Demystifying the Cost of Task Migration in Distributed Memory Many-Core Systems

Marcelo Ruaro, Fernando G. Moraes
FACIN - PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
marcelo.ruaro@acad.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Task migration plays a major role in the implementation of runtime adaptive techniques for many-core systems, as thermal and power management, load balancing, QoS, and fault tolerance. A fast task migration protocol contributes to implementing self-adaptive techniques with low overhead. State-of-the-art proposals still have limitations, with an important impact on the applications' execution time due to the latency to migrate tasks. Aware of the number of simultaneous task migrations required by self-adaptive techniques, this work proposes a low latency tasks migration protocol for many-core systems with distributed memory hierarchy. Our technique eliminates checkpoints, task code replication, enables simultaneous task migrations even in tasks of the same application and parallelizes the task migration with the application execution. These features induce a low latency in the task migration, demystifying the cost to adopt task migration in distributed memory systems. Results compare the proposed approach to the related works.**

Keywords— Task migration, many-core, MPSoC

I. INTRODUCTION

Self-adaptive techniques, as thermal and power management, load balancing, Quality of Service (QoS), and fault tolerance commonly use task migration to perform runtime management in many-core systems. In thermal and power management [1], task migration can be employed together with DVFS to distribute the tasks according to the performance of the cores. For load balancing and aging control [2][3], task migration can help to distribute the workload of the system, by migrating several tasks simultaneously to cores with lower utilization. For QoS [4], task migration may be used to reserve CPU resources for real-time tasks. For fault tolerance [5], task migration can be used to move tasks running in faulty Processing Elements (PEs).

Proposals related to task migration addressing many-core systems with distributed memory have at least one of the following features: (*i*) use of checkpoints, which requires source code annotation; (*ii*) task replication, with a replica of the task in one or more PEs that can receive the task to migrate (wasting memory); and (*iii*) significant migration latency.

The *goal* of this work is to propose a task migration technique between PEs having the same ISA (Instruction Set Architecture), for many-core systems with distributed memory hierarchy. The *contribution* of this work is a protocol with a lower latency compared to the related works. Further, this proposal eliminates the need of checkpoints, not requiring task replicas, and enable to migrate tasks of the same applications simultaneously.

The key point to reduce latency comes from a two steps procedure. First, the *text* memory section (object code) of the task to migrate, T_M, is migrated separately of the dynamic memory sections: *data*, *bss*, and *stack*. While the T_M's *text* is transferred from the source PE (S_{PE}) to the target PE (T_{PE}), T_M keeps running at the S_{PE} being blocked only during the dynamic memory migration. We assume a DMNI (Direct Memory Network Interface) module [6] to copy the *text* section while the CPU keeps running T_M. The second step comes from an inter-task synchronization protocol, which not migrates the messages produced by T_M. The synchronization is performed on-demand after the migration, helping to reduce the migration data volume while ensures no message loss.

II. RELATED WORK

Table I summarizes the main features of the related works in task migration for distributed memory many-core systems. The 2nd column details the adoption of checkpoints. Checkpoints simplify task migration because they statically define migration points where the task context is safely saved. These checkpoints are inserted on task's source code, being difficult in practice to the application developer define safe states to migrate. The 3rd column addresses the inter-task synchronization. El-Antably et al. [8] and Fu et al. [9] transfer all pending messages during the migration process (FIFO copy), Canella et al. [10] wait for the consumption of all messages to execute the migration (FIFO release), and Saint et al. [12] use the communication primitives as checkpoint to enable the migration. Our work adopts an approach similar to the theoretic proposal of Munk et al. [7]. The synchronization occurs on-demand according to the messages request by the consumer tasks. Subsection IV.A details this mechanism.

TABLE I. COMPARISON OF TASK MIGRATION WORKS.

Proposal	Check.	Inter-task sync	MMU	Migration
Munk et al. 2015 [7]	no	Message forward	no	recreation
El-Antably et al. 2015 [8]	yes	FIFO copy	no	replication
Fu et al. 2013 [9]	no	FIFO copy	no	replication
Canella et al. 2012 [10]	no	FIFO release	no	replication
Jahn et al. 2011 [11]	no	N/A	yes	pre and post copy
Saint et al. 2008 [12]	yes	Send / Receive	no	recreation
This proposal	*no*	*Message forward*	*no*	*recreation*

The 4th column of Table I evaluates the use of an MMU (memory management unit) in the migration process. The adoption of MMU simplifies the migration process since the operating system handles virtual addresses, at the cost of additional hardware. As most of the works, our proposal does not adopt an MMU. The memory management is simplified by using a paged memory organization managed by the OS (operating system).

The 5th column presents how the migration process is executed. Task replication [8][9][10] keeps a task replica at different processors. This procedure reduces the task migration latency because the *text* section is not transferred, but incurs in memory overhead. Task recreation [7][11][12] stops the task execution, transfer the *text* and the context to the target PE, and then resume the execution. Aided by an MMU, Jahn et al. [11] adopt a mixed mechanism, which pre-copies the most frequent data memory sections before the migration of the task context, and transfers the remaining memory sections by requests sent to the S_{PE} after the task migration.

978-1-4673-6853-7/17/$31.00 ©2017 IEEE

III. System Model

Fig. 1(a) presents the PE Architecture. The PE contains one CPU, a NoC router, a dual-port scratchpad local memory, and a DMNI module. The DMNI module [6], merges the functionalities of an NI (Network Interface) and DMA (Direct Memory Access) modules. The adoption of a DMNI module optimizes the process of transfer data between the NoC and the local memory. This proposal is not dependent on the DMNI module. Other platforms can implement NI and DMA separately to manage memory transaction with the NoC.

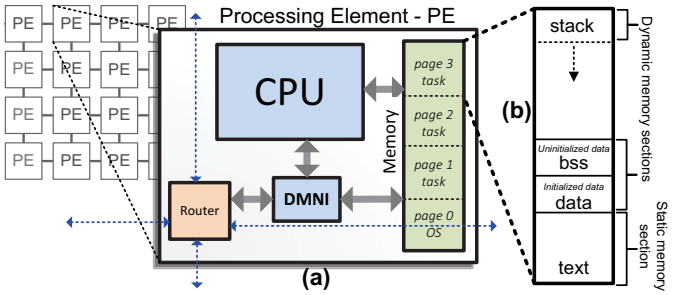

Fig. 1. (a) Processing Element Architecture. (b) Memory page organization.

A small OS runs at each PE. The OS provides the communicating API for tasks by implementing two MPI-like primitives: *Send* and *Receive*. The *Receive* is a blocking primitive called by the consumer tasks. The *Receive* generates a MESSAGE_REQUEST to the PE of the producer task, which deliveries the data by sending a MESSAGE_DELIVERY packet. When a task calls the *Receive*, it goes to a waiting state until the reception of the message. A scheduler enables multitask execution, and the OS assigns one task per page. The number of pages is a design-time parameter. The OS is loaded on page 0 at the system initialization. The application's tasks use the remaining pages.

The OS keeps a message FIFO for each task named *pipe*, which stores the task's messages produced but not requested yet. The *pipe* is managed by the OS of the producer task. When an MESSAGE_REQUEST arrives at the producer OS, it searches by the message in the *pipe*, if it is found, the message is delivered to the consumer task. If no message is found, the request is stored and when the producer task calls the *Send* primitive the produced message is directly forwarding to the consumer task, without storing it in the *pipe*.

Fig. 1(b) details the memory structure for each page. The memory has a *text* section used to store the task's object code. It is possible to migrate this section while the task is executing because the processor only read it, unlike the dynamic memory sections (*bss*, *data*, *stack*), which are changed during the task execution. This memory structure is generic, defined by the compiler.

IV. Proposed Task Migration

The task migration proposal is implemented as an OS service. Fig. 2 presents the task migration protocol. The task migration is triggered when a migration order arrives at the S_{PE} (the generation of this event is out of the scope of this work). The OS handles this order by configuring the DMNI to transfer the T_M *text* section (event 1). The *text* section is sent to the T_{PE} through a MIGRATION_CODE message (event 2). After the DMNI configuration, T_M continues its execution up to reach a *safe* state, where the dynamic data memory sections can be safely migrated.

Safe state definition. Assuming a simple scheduler, a given task may be in 3 states: (*i*) *ready*, the OS can schedule the task to execute; (*ii*) *waiting*, the task is blocked waiting a message delivery; (*iii*) *running*, the task is running changing the dynamic memory sections. *A safe state is defined when T_M is in the ready state.* If migration occurs during the *running* state, the dynamic data memory sections are in use, corresponding to an unsafe state. If migration occurs during the *waiting* state, the delivery of the requested message will arrive at old PE of T_M, inducing a message loss. The delay to start the data migration is a function of the scheduler time slice and the time to the producer task to deliver the requested message.

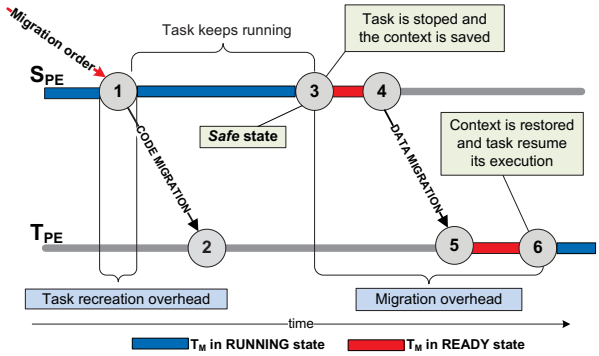

Fig. 2. Overview of the proposed task migration protocol.

Event 3 in Fig. 2 represents T_M in a safe state. The task stops, and the OS saves the context, transferring the CPU registers and the data memory sections to the T_{PE} by a DATA_MIGRATION message (event 4). At event 5, the DATA_MIGRATION arrives at T_{PE}. At event 6, the OS of the T_{PE} restore the T_M context (by copying the DATA_MIGRATION payload to T_M page), and T_M goes to the *ready* state. The overhead of the proposed protocol comes mainly from event 3 to event 6, which varies according to the size of the data section, and the interconnection mean. We detail the overheads in Subsection V assuming a 2D-mesh NoC as the interconnection infrastructure.

A. Inter-Task Synchronization

A key feature of the task migration protocol is the synchronization of the messages exchanged between tasks. Each OS has a task *location table*. The *Send* and *Receive* primitives use this table to find the address of the communicating tasks. When T_M migrates, this table must be updated in all PE with tasks that send and receive messages to/from T_M. The update is executed on-demand according to the following rules:

Rule 1: when T_M migrates to T_{PE}, all produced messages by T_M stay in the *pipe* of S_{PE}. After T_M migration, the consumer tasks continue sending MESSAGE_REQUESTs to S_{PE}. If there *is a message in the pipe*, the message is removed from the *pipe* and delivered to the consumer task. Fig. 3(a) presents this scenario. At event 1, task A migrates from PE0 to PE1. Messages produced by task A, to tasks B and C, stay in the pipe of PE0. At event 2, task B sends a MESSAGE_REQUEST to PE0 (old address of task A). As there is a message in the *pipe* to task B, this message is delivered to task B (event 3).

Rule 2: If there is *no message for the consumer task* in the *pipe* of S_{PE}, the MESSAGE_REQUEST is forwarded to T_{PE}, and T_{PE} deliveries the requested message to the consumer task. The OS of the S_{PE} has the new address of T_M, enabling to forward the MESSAGE_REQUEST. Fig. 3(b) presents this scenario. Task B sends a MESSAGE_REQUEST to task A (event 1), and the *pipe* at PE0 does not contain messages to task B. Thus, the MESSAGE_REQUEST is

forwarded to the new address of task A (event 2). Task A at PE1 handles the request and deliveries the message to task B (event 3). Fig. 3(c) corresponds to the consumption of the last message produced by T_M in PE0 after the migration.

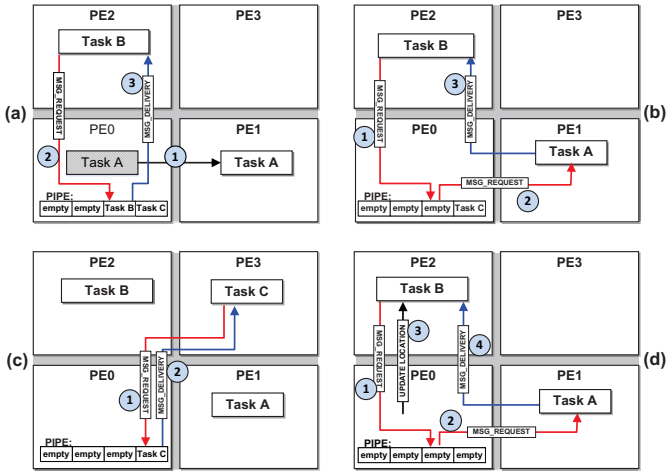

Fig. 3. Representation of the rules involved in the inter-task synchronization.

Rule 3: The task location table is updated in the OS of the consumer task with the T_{PE} address when a MESSAGE_REQUEST is received by S_{PE}, and there is *no message produced by T_M in the pipe* of S_{PE}. Thus, a MESSAGE_REQUEST is forwarded to T_{PE} as in rule 2, but a second message is sent to the OS of the consumer task to update the task location table with the T_{PE} address (UPDATE_LOCATION message). After this update, the OS of the consumer task uses T_{PE} address to send all requests to T_M. Fig. 3(d) presents this scenario. When PE0 receive a MESSAGE_REQUEST (event 1), this message is forwarded to PE1 (event 2), and a UPDATE_LOCATION message is sent to PE2 (event 3) to update the task location table with the new location of task A (PE1). After receiving the forwarded request, PE1 deliveries the message to PE2 (event 4).

V. Evaluation

The evaluation was conducted in a clock-cycle RTL SystemC description of the platform. The migration latency is due to the *task recreation* step and the *dynamic data migration*. The task recreation latency is constant in our proposal, regardless the T_M *text* size, once we adopt a dual-port memory and a module with DMA capabilities (DMNI). In systems with a single-port memory, this latency is proportional to the *text* size because the processor stalls during the transference. The task recreation time is due to the time spent by the OS to configure the DMNI with the memory address and size of the *text*. Once configured the DMNI module, T_M resumes it execution while its *text* is injected into the NoC. The task recreation step takes, in average, 2,700 *cc* (clock cycles). This value may suffer small variations due to the status of the DMNI at moment of OS request. The migration latency corresponds in fact to the time spent to transfer the dynamic memory section. The overhead grows linearly at a complexity of $O(n)$, where n is the sum of all dynamic memory sections and the T_M context (PC, SP, registers). Transferring this data requires stop T_M, and restart T_M at T_{PE} after the end of transference.

The evaluation of the dynamic data migration latency is carried out with a synthetic task and a variable data size. Fig. 4 presents the task migration latency between PEs at 1 hop distance, without disturbing traffic (each NoC router takes 5 *cc* to route a packet header). After a constant task recreation latency, 2,709 *cc*, the migration latency increases linearly with the data size, approximately 500 *cc* per KB, using 32-bit NoC channels. When compared to the related works (Section V.B), this value corresponds to a small task migration latency.

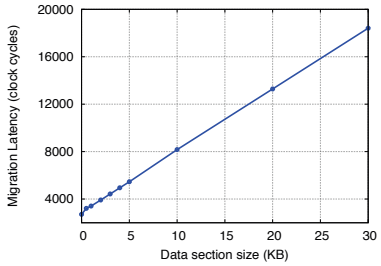

Fig. 4. Task migration latency according to the task data size, 32-bit NoC channels, 1 hop between PEs.

A. Impact of the Task Migration in the Applications' Performance

Fig. 5 presents the migration latency considering simultaneous task migration for the MJPEG application. Each point of Fig. 5 represents the latency to decode one frame with 128 bytes. The MJPEG application is modeled as a pipeline, with five tasks: *input*, *idct*, *ivlc*, *iquant*, *output*. The tasks are allocated alone in different PEs, and the task migration moves tasks to an idle PEs. The curve "*without task migration*" corresponds to the minimal application latency that the platform can sustain – baseline.

The migration order arrives when the simulation reaches 400,000 *cc*. The overhead over the baseline latency measured at the 4th frame was (in *cc*) 2,282 (+4%), 7,976 (+14%), and 10,794 (+19%), for 1, 2 and 3 migrations, respectively. The latency reduction at the 5th frame comes due to the message buffering caused by the blocking of the tasks, after the migration these buffered messages are consumed with high throughput. The application latency is restored one frame later.

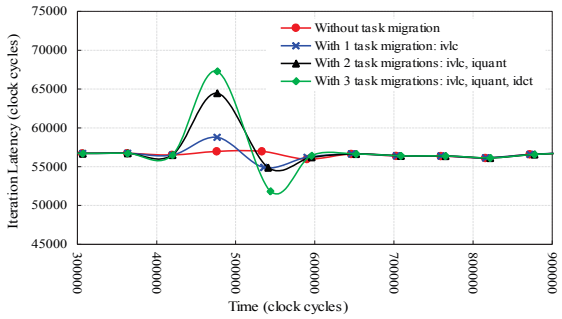

Fig. 5. MJPEG frame decoding latency for simultaneous task migrations.

The overhead over the application execution time, considering only the first *11 frames* of Fig. 5, was 0.05% (600 *cc* of penalty) for 1 migration, and 0.6% for 2, and 3 migrations. The overhead for 2 and 3 migrations is not cumulative due to the parallel execution of the migration in different PEs, and the inter-task message synchronization that masks the migration overhead due to its on-demand behavior to deliveries the pending messages. For many received frames, and frame sizes, this overhead is even smaller.

Fig 6. explores the proposed task migration targeting QoS. The MJPEG application is mapped dynamically on the system. At time **A**, a disturbing application is mapped increasing the frame decoding latency. At time **B,** a QoS heuristic (out of the scope of this work) fires 3 concurrent tasks migrations (all tasks belonging

to MJPEG application). The migration process finishes at time **C**, with the migrated tasks moved to free PEs, this restores the MJPEG latency to the baseline value. The MJPEG execution finishes at time **E** with task migrations. Without disturbing, the MJPEG finishes at time **D**, and with disturbing but without task migration it finishes at time **F**. The migration overhead over the application was 6,152 *cc* for 3 simultaneous task migrations (besides the speedup of 27% in the execution time).

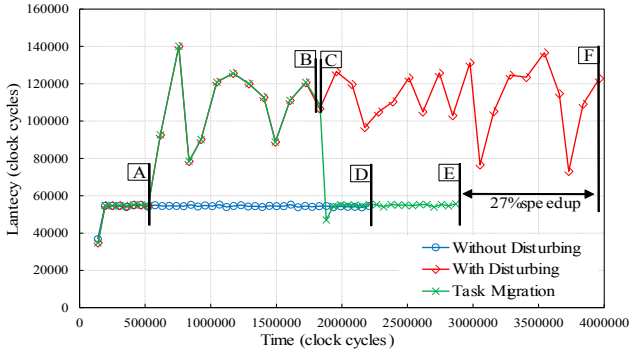

Fig. 6. Task migration applied for Quality of Service at MJPEG application.

B. *Latency Comparison with the State-of-the-Art*

Table II compares the proposed migration latency with results reported in the literature.

TABLE II. COMPARISON OF THE MIGRATION LATENCY AGAINST THE STATE-OF-THE-ART WORKS.

Proposal	NoC details	OS	Benchmark	Migration Latency
Munk et al. [7]	Theoretic proposal			
El-Antably et al. [8]	3-D torus	DNA-OS	Synthetic Apps	530,923 µs
Fu et al. [9]	N/A	uC/OS-II	Matrix Mult.	10,457 *cc*
Canella et al. [10]	2D mesh	in-house	MJPEG	25,000 *cc*
Jahn et al. [11]	2D mesh 4 Gbit/s per link	CARAT middleware	x264 / 7Zip / robotic app.	106,505 *cc* (average value)
Saint et al. [12]	2D mesh	HS-Scale OS	MJPEG	919,450 *cc*
This proposal	*2D mesh 3.2 Gbit/s per link*	*in-house*	*MJPEG*	*600-6,152 cc*

El-Antably et al. [8] evaluate the migration overhead using a synthetic application with three tasks: generator, processing, and consumer. The migration overhead is a function of the task size. The minimum overhead achieved was 530,923 µs for a task size of 4KB, increasing at a rate of 42.2 µs for each 4KB. Our proposal achieves an overhead of 4,945 *cc* based on Fig. 4 (equal to 49.45 µs @ 100MHz) for the same data size, at an increasing rate of 500 *cc* (5 µs) per KB.

Fu et al. [9] use a matrix multiplication benchmark to evaluate its migration protocol. NoC details are not available. The obtained application overhead was 10,457 *cc*, and the migration latency for one task was in average 7,500 *cc*. Our proposal achieves an MJPEG application overhead of 600 *cc* for one task migration. Fu et al. do not allow more than one simultaneous task migration and use task replication.

Cannella et al. [10] adopt as the benchmark a Sobel Filter and an MJPEG decoder. The MJPEG's task migration latency takes 25,000 *cc* in that work. Our overhead for MJPEG was, in the worst-case, equivalent to 24% to the one obtained in that work. Cannella et al. also employ task replication and require instrumentation in the task source code.

Jahn et al. [18] adopt three benchmarks: x264, 7Zip, and an embedded-systems robotic application. With the CARAT task migration mechanism, the migration latency may vary because the migration transfers the memory section dynamically according to the task behavior. For this reason, the application benchmarks presented different migration latency overheads. A video application, x264, resulted in a latency overhead 303,118 *cc*.

Saint et al. [12] use an MJPEG benchmark. The Authors employ a task migration in one task of the MJPEG application. The task migration process required 131.35 *ms* in a MIPS R3000, corresponding to a migration latency of 919,450 *cc*. For comparison purposes, the scenario of Fig. 5 detailed an overhead in the MJPEG decoding of 6,152 *cc* to migrate 3 tasks simultaneously.

VI. CONCLUSION

Task migration in distributed memory many-core systems is addressed with restrictions due to its overhead. The results obtained in this work demystify the high-cost of the task migration, presenting a low latency protocol compared to the State-of-the-Art. Additional relevant features of the proposal include: there is no need to replicate the code of the tasks; it is not necessary to modify the source code neither to add checkpoints; support to simultaneous migrations; and correct inter-task synchronization without migrating produced messages. A direction for future work is to evaluate the protocol with others self-adaptive techniques, as fault-tolerance.

ACKNOWLEDGEMENT

The Author Fernando Moraes is supported by CNPq and FAPERGS funding agencies.

REFERENCES

[1] Salami, B.; Baharani, M.; Noori, H. "Proactive Task Migration with a Self-Adjusting Migration Threshold for Dynamic Thermal Management of Multi-Core Processors". The Journal of Supercomputing. vol. 68(3), pp. 1068-1087, 2014.

[2] Johann, S.; et al. "Task Model Suitable for Dynamic Load Balancing of Real-time Applications in NoC-based MPSoCs". In ICCD, pp. 49-54, 2012.

[3] Marwedel, P.; et al. "Mapping of applications to MPSoCs". In: CODES+ISSS, pp. 109-118, 2011.

[4] Abbas, N. and Ma, Z. "Run-time Parallelization Switching for Resource Optimization on an MPSoC Platform". Design Automation for Embedded Systems, vol. 18(3-4), pp. 279-293, 2014

[5] Das, A.; Kumar, A.; Veeravalli, B. "Communication and Migration Energy Aware Design Space Exploration for Multicore Systems with Intermittent Faults". In: DATE, pp. 1631-1636, 2013.

[6] Ruaro, M., Lazzaroto, F.; Marcon, M.; Moraes, F. "DMNI: A specialized network interface for NoC-based MPSoCs". In: ISCAS, pp. 1202-1205, 2016.

[7] Munk, P. et al. "Position Paper: Real-Time Task Migration on Many-Core Processors". In: ARCS, pp. 1-4, 2015

[8] El-Antably, A. et al. "Transparent and portable agent based task migration for data-flow applications on multi-tiled architectures". In: CODES+ISSS, pp. 183–192. 2015.

[9] Fu, F.; Wang, L.; Lu, Y.; Wang, J. "Low Overhead Task Migration Mechanism in NoC-based MPSoC". In: ASICON, 4 p., 2013

[10] Cannella, E. et al. "Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks". VLSI Design, article nº 2, p. 17, 2012.

[11] Jahn, J.; Faruque, M.; Henkel, J. "CARAT: Context-aware Runtime Adaptive Task Migration for Multi Core Architectures". In: DATE, 6 p., 2011.

[12] Saint-Jean, N. et al. "MPI-Based Adaptive Task Migration Support on the HS-Scale System". In: ISVLSI, pp. 105-110, 2008.