A System-level Simulation Framework for Evaluating Task Migration in MPSoCs

Wei Quan[†,‡] Andy D. Pimentel[†]

[†]Informatics Institute
University of Amsterdam
The Netherlands
{w.quan,a.d.pimentel}@uva.nl

[‡]School of Computer Science
National University of Defense Technology
Hunan, China
quanwei02@gmail.com

ABSTRACT

Task migration is the transfer of the execution of a process (task) from one processing element to another. It originates from the massive deployment of distributed systems in the parallel computing field to enable dynamic load distribution, fault resilience and to enhance data access locality. With the development of MultiProcessor System-on-Chip (MPSoC) architectures, the topic of task migration has recently regained research interest in the embedded systems domain. In this paper, we present a high-level simulation framework to study task migration for MPSoC systems. With this framework, different migration methodologies on different underlying hardware systems can be easily and rapidly modeled, simulated and evaluated during the early stages of design. By using this high-level simulation framework, a designer can study the migration impact on the overall performance of the system by exploring different task migration mechanisms (determining *what* and *how* to migrate) or using different migration policies (determining *when* to migrate *which* tasks *whereto*) in a specific task migration mechanism. Using a number of experiments, we demonstrate the capabilities of our simulation framework.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms

Design, Verification, Measurement

Keywords

Embedded systems, MPSoC, task migration, simulation

ESWEEK'14 October 12 - 17 2014, New Delhi, India

http://dx.doi.org/10.1145/2656106.2656111.

1. INTRODUCTION

To fulfil the computation requirements of modern sophisticated applications, Multi-Processor Systems-on-Chip (MPSoC) architectures have in recent years received much attention in the embedded systems domain. MPSoC systems often require to support an increasing number of applications and standards, where multiple applications may concurrently execute and contend for resources. Consequently, to exploit the full potential and capabilities of such architectures, the mapping of multi-application workloads onto the processing elements of the MPSoC becomes increasingly challenging. This is also due to the fact that the initial task mapping may need to change at run time for different reasons such as the requirements of supporting application dynamism (the change of application execution mode), fault tolerance, load balancing or thermal balancing. For this reason, the concept of task migration has been gaining research attention in the domain of MPSoC design [5]. Task migration is the transfer of the execution of a process (task) from one processing element to another. The concept originates from the massive deployment of distributed systems (and, in particular, distributed operating systems) in the parallel computing domain.

The main idea of task migration is the transfer of task context (state)[1] and its address space between processors [11]. In the domain of MPSoC systems, two main aspects should be carefully considered to support task migration in different architectures, namely *what* and *how* to transfer during migration. Regarding the problem of *what* to transfer during migration, it depends on the architecture of the target system, i.e. homogeneous versus heterogeneous. Different processor types have different ISAs, address widths, register file sizes, etc. Migration of tasks between heterogeneous cores requires different program codes and task contexts. Even between homogeneous cores, the data that need to be transferred during migration varies among different migration mechanisms. The second problem – *how* to transfer – is related to the organization of the memory (shared and/or distributed) and interconnection (bus, NoC, etc) of the target system. This determines what kind of communication technique (load/store instructions or message passing) should be used for task migration.

[1]The context of a task or a process is the minimal set of data used by this task/process that must be saved to allow either task interruption and/or migration at a given instant, and a continuation of this task at the point it has been interrupted/stopped and at an arbitrary future instant.

Besides the architecture related task migration mechanism described above, the policies of task migration – determining *when* to migrate task(s), *which* task(s) will be migrated and *where* these tasks will be migrated to – are also very important. Such policies may highly depend on the goal of task migration (fault tolerance, load balancing, thermal balancing, etc.). To design migration-enabled MPSoC systems, a system designer needs to be able to determine what mechanism and policy of task migration are the best choices for the target system already during the early stages of design. To this end, this paper presents a system-level simulation framework that supports the flexible and efficient modelling, simulation and exploration of different task migration mechanisms and policies in MPSoCs. Using a number of experiments, in which we study task migration in both shared-memory and message-passing MPSoC architectures, we also demonstrate the flexibility and capabilities of our simulation framework.

The remainder of this paper is organized as follows. Section 2 introduces several task migration mechanisms. Section 3 provides a detailed description of our task migration simulation framework. Section 4 introduces two task migration case studies and presents their experimental results. Section 5 discusses related work, after which Section 6 concludes the paper.

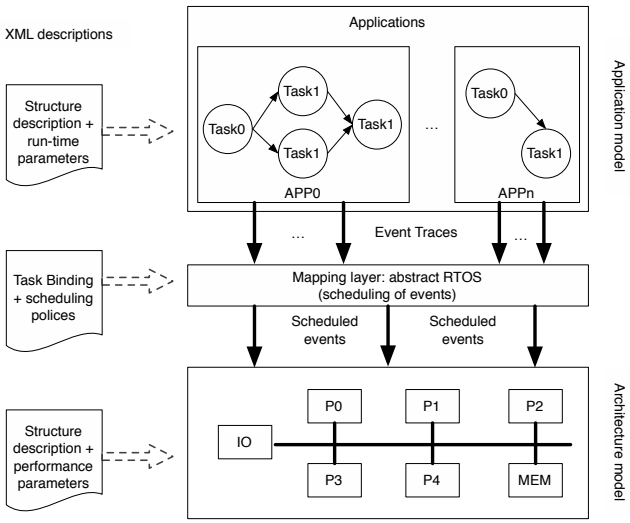

Figure 1: Sesame Framework

2. TASK MIGRATION MECHANISMS FOR DIFFERENT ARCHITECTURES

In this section, we will introduce several well-known task migration mechanisms for different hardware architectures. Here, system architectures can be divided into three categories according to the system memory organization: Uniform Memory Access (UMA) [13], Non-Uniform Memory Access (NUMA) [16] and NO Remote Memory Access (NORMA) [5, 11].

In a UMA system architecture, all the processors uniformly share the physical memory. The cost of accessing the memory is the same for all the processors in the system. A typical example of this architecture are the tightly coupled shared memory SMP (Symmetric Multi-Processor) systems, where all the processors run a single copy of an operating system that coordinates global activities. In SMP systems, task migration only needs to transfer the task's context between processors, while the address space of the migrating task does not need to be transferred since it is located in the shared memory that is shared by all processors. In this case, the cost of task migration is relatively low compared to the other multiprocessor architectures. In contrast to UMA, the memory access time in a NUMA architecture depends on the memory location. A processor can access its own local memory faster than non-local memory (e.g., memory local to another processor). In this kind of architecture, besides the task's context, the address space contents of the migrating task may also need to be transferred between different memories. Clearly, such task migrations come at the cost of a performance penalty and increased on-chip communication.

In the two previous types of architectures, processors share a single address space (i.e., the memory is physically/logically shared among processors) and therefore the data transfer of task migrations can be done via *load* and *store* instructions. This is, however, not possible for NORMA architectures where processors have a private memory (and address space) and do not grant access to their memory by other processors. In NORMA architectures, the task migration must therefore be coordinated using messages (i.e., message-passing) between processors. As a consequence, the cost of task migration in NORMA architectures typically can be high due to the need of transferring the migrating data over relatively slow, multi-hop communication channels like in a NoC.

For scalability reasons, future MPSoC systems will increasingly be equipped with distributed memory, i.e., either use NUMA or NORMA architectures. This means that the impact of task migration is not negligible with respect to system performance. To reduce the task migration cost, several task migration mechanisms like the *eager-copy*, *pre-copy* and *post-copy* [3, 27] have been proposed. For heterogeneous architectures, as different processor types requires different program code and task contexts, the complexity of task migration is much higher than for their homogeneous counterparts. State-of-the-art systems solve this through checkpointing [7, 18, 21] and application-level save-restore mechanisms [7], while there exists no mechanism that is fully transparent to applications [14].

3. TASK MIGRATION SIMULATION FRAMEWORK

To study and evaluate the impact of different task migration schemes on the overall performance of a target system, we have developed a flexible and efficient system-level simulation framework. This framework is based on and extends the open-source Sesame system-level MPSoC simulator [23]. The Sesame modeling and simulation environment, which is illustrated in Figure 1, facilitates efficient performance analysis of embedded (media) systems architectures. It recognizes separate application and architecture models, where an application model describes the functional behavior of an application and the architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular application, underlying architecture, and mapping.

During simulation, the application model issues trace events (*read*, *write* and *execute* events), which are an abstract representation of the computational (execute events) and communication (read/write events) workload imposed on the architecture. These events are processed by the architecture model to simulate their consequences in terms of performance and power consumption. The relationship between the tasks in an application and the hardware resources is captured by the mapping layer. Before starting a simulation, an

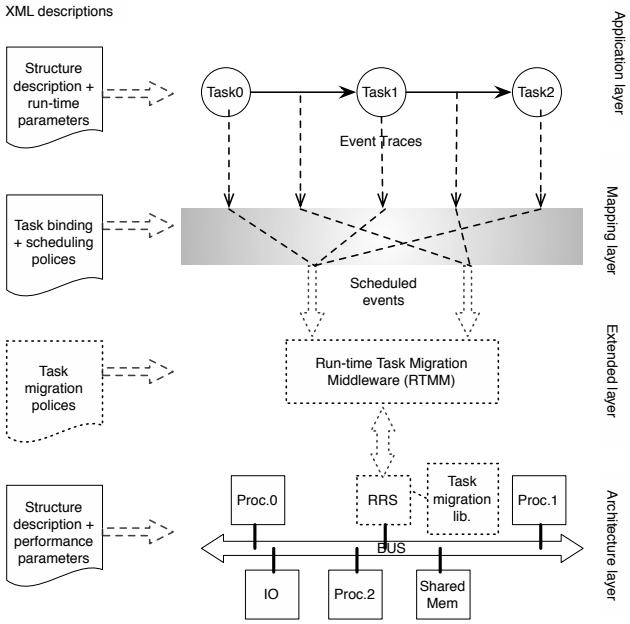

Figure 2: Extended Sesame's layered infrastructure

explicit task mapping[2] description should be provided to the mapping layer. In the original Sesame simulator, this mapping cannot be changed dynamically during simulation, which evidently limits the study of task migration mechanisms.

To support the modelling and simulation of task migration in Sesame, we have modified its structure to resolve the constraint mentioned above. More specifically, a new middleware layer which is in charge of run-time task migration has been added to the Sesame framework. This middleware layer, called the Run-time Task Migration Middleware (RTMM), removes the direct mapping relationship between applications and hardware resources, as shown in Figure 2. To coordinate task migration-related activities at the architecture level on behalf of the RTMM, a Run-time Resource Scheduler (RRS) module is provided in the architecture model layer. This RRS manages the hardware resources either in a centralized or distributed (in large-scale systems) manner. It takes care of collecting statistics (e.g., performance of each application, system execution information, etc.) from the underlying system during the simulation process, triggering the RTMM layer to make a decision when task migration is needed and sending migration commands to specific processors based on the decision of the RTMM. To facilitate the simulation of different task migration mechanisms, we provide a task migration library that implements several *migration micro instructions* as shown in Table 1. Using these micro instructions, different migration mechanisms can easily be modelled. The first two instructions are designed for systems that use shared memory whereas the other two instructions are used for message-passing systems. In these instructions, the parameter *mig_mode* is used to indicate different migration schemes like migrating task code only, task context only or both. Figure 3 illustrates how the micro instructions can be used in the RRS to support task migration for a system with shared memory. After having received the new mapping scheme calculated by the RTMM, the RRS will start

[2]The binding of tasks and communications to the underlying hardware resources.

Table 1: Micro instructions provided in the task migration library

Instruction	Parameters
MIG_STORE	old_pe, address_shmem, mig_mode, mig_datasize
MIG_LOAD	new_pe, address_shmem, mig_mode, mig_datasize
MIG_SEND	old_pe, new_pe, mig_mode, mig_datasize
MIG_RECEIVE	new_pe, old_pe, mig_mode, mig_datasize

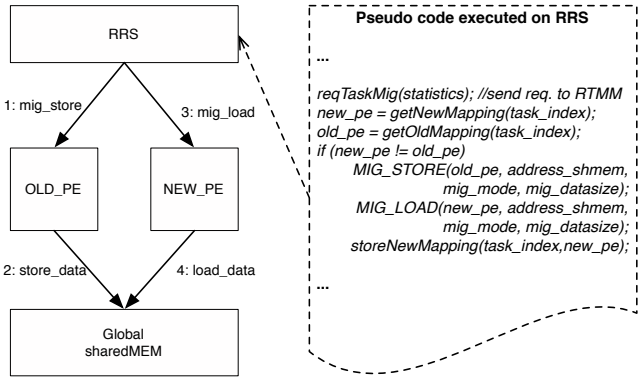

Figure 3: A simple example of task migration implementation on a shared memory system

the task migration process for the task(s) that will be migrated by issuing the micro instructions according to the migration mechanism implemented in the architecture. Here, the MIG_STORE triggers the storing of all migrating data into shared memory, while the MIG_LOAD triggers the loading of this data at the destination processor. For message-passing systems, the MIG_SEND and MIG_RECEIVE micro instructions will be sent to the processors from/to which a task is migrated, which will initiate a message-passing data transfer (realizing the actual migration) between these two processors.

As mentioned before, task migration can be implemented for different purposes such as a violation of application performance constraints, load balancing, fault tolerance and so on. To trigger task migrations, our framework supports different types of approaches that can be implemented in different layers, ranging from the application level to the architecture level. For example, at application level, explicit migration check-points can be inserted in the application code. In this case, the task migrations are controlled by the application designer. At the architecture level, each processor could also issue task migration requests to the RRS, triggered by e.g. the detection of undesired (execution) behavior like a timing violation, hardware fault or overheating. Beside these, the RRS can also trigger a task migration based on the statistics it has collected. In our framework, the task migration process is performed by means of coordinated actions between the RTMM and RRS. The exact responsibilities and workflow for each of these two components is shown in Figure 4. At run time, the RRS will continuously monitor the execution of applications and collect the running statistics of the target system. Whenever there is a pre-defined migration condition detected (e.g., a performance deadline violation), the RRS will send a task migration request to the RTMM. Currently, our simulation framework will stall all application processes until the migration

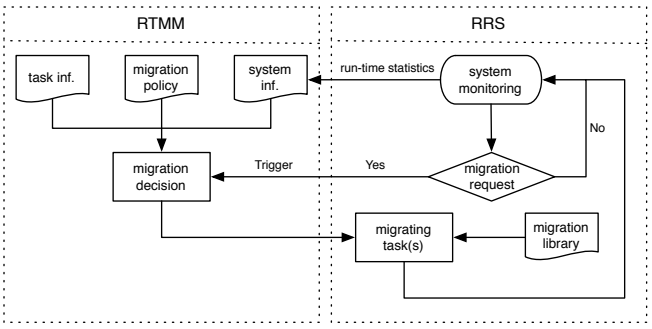

Figure 4: The task migration workflow of our framework

decision has been taken. The RTMM, after receiving a migration request from the RRS, will make a migration decision based on the task migration policy implemented, information of active tasks and the execution statistics collected by the RRS. Here, the migration decision includes the information of which task(s) should be migrated and where the task(s) should be migrated to. It does not involve information about how the task will be migrated. This is under the control of the RRS. Given this task migration decision, the RRS will start the task migration events according to the migration mechanism that has been implemented using the migration micro instructions.

Using the extended Sesame simulator, it is possible to evaluate the impact of task migrations on system performance for different MPSoC architectures. To achieve this flexibility, the following support for modelling and simulating different task migration-enabled architectures is available at each layer.

- **Application model layer:** Since we are targeting the streaming application domain in this work, applications are modeled using a process network (see e.g. the application model in Figure 1) which can be implemented in any high level programming language. To emit events to the architectural model, the application processes are annotated. By default, the Sesame framework supports the generation of read, write and execute events [23]. To support the study of application-level task migration mechanisms, we have added the possibility to instrument the code of processes such that special task migration events can be generated, which trigger task migrations in the RTMM. Besides the (instrumented) application source code, a structural application description (using an XML-based language) is provided to the simulator. This description includes the topology of the processes in the target application(s) and the execution parameters of each process.

- **Mapping layer:** This layer determines the mapping of processes (i.e. their event traces) onto architecture model components by dispatching application events to the correct architecture model component, as can be seen in Figure 2. The mapping also includes the mapping of communications at application level onto communication resources in the architecture model. The mapping layer has two additional purposes. First, the event dispatch mechanism in the mapping layer provides a variety of static and dynamic policies to schedule application tasks (i.e., their event traces) that are mapped onto shared architecture model components. Second, the mapping layer is also capable of dynamically transforming application events into (lower-level) architecture events to facilitate flexible refinement of architecture models [23]. In the extended Sesame simulator, the dispatching of trace events to architecture components is now controlled together with the RTMM / RRS tandem. The RTMM forwards events from the mapping layer to the RRS in the architecture model layer, where the latter is in charge of actually dispatching the trace events to the processor component onto which the application task in question is currently mapped. The mapping description that acts as input for this mapping layer includes the task migration related parameters like the minimal task context size and compiled task code size.

- **RTMM layer:** In this layer, the migration policy (algorithm) is implemented based on the goal of task migration. The policy defines *which* task(s) should be migrated and *where* the task(s) should be migrated to. The designer can implement different policies like the task remapping algorithms proposed in [26, 25] to test which one is the best for the design goal at hand.

- **Architecture model layer:** This layer models the (non- functional behavior of the) MPSoC hardware architecture, and can be generated using a system library that provides the template models for different components like processors, memories, communication channels and interconnects, etc. Also, designers can add and customise their own system components. To support task migration, the RRS component should be integrated into the architecture model. We also provide a template RRS implementation. In this template, one could use the micro instructions as described before to support different task migration schemes based on the target system architecture. Besides the architecture model, an architecture description file should be provided. It describes the structure of the architecture and includes the non-functional properties of hardware components like frequency, power, bandwidth/latency of communication channels, and so on.

4. TASK MIGRATION CASE STUDIES

In this section, we present two case studies in which we model task migrations in two very different systems, shown in Figure 5, to demonstrate the flexibility and wide application scope of our simulation framework. The target applications used in our experiments belong to the multi-media application domain. Each application has a (soft) real-time performance constraint which can be used to trigger task migrations as shown in the first experiment. For the application and system architecture description, the parameters needed for simulation are listed in Table 2. If needed, these parameters can be calibrated by designers using low(er)-level simulators or measurements on real systems. We would like to stress that this paper does not focus on the actual assessment of state-of-the-art task migration policies, but instead our aim is to demonstrate the flexibility and wide application scope of our simulation framework. Therefore, in the two case studies, we have chosen to implement only relatively simple task migration policies in the RTMM. The details of each of the two experimental cases will be explained in the following subsections.

4.1 A Heterogeneous UMA MPSoC

4.1.1 Target application and system architecture

In this experiment, the target application is a Sobel filter for edge detection in images (frames) which contains 6 processes (including 2 IO processes) and 6 communication channels between processes

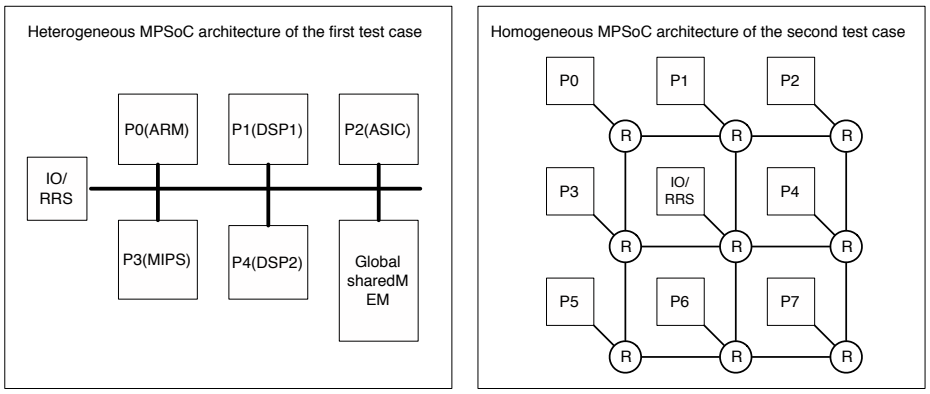

Figure 5: The MPSoC architectures used for the case studies

Table 2: The parameters for application and architecture description in the simulator

Parameters	Explanation
T_i^j	execution cycles of task i on processor j
CS_i^j	code size of task i on processor j
TC_i^j	minimal task context size of task i on processor j
S_m, S_b	size of memory m or buffer b in the system
B_m, B_c	bandwidth of memory m or comm. channel c
L_m, L_c	latency of memory m or comm. channel c
F_k	frequency of hardware component k on the system

in the application model. The target MPSoC system is shown in the left part of Figure 5. In the heterogeneous MPSoC, 5 processors with different architectures are connected by a bus. A global memory and a IO processor are shared by these processors. The RRS has been integrated in the IO processor. The IO processor can collect application performance statistics like Frame Execution Time (FET) at the end of each processed frame (i.e., the time between a frame is read and written by the IO processor). Based on these statistics, the RRS can trigger a task migration event when needed. Initially, all processes except the two IO processes in the Sobel application are mapped onto processor *p0* of the heterogeneous MPSoC (the IO processes are mapped onto the IO processor).

4.1.2 Migration mechanism and policy

As the target architecture in this experiment is a heterogeneous MPSoC with shared memory, the binary code of each task for each processor might be different. Here, we assume that the compiled code of each task for each processor is preserved in the global shared memory. Under this assumption, we have modeled a *task recreation* mechanism [11] in this experiment to support task migration. During task migration, the migrating task will be killed on the original processor and the task state information will be saved in global memory. The destination processor will load the binary code and state information from shared memory to restore the task. Also, the communication channels connected with the migrating task will be redirected to the new processor by remapping them.

Regarding the task migration policy in this experiment, we have modeled the following two algorithms to make the task migration

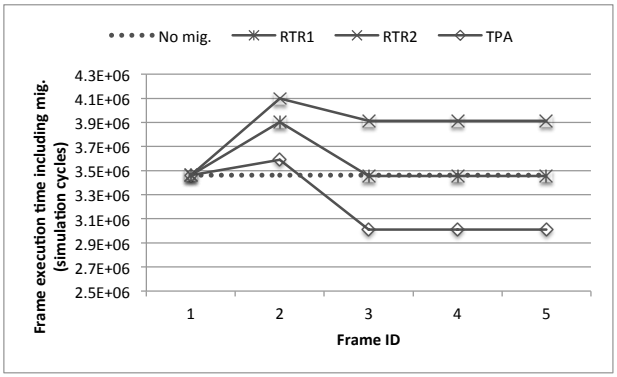

Figure 6: Task migration impact on application performance for the heterogeneous MPSoC

decision: a Random Task Remapping (RTR) which generates a random mapping for task remapping[3], and Task Processor Affinity (TPA) [20] which uses the affinity between tasks and processors to greedily determine a mapping without considering resource utilization. If the new mapping is different from the initial mapping mentioned above, then the system will enter the task migration state and restart the application after the migration process has finished.

4.1.3 Experimental results

In this experiment, we use a single picture as the continuous input stream of frames for the Sobel application. The migration trigger in this experiment is a violation of the application performance constraint as mentioned before. To this end, we have set a performance constraint for Sobel, using the Frame Execution Time (FET) metric, such that it enforces a task migration request after the first frame has been processed. The migration request is handled by the RTMM, which subsequently applies the implemented migration policy to make a task migration decision. During this process, all the tasks of the Sobel application will stall and wait for the task migration decision. After the migration has finished, the system continues to process the subsequent frames and monitor the execution of the application.

Figure 6 shows the experimental results of the migration impact

[3]RTR randomly decides which task(s) should be migrated and where the migrating task(s) should be migrated to.

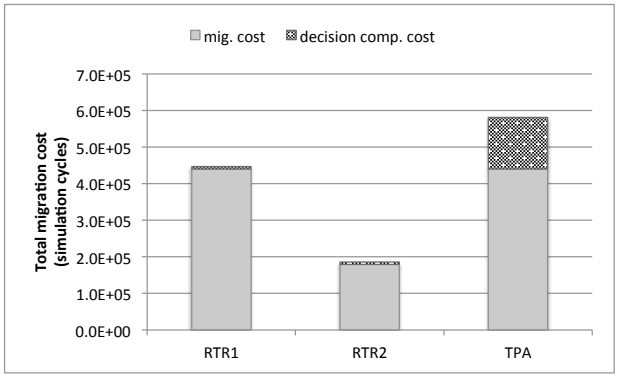

Figure 7: Task migration cost of different algorithms on the heterogeneous MPSoC

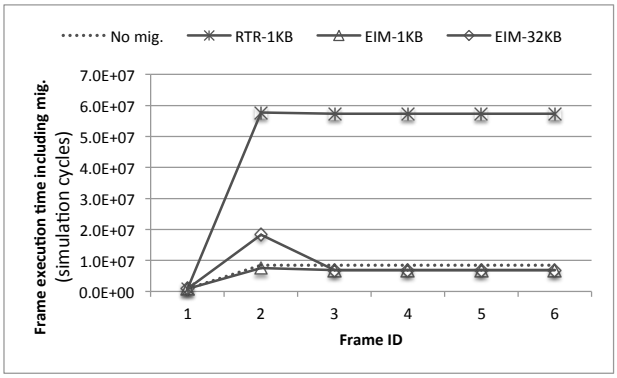

Figure 8: Task migration impact to application performance on the homogeneous MPSoC

to the application performance by using different task migration policies (algorithms) for our heterogeneous MPSoC. The marked lines labeled as *RTR1* and *RTR2* are the results of migrating tasks based on two different migration decisions computed by the RTR algorithm. The dotted line represents the execution using the initial mapping and without task migration. In Figure 7, we also show the task migration overhead for each policy, which includes two main elements: the computational cost of the task migration decision and the task migration cost itself. The computational cost of the task migration decision has been measured on a real CPU and then normalised to the simulation frequency of our simulator. In Figure 6, we clearly notice the task migration taking place after the first frame since the higher FET values for the second frame include the task migration overhead. After the second frame, the FET values stabilize again, i.e., no further task migrations are triggered. From the results, we can also see that the migration cost of *TPA* is the highest among three task migration scenarios. Here, *RTR2* has migrated 2 tasks, whereas *RTR1* and *TPA* both migrated 4 tasks. Consequently, *RTR2* has the smallest task migration overhead overall. However, after migration, the resulting mapping as derived by *TPA* clearly shows the best performance.

4.2 A Homogeneous NORMA MPSoC

4.2.1 Target application and system architecture

The applications used in this experiment are three typical multimedia applications: an M(otion)JPEG encoder, an MP3 decoder, and a Sobel filter as mentioned in the previous experiment. The application model of the MJPEG application contains 8 processes and 18 communication channels, and the MP3 application contains 27 processes and 52 communication channels. In total, there are 41 processes and 76 channels that need to be mapped onto the underlying hardware resources.

With respect to the target system in this experiment, the architecture is shown in the right part of Figure 5. In this system, 8 homogeneous processors and an IO processor are connected by a 2D mesh NoC. Similar to the system described in the last experiment, the RRS has also been integrated into the IO processor. Initially, we again map all processes except the IO processes onto processor *p0*.

4.2.2 Migration mechanism and policy

In this experiment, the target system architecture is a homogeneous MPSoC system with private memories connected to the processors (i.e., no remote memory access). As all processors have the same architecture, a task's context can be shared among processors. Therefore, we have modeled a *task replication* mechanism [11] to support task migration in this system. The idea is that each processor on the system has a replica of all tasks. Only one copy of a task can be active and running on a specific processor while the other copies are suspended and reside in memory of the other processors. When a task migration is needed, the task is suspended in the source processor and resumed in the destination processor, using the context of the migrating task. Also, the communication channels connected to the migrating task will be redirected to the new processor by the RRS. So, using this migration mechanism, only the context of the migrating task needs to be migrated between processors. This greatly reduces the communication overhead of task migration at the cost of increased memory usage because of the storage of task copies.

The task migration policies used in this experiment are slightly different than in the previous experiment. Since the target architecture is a homogeneous MPSoC, each task has the same task execution time on each processor on the system. Consequently, the TPA algorithm would not be very effective in this case. As a substitute, we have modeled the Energy-aware Iterative multi-application Mapping (EIM) algorithm from [25], where we have disregarded its energy constraint.

4.2.3 Experimental results

Figure 8 shows the results of the migration impact to the application performance for our homogeneous MPSoC. In this experiment, the three afore-mentioned applications are mapped onto the target system. For the purpose of results comparison, we also use the concept of *frame* to define the workload of applications. Here, we combine one unit workload (e.g., one picture) of each application together as one frame for all applications. In this case, the frame execution time of multiple applications is defined as the maximal frame execution time among applications (each application processes its own unit of workload). For example, the frame execution time F of our three target applications for processing one frame workload is represented as Equation 1.

$$F = max(F_{mjpeg}, F_{sobel}, F_{mp3}) \quad (1)$$

where F_{mjpeg}, F_{sobel} represents the frame execution time of processing one picture for MJPEG, Sobel respectively and F_{mp3} means the execution time of processing one short piece of encoded MP3 song.

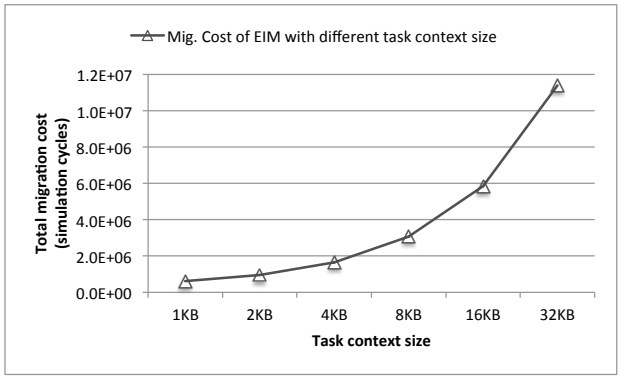

Figure 9: Task migration cost of algorithm EIM with different task context size on the homogeneous MPSoC

Similar to the previous experiment, a single input frame will be reused continuously to act as input stream for each application. In this experiment, task migration is used for dynamic resource reallocation, and therefore task migration is triggered when the system workload changes. For example, a new application enters the system or an active application finishes and quits the system. In Figure 8, there is only a single application (MP3) active in the system during Frame ID '1'. At the end of this frame, two other applications, namely Sobel and MJPEG, are activated on the system and execute from frame '2' until '6'. This means that a task migration is triggered during Frame ID '2', as is clearly visible in Figure 8. This figure shows the results for both the *RTR* and *EIM* migration decision algorithms. For EIM, the migration impact with different task context sizes have been considered (a 1KB and a 32KB context). From the results shown in Figure 8, we can see that the applications show poor performance after task migration when applying *RTR*. This can be explained by the fact that the communication costs of the NoC are high, especially for the applications that are communication intensive. In our case, it is better to map the MJPEG and Sobel applications each onto a single processor to reduce the communication cost. However, the RTR algorithm will generate mapping decisions without considering the communication cost at all. On the other hand, by applying the *EIM* algorithm, the final mapping has good performance behavior. Moreover, comparing the *EIM-1KB* and *EIM-32KB* curves, it can be seen that the application performance during task migration (Frame ID '2') can be substantially influenced by the task context size. To study how the task migration overhead is affected by the size of the migrating data in more detail, we have measured the migration overhead for different task context sizes. The results of this experiment are given in Figure 9. Clearly, the cost of transferring task context on our target homogeneous system linearly increases with the size of task context. However, in Figure 9, the task migration overhead also includes the computation cost of the EIM algorithm. From this figure, we can see that the task migration cost increases slowly with the task context size when it is under 8KB. However, when the task context size is bigger than 8KB, the task migration cost has a near linear relationship with the task context size. The reason is that when the task context size is small (like below 8KB in our test case) the task migration cost is dominated by the computation cost of the EIM algorithm. However, when the task context size increases to a certain amount, the cost of transferring task context dominates the task migration cost.

5. RELATED RESEARCH

Task migration has been traditionally studied in distributed systems for dynamic load balancing [29, 15, 8]. However, with the increasing popularity of MPSoCs in modern embedded systems, task migration has also gained research attention in this domain and has been studied for different purposes. For the purpose of thermal balancing, Cuesta et al. [10] provide three task migration polices to optimize the thermal profile of MPSoCs by dynamically balancing the weight of the on-chip thermal gradients, maximum temperature and effect of the underlying floorplan on heat dissipation properties of each core. In the work of [19], task migration-based thermal balancing policies are proposed to modulate power distribution between processing cores to achieve temperature flattening. The authors in [12] use proactive task migration among neighboring cores to balance the thermal profile for many-core systems. For the purpose of load balancing, in [5], task migration combined with intelligent initial placement are used to maintain load balancing in the MPSoC system. The authors in [6] analyze the impact of task migration in a NoC based MPSoC system. In their work, task migration is triggered after the resource allocation heuristic which tries to balance the system on demand is executed. To support fault tolerance on MPSoC systems, task migration is also a required technique [17, 9]. The idea in [17] is to improve dependability of the system by exploiting the migration method in case of run-time faults in the processing cores. In [9], a system-level fault-tolerance technique for application mapping, which aims at optimizing the entire system performance and communication energy consumption, is proposed. To this end, application components running on a faulty core are migrated altogether to available non-employed spare cores.

In the context of task migration mechanisms supported in MPSoC systems, quite some work has been done on task migration at application level, middleware level and architecture level. In [5], the authors propose a user-managed migration scheme based on code checkpointing and user-level middleware support as an effective solution for many MPSoC application domains. The work in [1, 24] implements task migration in a middleware layer which is built on top of the uClinux operating system running on a prototype multicore emulation platform. To support heterogeneity in task migration, [22] provides a middleware, called Low Level Virtual Machine (LLVM), to postcompile the tasks at runtime depending on their target processor. At the architecture level, [2] discuss the possible architectural support for MPSoC systems to allow dynamic task migration. In [4], the authors propose a hybrid memory organization for NoC-based MPSoC systems as the way to minimize the energy spent during the code transfer when task migration or dynamic task allocation needs to be performed.

With regard to task migration simulation, in [28], the authors extended Sesame to have a system-level simulator for run-time task mapping in the context of reconfigurable systems. In their simulator, tasks can be migrated between a general purpose processor and a reconfigurable accelerator which enables the system to be more efficient in terms of various design constraints such as performance, chip area and power consumption. However, no details of the task migration implementation are shown in this paper. Different with this simulator, our proposed simulator provides a general purpose task migration framework which is not limited by the target architecture, the task migration purpose and task migration mechanism. To the best of our knowledge, this paper presents the first effort in the direction of establishing a generic simulation infrastructure that allows for the efficient exploration of a wide range of migration mechanisms and policies in MPSoCs.

6. CONCLUSION

Task migration is a useful technique that can be used in MPSoC systems for different purposes such as load balancing, thermal balancing, fault tolerance, improving system energy efficiency and so on. Therefore, investigating the suitability of specific task migration schemes for a target system architecture is an important design step that needs to be addressed as soon as the early stages of design. For this purpose, in this paper, we have presented a high-level simulation framework that allows for simulating and exploring different task migration mechanisms and policies for a wide range of different system architectures. Using two case studies, we have demonstrated the flexibility and wide application scope of our simulator. To this end, the case studies evaluate different task migration policies and mechanisms for vastly different target architectures. The experiments point out that our task migration simulator can provide designers with useful insights on the suitability of a specific migration scheme for the target system and allows for exploring different migration policies.

7. REFERENCES

[1] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008:9:1–9:15, Jan. 2008.

[2] A. Aguiar, S. J. Filho, T. G. dos Santos, C. Marcon, and F. Hessel. Architectural support for task migration concerning mpsoc. *SBC*, page 169, 2008.

[3] Y. Artsy and R. Finkel. Designing a process migration facility: The charlotte experience. *Computer*, 22(9):47–56, Sept. 1989.

[4] D. Barcelos, E. W. Brião, and F. R. Wagner. A hybrid memory organization to enhance task migration and dynamic task allocation in noc-based mpsocs. In *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design*, SBCCI '07, pages 282–287, New York, NY, USA, 2007. ACM.

[5] 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*, DATE '06, pages 15–20, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[6] E. W. Briao, D. Barcelos, F. Wronski, and F. Wagner. Impact of task migration in noc-based mpsocs for soft real-time applications. In *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pages 296–299, Oct 2007.

[7] P. Bungale, S. Sridhar, and V. Krishnamurthy. An approach to heterogeneous process state capture/recovery to achieve minimum performance overhead during normal execution. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, April 2003.

[8] H. Chang and W. J. B. Oldham. Dynamic task allocation models for large distributed computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(12):1301–1315, Dec 1995.

[9] C.-L. Chou and R. Marculescu. Farm: Fault-aware resource management in noc-based multiprocessor platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[10] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii. Adaptive task migration policies for thermal control in mpsocs. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 110–115, July 2010.

[11] A. Elantably and F. Rousseau. Task migration in multi-tiled mpsoc: Challenges, state-of-the-art and preliminary solutions. Technical report, Marseille, France, June 2012.

[12] Y. Ge, P. Malani, and Q. Qiu. Distributed task migration for thermal management in many-core systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 579–584, June 2010.

[13] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill Education (India) Pvt Limited, 2003.

[14] J. Jahn, M. Faruque, and J. Henkel. Carat: Context-aware runtime adaptive task migration for multi core architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[15] C. Lu and S.-M. Lau. A performance study on load balancing algorithms with task migration. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 357–364 vol.1, Aug 1994.

[16] N. Manchanda and K. Anand. Non-Uniform Memory Access (NUMA). *New York*, 1, 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 *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 517–524. IEEE, 2012.

[18] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 986–991, 2003.

[19] F. Mulas, M. Pittau, M. Buttu, S. Carta, A. Acquaviva, L. Benini, D. Atienza, and G. De Micheli. Thermal balancing policy for streaming computing on multiprocessor architectures. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 734–739, March 2008.

[20] V. Nollet. *Run-time management for future MPSoC platforms*. PhD thesis, PhD thesis, Eindhoven University of Technology, 2008.

[21] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234–239 Vol. 1, March 2005.

[22] L. Ost, S. Varyani, L. S. Indrusiak, M. Mandelli, G. M. Almeida, E. Wachter, F. Moraes, and G. Sassatelli. Enabling adaptive techniques in heterogeneous mpsocs based on virtualization. *ACM Trans. Reconfigurable Technol. Syst.*, 5(3):17:1–17:11, Oct. 2012.

[23] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[24] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 59–64, Oct 2007.

[25] W. Quan and A. D. Pimentel. An iterative multi-application mapping algorithm for heterogeneous mpsocs. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, pages 115–124. IEEE, 2013.

[26] W. Quan and A. D. Pimentel. A scenario-based run-time task mapping algorithm for mpsocs. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 131:1–131:6, New York, NY, USA, 2013. ACM.

[27] M. Richmond and M. Hitchens. A new process migration algorithm. *SIGOPS Oper. Syst. Rev.*, 31(1):31–42, Jan. 1997.

[28] K. Sigdel, M. Thompson, A. D. Pimentel, C. Galuzzi, and K. Bertels. System-level runtime mapping exploration of reconfigurable architectures. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[29] T. Suen and J. Wong. Efficient task migration algorithm for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 3(4):488–499, Jul 1992.