up the cache lines before the task continues the execution. An other approach to decrease the overhead would be to enable the L2 cache (which in the experiments was off). Detaching the task (Part 1) had the least overhead with only 17 ms, and the physical data transfer 38 ms. The data transfer included a provided inter-core communications protocol binary, which enabled us to analyze clock synchronization but overhead analysis of the internal mechanism was not possible due to closed source.

A fair comparison with related mechanisms is not directly straight forward since methodologies, platforms and the notion of task migration usually differ. For example a NUMA architecture – with non-uniform access time to memory – would show a larger delay when migrating the task to a destination located further away. Moreover, with the introduced overhead, the video playback was – to the user – very smooth and the slight freeze during the task migration of a 25 fps video was hardly noticed. The task migration mechanism introduced in total 160 additional bytes to the application, which corresponded to 40 additional instructions to run (with gcc -O3 flag).

VII. Conclusions

This paper has demonstrated the dynamic performance and power properties of task migration for distributed many-core operating systems. A task migration mechanism has been presented and evaluated for maximum system performance due to parallelization of tasks, and power conservation as a result of utilizing enabled hardware as much as possible while keeping idle hardware shut down. We have evaluated the task migration mechanism with a set of video tasks representing a use-case in which the notion of quality of service is of importance. In the experiments, we have shown a typical example in which insignificant performance sacrifice is traded for substantial energy savings.

A key point of distributed operating systems is to utilize the spatial location for executing different parts of the OS efficiently. In our case study, we have shown how the location of a CPU intense video task directly influences the performance and the power dissipation. In larger and more diverse NoC many-core systems, different parts of the distributed OS can be mapped and migrated close to its related hardware depending on its functionality. Network intense parts should for example be mapped closest to the physical network interface and memory intense parts should be mapped close to the memory controllers.

With more and more diverse and complex hardware, more intelligent software solutions are required in order to fully utilize the potential of the hardware. The distributed OS design together with task migration provides better location dependent task mapping, and is a step towards this goal.

VIII. Future work

Tasks using inter-task communication need to update the references for passing messages if one of the communicating task changes its spatial location. Core-to-core communication must be explicitly pointed to tasks located on different cores, while tasks on the same core can use simple message queues. In order to hide these details from the programmer, we have developed a lightweight component framework for for real-time systems [31]. With the framework, the developer is able to set-up specific communication interfaces used for inter-task communication and could be used to rise the level of abstraction for the inter-task communication. We intend to integrate the possibility of task migration into the framework, which simplifies the updating of communication references.

Also, currently the task migration mechanism does not support load balancing. Load balancing requires a decision making mechanism capable of deciding what task should move to what target core, and at what time. The task migration mechanism should also be evaluated on a larger many-core platform in order to show the real benefits of spatial re-location of tasks.

Acknowledgment

This work has been supported by the Artemis JU project RECOMP: Reduced Certification Costs Using Trusted Multi-core Platforms (Grant Agreement number 100202).

References

[1] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii, "Adaptive task migration policies for thermal control in mpsocs," in *Proceedings of the IEEE 2010 Annual Symposium on VLSI*, vol. 1. Ecole Polytechnique Fédérale de Lausanne and Politecnico di Torino, July 2010.

[2] F. Mulas and D. Atienza, "Thermal balancing policy for multiprocessor stream computing platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 1870–1882, December 2009.

[3] E. Musoll, "Hardware-based load balancing for massive multicore architectures implementing power gating," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, no. 3, pp. 493 –497, march 2010.

[4] K. Matsumoto, S. Ibaraki, M. Sato, K. Sakuma, Y. Orii, and F. Yamada, "Investigations of cooling solutions for three-dimensional (3d) chip stacks," in *Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010. 26th Annual IEEE*, feb. 2010, pp. 25 –32.

[5] M. T. Jones, "Inside the linux scheduler," Jun 2006. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-scheduler/

[6] ——, “Inside the linux 2.6 completely fair scheduler,” December 2009.

[7] A. Baumann and P. Barham, “The multikernel: a new os architecture for scalable multicore systems,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 29–44.

[8] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, “Helios: heterogeneous multiprocessing with satellite kernels,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 221–234.

[9] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, “Corey: an operating system for many cores,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57.

[10] T. J. E. Engin, “Bag distributed real-time operating system and task migration,” *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 9, no. 2, 2001.

[11] P. K. Saraswat, P. Pop, and J. Madsen, “Task migration for fault-tolerance in mixed-criticality embedded systems,” *SIGBED Rev.*, vol. 6, no. 3, pp. 6:1–6:5, Oct. 2009.

[12] 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 conference on Design, automation and test in Europe: Proceedings*, 3001 Leuven, Belgium, Belgium, 2006, pp. 15–20.

[13] J. B. Armstrong, “Dynamic task migration from simd to spmd virtual machines,” in *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, ser. ICECCS ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 326–.

[14] M. DeVuyst, A. Venkat, and D. M. Tullsen, “Execution migration in a heterogeneous-isa chip multiprocessor,” in *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. IEEE Computer Society, 2012.

[15] A. Aguiar, S. J. Filho, T. G. dos Santos, C. Marcon, and F. Hessel, “Architectural support for task migration conserning mpsoc,” in *SBC*, 2008.

[16] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, “Assessing task migration impact on embedded soft real-time streaming multimedia applications,” *EURASIP Journal on Embedded Systems*, no. 9, January 2008.

[17] L. G. adn S. Layouni, M. Benkhelifa, F. Verdier, and S. Chauvet, “Multiprocessor task migration implementation in a reconfigurable platform,” in *International Conference on Reconfigurable Computing and FPGAs*, 2009.

[18] E. Brio, D. Barcelos, and F. Wagner, “Dynamic task allocation strategies in mpsoc for soft real-time applications,” in *Proceedings of the conference on Design, automation and test in Europe*, IEEE Council on Electronic Design Automation and EDAA : European Design Automation Association. ACM, 2008, pp. 1386–1389.

[19] P. Smith and N. C. Hutchinson, “Heterogeneous process migration: The tui system,” *Software — practice and experience*, vol. 28, no. 6, pp. 611–639, March 1998.

[20] S. Potluri, K. Tomko, D. Bureddy, and D. K. Panda, “Intramic mpi communication using mvapich2: Early experience,” *Texas Advanced Computing Center (TACC)-Intel Highly Parallel Computing Symposium*, April 2012.

[21] J. Howard, S. Dighe, Y. Hoskote, and Vangal, “A 48-core ia-32 message-passing processor with dvfs in 45nm cmos,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, feb. 2010, pp. 108 –109.

[22] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, “On-chip interconnection architecture of the tile processor,” *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[23] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of linux scalability to many cores,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[24] A. Kleen, “Linux multi-core scaleability,” in *Linux Kongress 2009*, Dresden, 2009.

[25] D. Wentzlaff and A. Agarwal, “Factored operating systems (fos): the case for a scalable operating system for multicores,” *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 76–85, April 2009.

[26] R. Barry, *FreeRTOS Reference Manual: API functions and Configuration Options*, Real Time Engineers Ltd, 2009. [Online]. Available: http://http://www.freertos.org/

[27] D. Ågren. Freertos cortex-a9 mpcore port. Åbo Akademi University. [Online]. Available: https://github.com/ESLab/FreeRTOS---ARM-Cortex-A9-VersatileExpress-Quad-Core-port

[28] ARM. (2011) Coretile express a9x4 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.dui0448e/DUI0448E_coretile_express_a9x4_trm.pdf.

[29] ——. (2009) Cortex a9 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e/DDI0388E_cortex_a9_r2p0_trm.pdf.

[30] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana, “Tackling consistency issues for runtime updating distributed systems,” in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, april 2010, pp. 1 –8.

[31] R. Slotte, “A lightweight rich-component framework for real-time embedded systems,” Master’s thesis, Åbo Akademi University, January 2012.