Hardware Task Migration Module for Improved Fault Tolerance and Predictability

Shyamsundar Venkataraman, Rui Santos and Akash Kumar
Department of Electrical and Computer Engineering
National University of Singapore
Email: {shyam, elergvds, akash}@nus.edu.sg

Jasper Kuijsten
Department of Electrical Engineering
Eindhoven University of Technology
Email: j.j.a.kuijsten@student.tue.nl

***Abstract*—Task migration has been applied as an efficient mechanism to handle faulty processing elements (PEs) in Multiprocessor Systems-on-Chip (MPSoCs). However, current task migration solutions are either implemented or emulated in software, compromising intrinsically the predictability and degrading the system robustness. Moreover, the initial placement and mapping of the tasks in the MPSoC plays an important role in minimising the task migration overhead and overall system energy.**

This paper proposes a hardware-based task migration scheme for MPSoC systems, offering better predictability as well as an improved method of fault tolerance. The proposed scheme intelligently generates an initial placement for the tasks with improved fault tolerance and stores these mappings on a hash map, which is looked up at run-time as and when faults occur. Compared with the state-of-the-art, our scheme performs up to $1500\times$ faster task migration without any significant overheads.

I. Introduction

Over the last few years, Multiprocessor Systems-on-Chip (MPSoCs) are gaining significance in modern embedded systems, since they satisfy an increasing demand of performance and scalability required by the emerging applications. However, with reducing feature size and increasing transistor count, MPSoCs are becoming more susceptible to permanent and transient faults [1]. This research focusses on permanent fault-tolerance techniques.

Permanent faults are traditionally tackled by using hardware redundancy [2]. However, stringent area and power requirements prohibit the use of such traditional methods in modern systems. Task migration has been applied as an effective solution to overcome such errors by remapping a task from a faulty Processing Element (PE) to other functional PEs [3]–[9]. Most recent MPSoC systems are implemented with their PEs interconnected with networks-on-chip (NoCs) generally in a mesh-based architecture. Multimedia applications implemented on MPSoC systems are typically executed multiple times in a periodic fashion. Consistent throughput is required for multimedia applications to satisfy the needs of the users. However, when an error occurs in one of the PEs, the task has to be remapped to another PE to continue the system operation. In this sense, the time taken to migrate the task is pivotal in maintaining the throughput of the application.

One major drawback of current task migration techniques is that they work on the principle of *pro-active* fault tolerance. This requires knowledge about the occurrence of faults a-priory, which is not very realistic since critical faults are hardly ever predictable. A solution that provides *reactive* fault tolerance, i.e. tolerating faults after it has occurred, would be more pragmatic than the proactive solution.

Another drawback of the current task migration implementations, is the reduced predictability. Predictability is especially important in real-time embedded platforms, where the timing constraints are strict. On the destination PE, the migration routine execution can interfere with the timeliness of the current running tasks, or on the other hand, if the migration routine is not executed on time the migrated task can miss its deadline. Moreover, software implementations have intrinsic overheads and high jitter in their actions, which also affects the system predictability.

Contributions: This paper focusses on a hardware-based task migration scheme for tolerating permanent faults in MPSoC systems with applications modelled using Synchronous Data Flow Graphs (SDFGs) [10]. Key contributions of this paper are the following:

- A processor independent task migration scheme, using dedicated hardware modules with direct access to the processor memory, making reactive fault tolerance possible.
- A predictable task migration operation with minimum hardware overhead, suitable for real-time multimedia applications with strict timing constraints.

Intelligent mapping of tasks to each of the PEs are generated at compile-time. These mappings are then stored in a hash map, for look-up as and when faults occur. When a fault occurs, the destination PE is selected to migrate by referring to the table generated earlier. A hardware based task migration migrates the task to the destination PE in a predictable way and the task is then continued to run from the new PE. Compared with the state-of-the-art, our scheme performs up to $\mathbf{1500}\times$ faster task migration without any significant overheads.

The remainder of the paper is organised as follows. Section II presents related works concerning existing task migration methods for MPSoCs in the context of fault tolerance. Section III introduces the application and architecture model considered in this research. Section IV then discusses the implementation details followed by a case study in Section V. The next Section VI presents the experiments conducted and the results obtained and finally, Section VII concludes the paper with future directions.

II. Related Works

Task migration mechanisms have been applied for different purposes in MPSoCs such as load balancing, performance improvement, power management and fault-tolerance. For this reason, it has been a topic of intense research and study. In particular, researchers have put their efforts in finding software solutions to reduce and limit the task migration overheads. Nollet *et al.* [3] propose to reduce overhead of the migration software daemon by exploiting platform specific debug hardware. The migration is triggered by the Operating System, which informs the task to be migrated. Aguiar *et al.* [4] explore different architectures for task migration, exploring trade-offs like flexibility vs. resource usage. Moreover, they propose checkpoint locations on application-level to reduce the size of the migrated task. Saraswat *et al.* [5] propose an online heuristic for task migration in fault-tolerant embedded systems, where the critical tasks are assured to still meet their deadlines and the Quality of Service of non-critical tasks is optimized. However, they assume a platform where all PEs have access to a shared memory with zero migration transfer duration, hence their solution is not applicable to platforms with distributed memory. Furthermore, they only consider pro-active fault-tolerance. Abebei *et al.* [6] present an algorithm for re-mapping an application execution in case a processor becomes faulty. However, important details are missing, in particular how the task migration is performed, using a shared or distributed memory. Chakravorty *et al.* [7] and Engelmann *et al.* [8] employ task migration for fault-tolerant HPC-cluster systems. Both these techniques require knowledge of the critical PE faults a-priori. Acquaviva *et al.* [9] also propose and assess a middleware layer that implements task migration in MPSoCs, but targeted to soft real-time multimedia applications. Sarkar *et al.* [11] reduce a case of second-order overhead due to task migrations, by using a push-based protocol to replace data in caches before migrations, to reduce the amount of cache misses after the migration.

From the implementation perspective, different migration mechanisms have been proposed in the literature. These migration mechanisms can be classified in two categories, namely using *replication* and *re-creation*. In replication mechanism, one task that is created in one core is also replicated into the other cores [12]. Therefore, when the task migration is required, only the execution pointer and the corresponding context states are migrated to the core that will continue the execution. Using the latter mechanism, one task is created in only one PE. During the execution, when the migration is required, the entire task (code, data and context execution) is migrated to the receiver core, to continue its execution from the point where it was suspended.

Bertozzi *et al.* [13] present a re-creation task migration mechanism, based on a software middleware layer, which interfaces the process' execution and the kernel. This layer handles the whole migration process, which can be performed in a set of points defined by the user, at the application code. Briao *et al.* [14] also follow a re-creation approach and investigate the impact of migration overhead in real-time systems. The task migration is performed through the interprocess communication messages, used to migrate the application code, data and processor context.

Gantel *et al.* [15] propose a task migration approach based on the replication method. In particular, they present a software layer, which handles the migration among soft-core processors. Cannella *et al.* [16] present a middleware layer, which allows the integration of Polyhedral Process Network (PPN) processes with NoC-based MPSoCs. This middleware layer includes two main components. The first one is called PPN communication and it is responsible for the communication and respective synchronization between the PPN processes in different PEs on the MPSoCs. The second one is responsible for the migration process among the different PEs. The proposed solution uses the replication mechanism, since the faster execution is privileged. An extension of that work is introduced by Meloni *et al.* [17], where the authors propose a hardware task migration module similar to our work. They argue that their module performs independent task migration from a faulty PE. However, they emulate its behaviour in software and, consequently, no hardware overhead and predictability evaluation are performed.

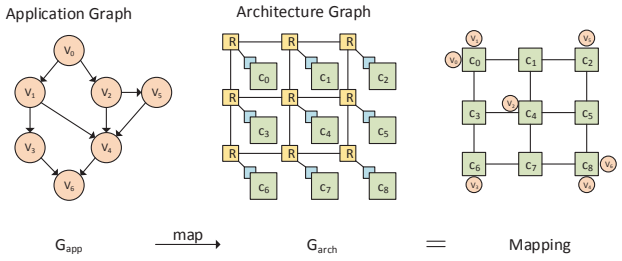

Fig. 1. Application and architecture model

All the presented works implement or emulate the task migration mechanisms in software, creating middleware layers to increase the operating system functionalities. However, from the timeliness and predictability perspective, the software implementations have intrinsic overheads and high jitter in their actions. On the other hand, from the fault-tolerance perspective, if a PE fails during the execution, the migration of this task in software to another PE (to complete its execution) would be impossible, since its own PE is faulty. To overcome this limitation, we propose a new hardware approach for the task migration mechanism in MPSoCs, which provides predictable task migration suitable for multimedia applications with strict timing and throughput constraints.

III. Application and Architecture Model

A. Application Model

The application is modelled as a directed graph $G_{app} = (V_{app}, E_{app})$, where V_{app} is the set of nodes representing the tasks of the application and E_{app} is the set of edges representing the data dependency among the tasks. Each task $v_i \in V_{app}$ is a tuple $< T_i, S_i, \{D_{ij}\} >$ where T_i is the execution time of v_i, S_i is the state space (program and data memory) and D_{ij} is the data produced on edge e_{ij} on every execution of v_i.

B. Architecture Model

A mesh-based Network-on-Chip (NoC) architecture has been used for this research. However, the scheme presented in this paper can be extended easily to other NoC architectures such as torus and trees. The NoC is represented as $G_{arch} =$

(V_{arch}, E_{arch}), where V_{arch} is the set of nodes representing the PEs and the E_{arch} is the set of edges representing the connection between the PEs. Mapping an application to an MPSoC is represented as follows.

$$G_{app} \xrightarrow[\text{n}]{\text{map}} G_{arch} := G(V_{arch}, V_{app}) := M_n$$

where n is the number of cores of the MPSoC used by an application. With every core $c_i \in V_{arch}$, a set A_i is associated, consisting of the task(s) mapped to c_i. Thus, $V_{app} = \cup_{i=1}^{n} A_i$. It is important to note that the proposed architecture also requires a global shared memory, which is accessible by all the PEs through the NoC. The need for the shared memory is discussed further in Section III-D. Figure 1 describes the model used for the application and the architecture and a sample mapping of the application to the MPSoC.

C. Fault Model and Assumptions

- Permanent faults are self-revealing
- Faults on active cores are independent
- Only faults on the PEs are of concern
- Checkpointing is done to save the state space of the tasks
- Probability of more than one PE failing in a small interval Δt is negligible
- Each processor of the MPSoC has its own private data and instruction memory

D. Design Time Task Mapping

Definition 1: (COMMUNICATION ENERGY) *The energy used in sending and receiving of data between two different tasks of the same application. Communication energy is represented as E^{ij}_{comm} where $i, j \in V_{app}$ and communication energy for two tasks mapped to the same PE is considered zero. Thus,*

$$E^{ab}_{comm} = 0 | a \in V^i_{arch} \wedge b \in V^j_{arch} \wedge i = j$$

Definition 2: (MIGRATION ENERGY) *The energy used in migrating a task $v_i \in V_{app}$ from V^i_{arch} to V^j_{arch}. Migration energy is represented as E^{ij}_{mig}, where $i, j \in V_{app}$.*

As discussed in the introduction of this paper, the initial mapping of the tasks V_{app} is pivotal in determining the time taken to migrate the task in case of a PE failure. Moreover, the initial mapping also determines the amount of communication and migration energy utilised by the application.

To generate an initial task mapping for the tasks in G_{app}, we use the technique proposed by Das *et al.* [18]. In this technique, the authors propose a communication energy aware fault-tolerant task mapping of throughput constrained multimedia applications. The technique calculates the communication energy (E_{comm}) and the migration energy (E_{mig}) for a given G_{app} and G_{arch}, and generates an initial task mapping that results in minimal communication and migration energy usage.

The technique calculates in design time, the various fault scenarios and the mapping that would provide the minimum communication and migration overhead. These mappings are then stored in a *HashMap*, which is then accessed in the case of a fault during run-time. The HashMap specifies the corresponding mapping for each fault scenario possible. Since the HashMap has to be accessible by all the PEs in the MPSoC, the HashMap is stored in the global shared memory of the system. For further details on the implementation of the technique, the reader is urged to refer to the paper [18].

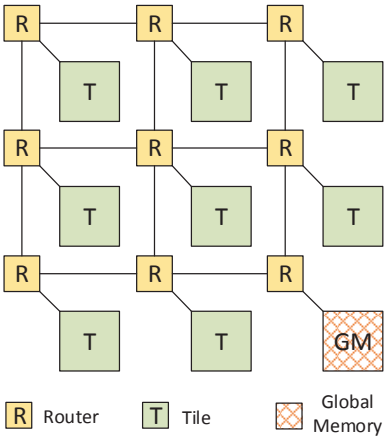

Fig. 2. Overall system architecture

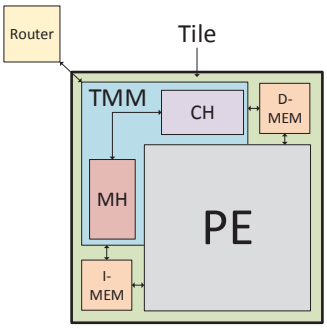

Fig. 3. Architecture of each tile

IV. TASK MIGRATION MODULE

To provide a stable throughput guarantee of multimedia applications, even in the event of a fault in one of the PEs, the task migration mechanism has to be predictable. Moreover, to provide reactive fault tolerance, the task migration should be independent from the working of the processor. With these goals in mind, a hardware based task migration scheme has been proposed, which can provide both proactive as well as reactive fault tolerance. The following sections describe the Task Migration Module (TMM) architecture and details the flow of the TMM.

A. TMM Architecture

Figures 2 and 3 detail the overall architecture of the MPSoC along with the architecture of the TMM. The TMM sits between the PE and the NoC router and acts as a Network Interface (NI). Reactive fault tolerance is made possible by making the TMM independent from the processor and is given direct access to the state space (instruction and data memory) of the processor. In the event of a fault, the TMM stops forwarding data and migrates the tasks on the PE according

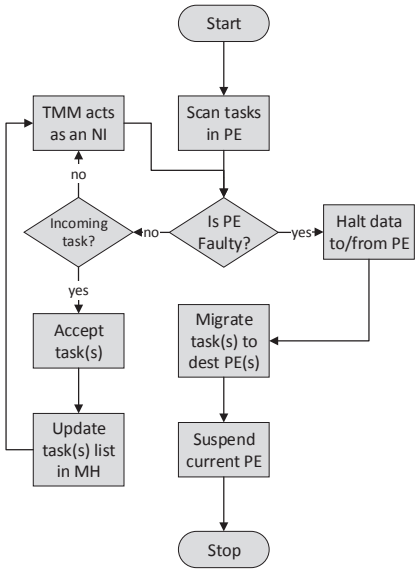

Fig. 4. Flow of the TMM

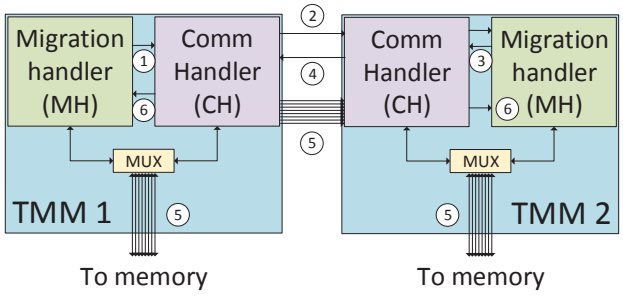

Fig. 5. Flow of the task migration in hardware

to the HashMap computed at design time (retrieved from the global memory). The TMM is made of two major components – Migration Handler (MH) and the Communication Handler (CH). The MH performs the main operations of forwarding packets to and from the PE. Moreover, and also initiates the task migration in the case of a permanent fault. Referring to the global shared memory that contains the HashMap, the MH can decide which destination PE to send a task to. The CH handles all the communication to and from the router. In the case of an outgoing task migration, it forwards all the task state space to the destination PE. In the case of an incoming task migration, it forwards the task details to the MH and writes the task state space into the PE memories.

B. TMM Flow

Figure 4 shows the entire flow of a single TMM connected to a PE. When the system is initialised with tasks in the PEs, the TMM scans the PEs for the list of tasks that have been loaded. The TMM stores this data locally in its memory. As long as the PE is not faulty, and there are no incoming task migration requests, the TMM basically acts as a Network Interface (NI) and connects the PE to the NoC router. If there is an incoming task request, the TMM migrates the incoming task(s) and loads them into the current PE memory. The TMM also updates the task list with the new tasks migrated. In the case of the current PE failing, the TMM immediately stops all the flow of data to and from the PE. It then migrates all the tasks in the current PE to a destination PE as has been computed during the design time and finally, the TMM suspends the faulty PE.

The indication of incoming and outgoing task migration requests is done through the use of *header packets*. These header packets are small packets that are sent to the destination PEs before the tasks are actually migrated. The TMM in each of the PEs continuously look out for these header packets and hence are able to identify if a permanent fault has occured in any PE. Once the destination PE receives the header packet, it sends a *reply packet* to the migrating TMM, confirming that it is ready to accept the new task. Once the migrating TMM receives the reply packet, it proceeds to send the task code and data to the receiving PE.

For the reader to obtain a better idea of the entire flow of the TMM, an example sequence of migration is presented along with an illustration shown in Figure 5. The figure only shows the interaction between the two TMMs and does not show the NoC between. The following points details the steps taken by the system when PE1 fails due to a permanent fault and the current task(s) running on PE1 is migrated to PE2. PE1 is associated with TMM1 and similarly, PE2 with TMM2.

1) Migration Handler (MH) of TMM1 detects a permanent fault on PE1 and prepares to migrate the tasks running on the PE1 to PE2.
2) Communication Handler (CH) of TMM1 creates a *header packet* and sends a request to the CH of TMM2.
3) CH of TMM2, after receiving the header packet, forwards it to the MH, which then prepares to accept the task being migrated from PE1.
4) CH of TMM2 conveys that the TMM2 is ready to accept the incoming migration by sending a *reply packet*.
5) CH of TMM1 reads data from its PE and sends it to the CH of TMM2, which in turn writes it to the memory of its PE.
6) Once the data transfer is done, both the CHs notify their corresponding MHs. The task can now continue from the new PE context loading the task from memory.

C. TMM Implementation

To measure time taken to migrate a task from one PE to another, a hardware of the TMM has been implemented. For the sake of simplicity, the system comprises of a 2×2 MPSoC implemented on a Virtex-6 ML605 FPGA board, which has a XC6VLX240T-1f1156 FPGA chip. The multiprocessor system was built using the Xilinx Microblaze (MB) Cores for the 4 PEs. The Xilinx Microblaze IP is a general purpose configurable x86 CPU designed by Xilinx and is highly optimised for FPGA synthesis. The implemented design employs the MBs in their most lightweight form without any kind of optimisation, e.g. caches or memory management units. The TMM was implemented as a custom hardware peripheral and connected to the Processor Local Bus (PLB) of the MB. The MB is further connected to its memories through dual-ported BRAM

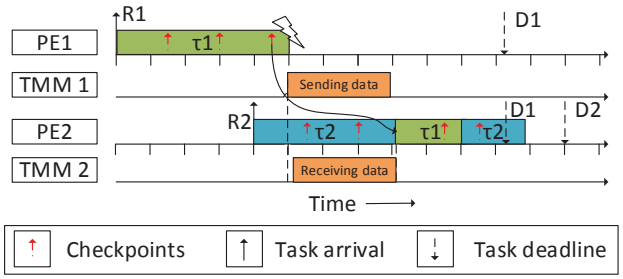

Fig. 6. Migration of the PRESENT algorithm task at runtime

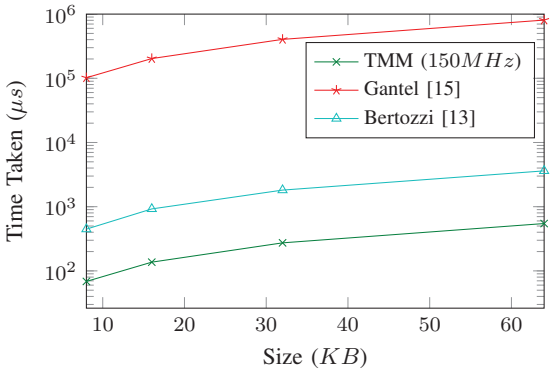

Fig. 7. Comparison of time taken to migrate different task sizes

cores. All the communications between the TMMs use the Fast Simplex Link (FSL), which is an asynchronous bidirectional 32-bit wide link with FIFO-buffers both at the sending and receiving side.

V. Case Study

To evaluate the hardware implementation of the TMM, the PRESENT cipher algorithm was implemented on one of the PEs. PRESENT is a lightweight block cipher algorithm developed quite recently [19]. It is one of the most compact encryption methods ever designed and is $2.5\times$ smaller than the Advanced Encryption Standard (AES). The block size implemented was 64 bits and the key size was set to 80 bits for the purpose of the case study.

Since the PRESENT algorithm is a lightweight block cipher, it was implemented as a single task in one of the four processors of the MPSoC. Regular checkpoints were setup while the algorithm was running on the MB by copying the values of the registers to a pre-assigned memory of the MB. To simulate the failure of the PE, a push button was used to trigger a permanent fault. An image file was given as input to the algorithm and was made to execute in the initial PE. During the execution, the push button was triggered to simulate a permanent fault. Figure 6 traces the flow of the task execution before and after the task is migrated. As soon as a fault is detected by the TMM, it immediately halts further data from being processed. It then sequentially reads and transmits the instruction and data memory of the program to the destination PE. Once the TMM in the destination PE receives the task, it resumes the task from its last checkpoint. It is to be noted here that it is the decision of the destination processor on when to execute the newly migrated task. Moreover, the execution would also depend on the scheduling algorithm being run by each of the PEs. For the purpose of the case study, another PRESENT algorithm was running in the destination PE and was pre-empted when the new task was migrated.

VI. Experiments and Results

The following experiments discuss the results of the hardware implementation.

A. Timing Analysis

The time to migrate a task from one PE to another can be given by the following equation.

$$T_{mig} = ((d \times n_{byte}) + n_{chnl} + n_{rd/wr}) \times 10^{-2}$$

TABLE I. Speed up obtained with TMM

	Time to migrate $8KB$ (μs)	Speed up
TMM (150 MHz)	68.5	1
Gantel [15]	102000	1489.05
Bertozzi [13]	447.1	6.52

T_{mig} is the number of cycles to migrate the task from one PE to another. d is the size of the task to be transferred in bytes and n_{byte} is the number of cycles taken to transfer a single byte from one PE onto the NoC. In our current implementation, n_{byte} is 1.25 while d is varied for a wide range of values. n_{chnl} and $n_{rd/wr}$ are the number of cycles taken for the communication delay in the NoC and the read and write delay, respectively. While $n_{rd/wr}$ is constant (depending on the underlying architecture of the PE), n_{chnl} depends on the traffic existing on the NoC channel.

Experimental timing analysis was conducted with different data memory sizes and the time it took to completely migrate the task from the host to the destination PE was noted. As expected, the time to migrate a task grows linearly as the size of the data increases. On average, the time taken to migrate a couple of tens of kilobytes only takes a few hundred microseconds. Moreover, the migration time measured experimentally matches perfectly with the theoretically estimated value.

Figure 7 shows the time taken to migrate 8, 16, 32 and 64 KB of data using techniques proposed in this paper, by Gantel *et al.* [15] and Bertozzi *et al.* [13]. Since both [15] and [13] use a software level or middleware level task migration schemes, the time taken to migrate is in the order of milliseconds. The task migration proposed in this paper however is only in the order of microseconds due to the implementation of the migration in hardware. The synthesis report for TMM states a maximum allowed clock frequency of 165.782 Mhz. Hence, the TMM is set to run at 150 MHz. It is to be noted here that the migration results in [13] do not specify the frequency at which the system was run. As such, it was assumed that the processor was run at a frequency which would produce the best possible result for task migration.

Table I shows the speed up obtained as compared with other techniques. When the TMM is run at 150 MHz, it performs about $1500\times$ better than the one in Gantel *et al* and more than $6\times$ better than the one in Bertozzi *et al.*

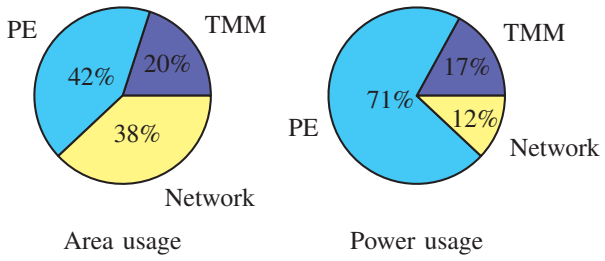

Fig. 8. Area and power usage of different components in the system

B. Area and Power Analysis

The current implementation of TMM uses 20% of the entire system while the communication network and PEs take 38% and 42% of the system, respectively. Moreover, each PE, TMM and network of the system uses 1389, 596 and 612 Look-up Tables (LUTs), respectively. Figure 8 compares the area and power usage of the TMMs along with the PEs and the FSLs. The TMMs uses only 17% of the power of the PEs. The power consumption was measured using the XPower Analyser tool provided by Xilinx, for an ambient temperature of 25° Celcius and an airflow of 250 Linear Feet per Minute (LFM) without any heat sinks.

C. Memory Overhead

Since the context is saved and loaded during migration, extra memory space is required for storing the task context. The size of the task context is dependent on the processor on which the technique is implemented. Microblaze has 32 32-bit general purpose registers and 11 other special purpose registers. These 43 registers hence require an additional 172 bytes of memory. Moreover, 16 additional bytes of memory was used for synchronization of data between the TMMs. Therefore, the total memory overhead amounts to 188 bytes per processor. It is to be noted that any system that requires context saving and loading would require this overhead and is hence not specific to this technique.

VII. Conclusion and Future Works

In this paper, we present a new scheme to tolerate permanent faults in an MPSoC system. Tasks are mapped to the MPSoC in such a way that they minimise the communication and migration energy overhead. Moreover, a processor independent hardware task migration module with a predictable delay has been proposed to speed up the task migrations in the event of a permanent fault in one of the processors. Compared to the state-of-the-art, the proposed scheme performs much faster in terms of task migration. This helps guarantee the throughput of applications having strict timing constraints.

In the future, we would like to explore using a single TMM for multiple processors, leading to a further decrease in the area overhead. Extending the scheme to heterogeneous architectures is also another improvement that can be looked into in the future. Moreover, an open source tool release is planned to help researchers world-wide to benefit from our work and easily merge their designs with the proposed architecture and test their techniques with various benchmarks and compare with state-of-the-art techniques.

VIII. Acknowledgement

This work was supported in parts by the DSO National Laboratories under the Low Power Platform for Sensing, Signal Processing, and Communications in Nanosat (Grant No. R-263-000-B33-112).

References

[1] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE micro*, 2003.

[2] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware co-synthesis for embedded systems," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 2007.

[3] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, "Low cost task migration initiation in a heterogeneous MP-SoC," in *Design, Automation and Test in Europe (DATE)*, 2005.

[4] A. Aguiar, S. J. Filho, T. G. dos Santos, C. Marcon, and F. Hessel, "Architectural Support for Task Migration Concerning MPSoC," *Congresso da Sociedade Brasileira de Computao* , 2008.

[5] P. K. Saraswat, P. Pop, and J. Madsen, "Task migration for fault-tolerance in mixed-criticality embedded systems," *Special Interest Group on Embedded Systems (SIGBED)*, 2009.

[6] C. Ababei and R. Katti, "Achieving network on chip fault tolerance by adaptive remapping," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[7] S. Chakravorty, C. Mendes, and L. Kal, "Proactive Fault Tolerance in MPI Applications Via Task Migration," in *International Conference on High Performance Computing (HiPC)*, 2006.

[8] C. Engelmann, G. Vallee, T. Naughton, and S. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2009.

[9] 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*, 2008.

[10] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, 1987.

[11] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted migration of real-time tasks in multi-core processors," in *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2009.

[12] S. Holmbacka, W. Lund, S. Lafond, and J. Lilius, "Task Migration for Dynamic Power and Performance Characteristics on Many-Core Distributed Operating Systems," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2013.

[13] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Design, Automation and Test in Europe (DATE)*, 2006.

[14] E. W. Briao, D. Barcelos, F. Wronski, and F. Wagner, "Impact of task migration in NoC-based MPSoCs for soft real-time applications," in *IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2007.

[15] L. Gantel, 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 (ReConFig)*, 2009.

[16] E. Cannella, O. Derin, P. Meloni, G. Tuveri, and T. Stefanov, "Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks," *VLSI Design*, 2012.

[17] P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin, and M. Sami, "System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach," in *Euromicro Conference on Digital System Design (DSD)*, 2012.

[18] A. Das, A. Kumar, and B. Veeravalli, "Energy-Aware Communication and Remapping of Tasks for Reliable Multimedia Multiprocessor Systems." in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[19] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, *PRESENT: An ultra-lightweight block cipher*. Springer, 2007.