Semi-partitioned scheduling and task migration in dataflow networks

Cannella, E.

Chapter 4

Process Migration Mechanism in a Mapped PPN

Most of the work presented in this chapter has been published in [CDM$^+$12].

IN Chapter 3 we investigated several approaches that allow to execute applications specified as PPNs on Network-on-Chip (NoC) based MPSoCs. The approaches presented in Chapter 3 represent alternative implementations of the first component of our proposed middleware[1], introduced in Section 1.4.1 and depicted in the software stack of Figure 1.6 on page 20. The first component of our middleware, *PPN Communication*, allows PPN processes to communicate in NoC based MPSoCs, regardless of the mapping of processes to the available PEs. This is a necessary property in order to achieve system adaptivity by means of process migration. However, it is also necessary to define how to perform the transition between the current mapping and the next desired one. That is, we have to provide a mechanism to perform process migration. In our approach, this mechanism is implemented by the second component of our proposed middleware shown in Figure 1.6, namely *Process Migration*, which is proposed in this chapter. We recall that the techniques proposed in Chapter 3 and this chapter are aimed at **best-effort systems**.

4.1 Problem Statement

In this chapter, we address the problem of defining and implementing a process migration mechanism, targeted at PPN processes on NoCs, that satisfies the following three requirements:

[1]We recall that the proposed middleware is aimed at achieving system adaptivity in embedded NoC based MPSoCs by exploiting process migration.

1. Once the process migration is triggered, it has to be completed within a certain, known time frame. We refer to this property as *predictability*.
2. The process migration can be triggered in the system at any time. We consider this requirement because we want to cover the scenario in which process migration is needed in response to a hardware fault, for which the moment of occurrence is unknown.
3. The code used to allow process migration has to be generated automatically, without the manual intervention of the designer. This is needed to relieve designers from the time-consuming and error-prone task of inserting the code necessary to allow task migration by hand.

4.2 Contributions of this Chapter

We devise and develop a predictable process migration mechanism that allows run-time process remapping among the tiles of the NoC, which is a fundamental requirement for system adaptivity. The peculiarity of our solution is that, leveraging the PPN operational semantics and process structure, the migration can actually start at any point during the execution of the main body[2] of a process without the need of moving a large state. Moreover, an upper bound of the process migration overhead can be found, based on the PPN topology and FIFO buffer sizes. Finally, the code used to allow process migration is minimally invasive with respect to the original code structure and can be generated in a completely automated way.

4.3 Related Work

Run-time resource management is a widely studied topic in general purpose distributed systems scheduling [CJK88]. In particular, process migration mechanisms [Smi88, MDP$^+$00], have been developed and evaluated in this context to enable dynamic load distribution, fault resilience, and improved system administration and data access locality. In recent years, run-time management is gaining popularity and finding applications also in multiprocessor embedded systems. This domain imposes tight constraints, such as cost, power, and predictability, that run-time management and process migration mechanisms must consider carefully. [NVC10] provides a survey of run-time management examples in state-of-the-art academic and industrial MPSoCs, together with a generic description of run-time manager features and design space.

Our work is focused on a specific component of run-time management strategies, namely the process migration mechanism. Papers which specifically address process (or task) migration implementation in MPSoCs can also be found in the literature. The closest to our work is [Gab09], in which the goals of scalability and system adaptivity are achieved through a distributed task migration decision policy over a

[2]With the term "main body" we mean that a migration can happen during most of the execution time of a process. This concept will be explained in greater detail in Section 4.5.1.

purely distributed-memory multiprocessor. Similar to our approach, their platform is programmed using a process network MoC. However, in their approach the actual task migration can take place only at fixed points, which correspond to the communication primitive calls. Our approach, instead, enables migration at any point in the execution of the main body of processes. This leads to a faster response time to migration decisions, which is preferable for instance in case of hardware faults.

Other task migration approaches are explained and quantitatively evaluated in [BABP06] and [AACP08]. Dynamic task re-mapping is achieved at user-level or middleware/OS level, respectively. In both these approaches, the user needs to define checkpoints in the code where the migration can take place. This can require a consistent manual effort from the designer which is not needed in our approach. Moreover, a relevant difference with respect to our work is the inter-task communication implementation, which exploits a shared memory system. We argue that our approach, which uses purely distributed memory, can perform better in emerging MPSoC platforms since it provides better scalability.

Finally, the authors in [NKG+02] propose an MPSoC hardware and software architecture template that allows the system to change the mapping of applications' tasks at run-time. Compared to the work presented in this chapter, their approach uses a distributed *shared* memory to implement inter-task communication. That is, in their execution platform a process can read data tokens directly from the memory of a different processor. As mentioned earlier, by contrast, our approach uses completely distributed memories, with the goal of providing better scalability in emerging MPSoCs.

It is worth noting that the process migration mechanism presented in this chapter has been devised and implemented within the MADNESS EU FP7 Project, in close cooperation with DIEE, University of Cagliari, and ALaRI, Faculty of Informatics, University of Lugano. The mentioned migration mechanism, initially presented in [CDM+12], has been used as base infrastructure for other works within the MADNESS project [MTR+12, DCT+13]. In particular, in his PhD dissertation [Der15], Derin has proposed migration techniques that are complementary to the one described in this chapter. In Chapter 5 of [Der15], the author makes the following contributions which are closely related to the process migration mechanism of [CDM+12].

- First, he shows how software self-testing routines, capable of detecting faults, can be coupled with the migration mechanism of [CDM+12].
- Second, he devises a *task migration hardware* module that is included in each tile of the NoC-based MPSoC. In case of a fault, this task migration hardware is in charge of extracting the state of processes from the faulty tile to make it available to the resource manager of the MPSoC.
- Third, he proposes an alternative way of handling faults at the PPN application level. In [CDM+12], when a PPN process is interrupted by a fault, the execution is resumed on another tile by rolling back to the beginning of the interrupted iteration of the PPN process. Chapter 5 of [Der15] provides also a different way of fault recovery, that resumes the execution on a different tile by rolling forward to the PPN process iteration that *follows* the interrupted iteration. This yields to

a simpler task migration hardware implementation, although the application output can be temporarily incorrect.

The remainder of the chapter is organized as follows. In Section 4.4, we summarize the assumptions of our system adaptivity approach and we provide an overview of our proposed process migration mechanism. Then, in Section 4.5, we describe the proposed process migration mechanism in greater detail. Finally, Section 4.6 concludes this chapter reporting the experiments performed to test our process migration mechanism, and the achieved results.

4.4 Proposed Migration Approach

In the following paragraphs we recall some assumptions, related to the structure of the MPSoCs that are considered by our approach, that have an important influence on our proposed process migration mechanism. Then, we provide an overview of the migration mechanism itself.

The starting assumption of our system adaptivity approach, as depicted in the right part of Figure 1.6 on page 20, is that we target an MPSoC composed of tiles, connected by a NoC, with completely distributed memories and no direct remote memory access. This means that the processing element of a tile can only directly access the content of its own local memory. All the communication and synchronization between processes mapped on different tiles can only happen using messages sent over the NoC.

Our approach for realizing system adaptivity consists of deploying the processes of the application(s) modeled as PPNs over the NoC-based MPSoC and allowing their run-time remapping to adapt the system to the changing operating conditions such as variation in quality of service requirements, availability of resources, or power budget constraints. In particular, system adaptivity in our system is supported by using the dedicated middleware highlighted in the software stack in the left part of Figure 1.6 on page 20. For reasons of convenience, we copy Figure 1.6 on page 20 in Figure 4.1.

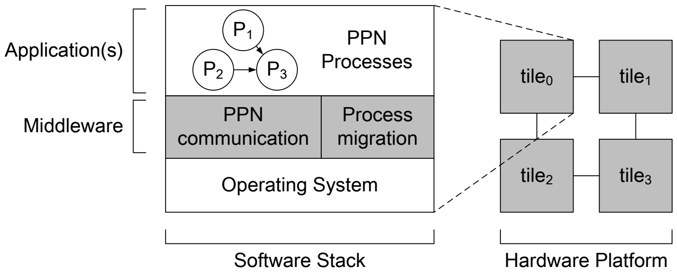


Figure 4.1: *Software stack (left) proposed to achieve adaptivity in BE systems. The middleware layer is denoted by the shaded area. The stack is deployed on each tile of the hardware platform (right).*

At the top of the software stack, applications are specified as a set of PPN processes, which are implemented as separate threads. An example of a thread representing a

PPN process is given in Figure 2.3(b) on page 29. However, in our work the basic structure of PPN processes will be modified to ease the realization of a predictable process migration mechanism, as described in Section 4.5.

At the bottom of the software stack in Figure 1.6 on page 20, the operating system (OS) is responsible for all kinds of process management (process creation, deletion, setting its priority, suspending or resuming it). These features are essential for the run-time management of the system, and in particular to perform process migrations. Moreover, each processor has multi-tasking capabilities thanks to the OS. In case of *many-to-one* mapping, i.e., when more than one process are mapped on the same processing element (PE), the scheduling is data-driven. This means that a process keeps executing successive iterations until it blocks in reading or writing (recall the PPN process structure of Figure 2.3(b) on page 29). When the process blocks, it yields the processor control to the next process in the ready queue in a round-robin fashion.

In between the applications and the operating system, in this thesis we propose the middleware which is highlighted in Figure 1.6 on page 20, which comprises two main components. The first one is *PPN communication*, which realizes the communication and synchronization between processes located in separate tiles, according to the PPN semantics. This middleware component has been already described in Chapter 3. The second component is *Process migration*, which is mainly responsible for the following activities, performed during the migration of processes:

- Coordinates the creation and deletion of processes among different tiles;
- Guarantees the correct transfer of the process' state during process migration.

The second component of our proposed middleware is the main subject of this chapter.

4.5 Process Migration

This section details the proposed mechanism to perform migrations of PPN processes over NoC-based MPSoC systems. It is a fundamental part of the middleware depicted in Figure 1.6 on page 20 because it defines how to perform run-time remapping of processes. This, in turn, allows designers to implement system adaptivity strategies.

The migration mechanism depends on the considered communication approach. As a starting assumption to devise the migration mechanism, we consider the *request-driven (R)* communication approach described in Section 3.4.3. This choice is made because the *R* approach leads to a considerably easier implementation of the migration mechanism since it requires less synchronization points. At the same time, it gives performance comparable to the other approaches for computation-dominant applications, as shown in Section 3.6.1.

We recall that to take into account the run-time remapping of processes over the NoC, each PE stores in its local memory a *middleware table* which is used during the conversion of the generic PPN communication primitives (i.e., *READ, WRITE* in Figure 2.3(b) on page 29) to the corresponding hardware platform primitives, such that the messages are sent to the right tiles in the system. A simple diagram showing the migration of a PPN process is depicted in Figure 4.2. An example of a middleware table generated for the initial mapping in Figure 4.2 is given in Table 4.1. For each

Table 4.1: *Middleware table example*

ch	prod(ch), cons(ch)	map(*prod(ch)*), map(*cons(ch)*)
1	P_1, P_2	$tile_0$, $tile_1$
2	P_2, P_3	$tile_1$, $tile_2$

channel of the PPN, the table lists which processes are the producer and consumer of that channel, together with the current mapping of producer and consumer processes in the system. Auxiliary information, for instance requests that are pending when a process migration is triggered, is also saved for each channel.

Mainly two kinds of process migration mechanism are considered in the literature, namely *process replication* and *process recreation*. In process replication, the program code of a migratable process is copied in each tile that may execute it, thereby creating replicas of the process. When a process needs to be migrated from one tile to another, the process is suspended on the first tile and restarted on the second. The state of the process must be copied from the first tile to the second because the process cannot be just restarted from scratch.

The second kind of process migration mechanism is based on the so-called *process recreation*. In this case, if a migration is needed, the process is killed on the initial tile (it runs) and created on another tile by moving both the process code and state. The OS/middleware in this case must support dynamic loading of processes to processors. This way, only one instance of the process code exist at a given time in the system.

The process replication mechanism is less efficient in terms of memory usage, compared to process recreation. Yet, it offers significant advantages such as easier implementation and faster migration procedure. Thus, for our proposed process migration mechanism we choose process replication because we aim at guaranteeing a quick completion of the migration procedure. Moreover, the memory constraint in our system is not critical.

Consider again the simple diagram in Figure 4.2, which shows the migration of a PPN process. Even though it is a simple example, it can be easily generalized for more complex PPN topologies. The diagram highlights the tiles involved in the process migration procedure, which are referred to as:

- the **source tile**, namely the tile which runs the process before the migration takes place;
- the **destination tile**, which is the tile that will execute the process after the migration;
- the **predecessor tile(s)**, which run(s) the predecessor process(es);
- the **successor tile(s)**, which execute(s) the successor process(es).

The structure of PPN processes, modified to allow migration at any point during the execution of the process main bodies, and the proposed process migration mechanism are presented in the following two subsections.

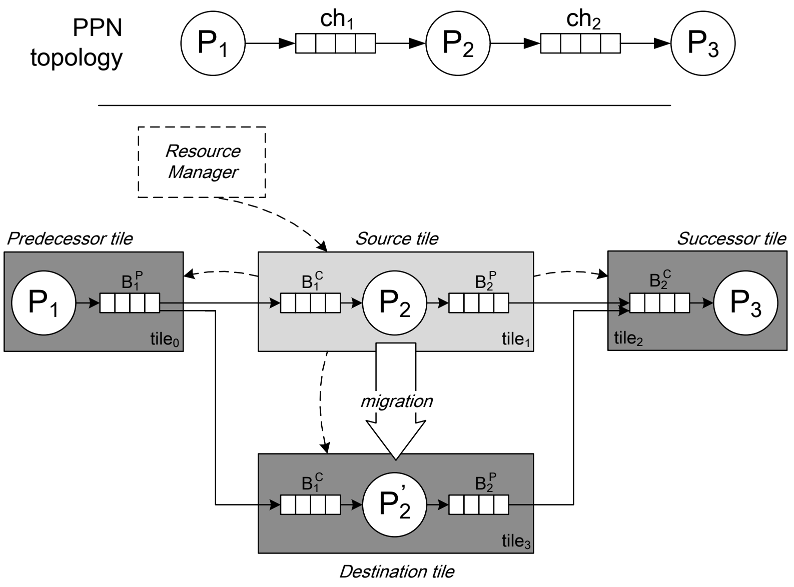


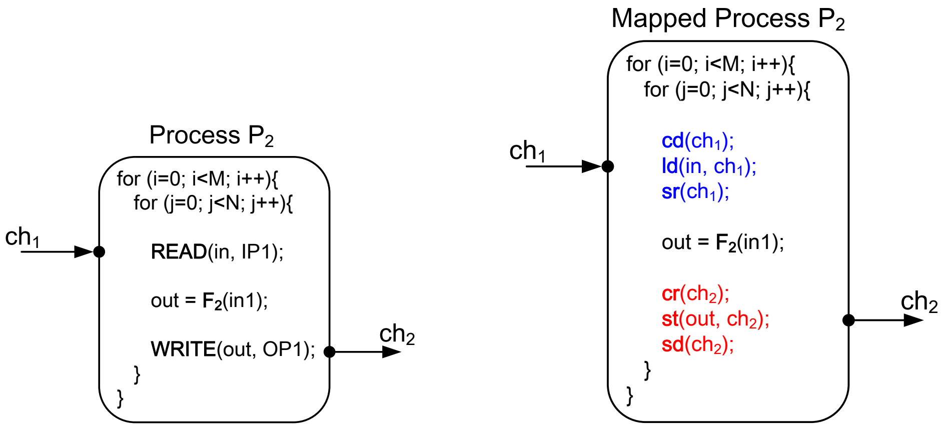
Figure 4.2: *Example of a migration procedure. The PPN topology considered in this example is shown in the top part of the figure. The initial mapping of this PPN is the following: process P_1 on tile$_0$, P_2 on tile$_1$, P_3 on tile tile$_2$. The resource manager (denoted by a dashed box) triggers the migration of P_2 from tile$_1$ to tile$_3$ by sending a specific control message to tile$_1$. This control message is forwarded by tile$_1$ to all the tiles involved in the migration. Control messages are represented by dashed arrows. To perform the actual migration, a few more steps are required. First, process P_2 is suspended on tile$_1$ and its iteration vector is transferred to tile$_3$. Second, the state of the input and output channels of P_2 on tile$_1$ (the content of B_1^C and B_2^P) are also moved to tile$_3$. Finally, the migration procedure is completed by starting the replica of P_2 on tile$_3$. This replicas is denoted by P_2'.*

4.5.1 Migratable PPN process structure

Our goal is to allow the migration to occur at any time during the execution of the process main body, which means that a migration can happen during most of the execution time of a process, as will be explained later in this section. In turn, this improves the latency incurred from the moment that a migration is triggered to its completion. To this end, we extend the NI of a tile with the ability to generate an interrupt for the processing element when a message with a specific tag is received. This extension is made because the detection of migration commands by polling at specific migration points in the code may cause undesired latency in the migration procedure.

In the scenario depicted in Figure 4.2, process P_2 is migrated from tile$_1$ to tile$_3$. The original structure of the code of P_2, as generated by the pn compiler, is reported in Figure 4.3(a). This structure of process P_2 hides all the details of the actual mapping of P_2 onto the execution platform.

In particular, to perform such mapping, the PPN communication primitives of P_2



(a) Structure of PPN process P_2. (b) Basic code structure of the mapped process P_2.

Figure 4.3: *Sub-figure (a) shows the structure of PPN process P_2 in Figure 4.2. Sub-figure (b) depicts the basic code structure used to map P_2 onto the considered execution platform. Notice that PPN communication primitives have been refined into several execution platform primitives.*

must be converted to (a set of) communication primitives of the execution platform. This conversion is a problem already studied in the literature [LvdWD01]. Typical communication and synchronization primitives of an execution platform are the following[3]:

- Check Data (**cd**): Checks if there are available data tokens in the considered FIFO buffer. Otherwise, it stalls the calling process.
- Check Room (**cr**): Checks if there is available space in the considered FIFO buffer. Otherwise, it stalls the calling process.
- Load Data (**ld**): Transfers a token from the considered FIFO buffer to the local space of the process.
- Store Data (**st**): Transfers a token from the local space of the process to the considered FIFO buffer.
- Signal Room (**sr**): After a *ld* operation, it signals that (additional) room is available in the considered FIFO buffer.
- Signal Data (**sd**): After a *st* operation, it signals that (additional) data is available in the considered FIFO buffer.

To derive an efficient mapping of PPN processes to an execution platform, designers have to consider the structure of the execution platform itself. For instance, it is fundamental to know whether the FIFO buffers which implement the channels of a certain process are located in a shared memory or in the local memory of the PE that executes that process.

In our approach, as shown in Figure 4.2, each process can only access the local memory of its tile. The transfer of tokens among tiles of our execution platform is

[3]Using the notations of [LvdWD01].

handled by the request-driven middleware approach, as mentioned earlier. Therefore, each process reads and writes tokens only from/to its local memory. In this scenario, the *READ* and *WRITE* PPN communication primitives are typically converted to the aforementioned primitives of the execution platform in the following way.

$$\text{READ} \implies cd \rightarrow ld \rightarrow sr \tag{4.1}$$

$$\text{WRITE} \implies cr \rightarrow st \rightarrow sd \tag{4.2}$$

Using the above conversion of communication primitives, we derive the implementation of PPN process P_2 onto the considered execution platform. The structure of such implementation is shown in Figure 4.3(b). We will refer to the structure shown in Figure 4.3(b) as *basic mapped process structure*. Since we require that migration may happen at any point within the execution of the processes main body, a modification of the process structure is required. In the rest of this section, we will explain why this modification is required and in what it consists.

In order to maintain correct functionality of the application, the state of the whole PPN must be consistent before and after a process migration has occurred. We divide the state of the PPN in two components, as follows:

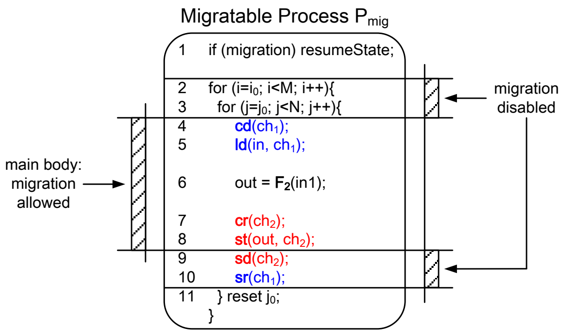
1. State of PPN processes. As explained in Section 2.1.3, the only internal state of a PPN process is its iteration vector $\vec{I}$, which represents the value of the *for*-loop iterator variables.
2. State of PPN channels. The state of a channel in the PPN is represented by the tokens which are currently stored in the FIFO buffers which implement that communication channel.

Regarding the second component of the PPN state listed above, note that a PPN channel *ch* is actually implemented by two FIFO buffers in the request-driven communication approach which is considered in this chapter. One of these buffers reside on the tile on which the producer of *ch* is mapped, whereas the other resides on the tile in which the consumer of *ch* is mapped (recall Figure 3.4 on page 56). Therefore, when migrating a process P from its source tile to its destination tile, two components of the PPN state have to be migrated:

ST1: The iterator vector of P. For instance, the iterator vector of the process depicted in Figure 4.4 is $\vec{I} = [i, j]$.

ST2: The state of the input and output channels of P residing on the source tile of P. This state is in fact represented by the content of the input and output FIFO buffers connected to P. For example, refer to Figure 4.2, which illustrates the migration of process P_2 from tile$_1$ to tile$_3$. If, when the migration is performed, FIFO buffers B_1^C and B_2^P on tile$_1$ contain tokens, their content has to be moved to the destination tile, tile$_3$, into the corresponding FIFO buffers.

Having defined the state that has to be transferred during a process migration, we comment and describe the migratable PPN process structure shown in Figure 4.4 in the following. We denote this migratable process as P_{mig}. When P_{mig} starts, in line 1 of Figure 4.4, it checks if the *migration* flag is set. If the checking is positive, then this means that a migration has been performed, so the process state is reloaded.



```
1    if (migration) resumeState;
2    for (i=i0; i<M; i++){
3      for (j=j0; j<N; j++){
4        cd(ch1);
5        ld(in, ch1);
6        out = F2(in1);
7        cr(ch2);
8        st(out, ch2);
9        sd(ch2);
10       sr(ch1);
11     } reset j0;
     }
```

Figure 4.4: *Structure of migratable process P_{mig}. Compared to the basic mapped process structure of Figure 4.3(b), the order of the execution platform communication and synchronization primitives is changed. This allows migrations to be performed at any point within the main body of the P_{mig} (lines 4-8).*

Both state components listed above (*ST1, ST2*) are transferred from the source tile to the destination tile upon migration. If the migration flag is false, then this means that the process P_{mig} starts from scratch, with empty input and output FIFOs and $i_0 = j_0 = 0$.

Lines 2 and 3 differ from the basic mapped process structure in Figure 4.3(b) because the iterators inside the *for* loops do not start from zero in case of migration. Instead, they start from the values i_0 and j_0, which represent the iteration at which the process was interrupted by the migration while running on the source tile. After the first complete execution of the inner *for* loop, starting from j_0, the value of j_0 is set to zero in line 11 such that the next execution of the inner loop starts correctly with $j = 0$.

Moreover, the order in which communication and synchronization primitives are executed in P_{mig} differ from the one used in the basic mapped process structure of Figure 4.3(b). In fact, the execution platform primitives that implement the PPN *READ* primitive of P_{mig} (i.e., *cd*, *ld*, and *sr*) are not executed in a continuous sequence. They are, instead, executed in lines 4, 5, and 10, respectively, with several other operations occurring between *ld* and *sr*.

The reordering of execution platform communication primitives has already been studied in [LvdWD01]. The work in [LvdWD01] defines rules by which the reordering preserves the correctness of the execution of the mapped PPN process. The migratable process structure P_{mig} in Figure 4.4 complies with the rules defined in [LvdWD01].

Recall that the actual *release* of memory locations is performed by the *sr* operation, which consumes the data token from the FIFO by increasing the read pointer. This operation takes place only outside the main body of P_{mig}, in line 10. Then, if a migration is triggered before the *sr* operation, P_{mig} can be correctly resumed on the destination tile since it will read again the same input token, because the read pointer is unchanged. Similarly, the *sd* operation that concludes the *WRITE* primitive is executed at the end of the mapped PPN process, outside its main body. Finalizing the

READ and *WRITE* operations at the end of an iteration allows the process migration to happen anywhere within lines 4-8 correctly. Note that, in case of multiple input or output channels, the *sd* and *sr* operations of all channels are performed together right after the main body of the process, in order to update the state of P_{mig} and of the FIFO buffers in the shortest possible time.

Process migration cannot happen in the migratable process P_{mig} within the lines 9-11 and 2-3 because that will cause an inconsistency in the state of the PPN. This is because lines 9 and 10 can be considered as an update of the output and input FIFOs state, while lines 11, 2 and 3 represent an update of the state of P_{mig}. If, for instance, a migration happens after the FIFO state update but before the iterator set update, the following scenario will occur: (i) the state of the input and output FIFOs connected to P_{mig} are modified as if the current iteration was successfully completed; (ii) P_{mig} restarts the current iteration from the beginning, because the iteration vector was not updated accordingly. This condition will certainly cause a deadlock. Although the process migration cannot happen within lines 2-3 and 9-11, note that these sections represent a minimal part of the process execution, because performing the *sd* and *sr* operations and updating the iterator set is a matter of a few simple instructions. Therefore, disabling the migration within these sections does not increase the migration latency significantly.

The principle behind the proposed migratable process structure is that the state of the PPN must be *consistent* and *up-to-date* when a migration is performed. This allows the PPN to correctly resume its execution, with the migrated process mapped on the destination tile. Leveraging the PPN process structure, our approach does not require the designer to specify the context that has to be transferred upon migration as in [BABP06]. This burden is neither moved to the OS/middleware level as in [AACP08]. Determining the state to be migrated is not needed because the PPN state simply consist of the two components (*ST1*, *ST2*) described above. Moreover, our approach does not need designer-generated checkpoints/migration points. The resource manager in Figure 4.2 can interrupt the process execution at any time during the execution of the process main body. The migrated process will then resume its execution from the beginning of the interrupted iteration. On the one hand, this implies that if the migration is triggered in the middle of the function execution, the time spent in computation since the start of the iteration is lost. On the other hand, this approach leads to a more efficient implementation and predictable migration response time, which we consider more important for our goals.

4.5.2 Process migration mechanism

The migration mechanism requires actions from all the tiles depicted in Figure 4.2. Note that, in the figure, a *resource manager* is in charge of taking the migration decision. How the resource manager makes this decision is out of the scope of this thesis because we focus only on the process migration mechanism itself. Our contributions are in fact complementary to other research works (see [DKF11, AK09, LKwP^{+}10, Gab09, SSHT06]) which provide techniques to determine if a process migration is necessary and/or beneficial and, in that case, the actual destination tile of the process that has to be migrated.

When a migration decision is taken by the resource manager, it initiates the migration by sending a specific control message to the source tile. The source tile then forwards this control message to the destination, predecessor and successor tiles to inform them that the migration procedure has been initiated.

The control messages which notify the involved tiles for the start of the process migration contain the ID of the migrated process and the new mapping of that process. On all of the involved tiles, and on the resource manager, the middleware tables are then updated taking into account the new mapping of the migrated process.

For each of the tiles involved in the migration procedure, the detailed list of required actions are explained below.

Actions on the source tile

The behavior of the source tile depends on whether the tile is functional or faulty.

- In case the source tile is **functional**, the migrating process is stopped on the PE of the tile and the two state components, *ST1* and *ST2* mentioned in Section 4.5.1, are moved to the destination tile. These state components are transferred by means of dedicated messages sent over the NoC. Moreover, the middleware table is updated as described above. The source tile takes also care of propagating the migration decision to the other tiles involved in the migration procedure. This propagation is depicted by the dashed arrows in Figure 4.2.
- In case the source tile is **faulty**, the actions described in the previous point are emulated by a dedicated hardware IP, as proposed in [DCT+13].

Actions on the destination tile

The destination tile receives a specific message for process activation. The migration procedure is handled by creating the required software FIFOs and by activating the replica of the migrated process using the corresponding OS call. Before the process replica is started, the *migration* flag is set to 1 so that the state of the migrated process is resumed (see line 1 in Figure 4.4). This implies that the input and output FIFOs connected to the migrated process are copied, and the iterator set (in the figure, i_0 and j_0) are set such that the execution starts from where it was suspended on the source tile. The middleware table is also updated in the way described above.

Actions on predecessor tile(s)

On these tiles, the only required step is the update of the middleware tables according to the new mapping of the migrated process. This way, new tokens meant for the migrated PPN process will be sent to the destination tile.

A corner case of the communication between the migrated process and its predecessor processes may happen when the migrating process has sent a *request* for new tokens just before the migration command arrives. For instance, it may happen that process P_2 in Figure 4.2 has sent a request for tokens to P_1 just before receiving the migration command from the resource manager. If that request has been served, before the migration command reached the predecessor tile, it means that new tokens are either traversing the NoC or they are already stored in the source tile. The predecessor tile in this case has to send another interrupt-generating message to the source tile, in order to force the forwarding of these data tokens to the destination tile.

Actions on successor tile(s)

Similarly, the successor tiles have to update their middleware tables so that successors of the migrating process will send new requests for data tokens to the destination tile. A particular case in the protocol between successor processes and the migrated process is represented by requests which are sent to the source tile just before the migration command arrives at the source tile. Each successor process checks if its requests have been served before the arrival of the migration command. If this is not the case, the successor tile has to send an interrupt-generating message to the source tile, in order to force the redirection of requests from the source tile to the destination tile.

4.6 Experiments and Results

In this section, we assess the benefits and overhead of the process migration mechanism proposed in this chapter. We perform our experiments on the same hardware platform setup used in Chapter 3, which is described in Section 3.5.3.

The setup of this experiment is shown in the left part of Figure 4.6. We use as a case study the M-JPEG application described in Section 3.5.2. $Tile_1$ initially runs all M-JPEG processes, which are listed in Figure 3.15, in a sequential way. P_1 is derived by merging *initVideoIn* and *videoIn* processes, P_2 and P_3 represent respectively the *DCT* and *Q* processes, and P_4 is obtained by merging the *VLE* and *videoOut* processes. We use the M-JPEG application as a case study because, compared to the Sobel application, M-JPEG processes are coarse-grained with high computation/communication ratio and therefore M-JPEG represents better the kind of applications which are likely to be mapped on a NoC-based MPSoC. The scheduling of the M-JPEG processes on $Tile_1$ before the migration is represented in Figure 4.5. Scheduling charts have been obtained using the GRASP [HvdHBL10] trace visualization tool to plot the information gathered at run-time. The trace shows the periodic scheduling which is obtained when all the processes are mapped on one tile and the scheduling policy is round-robin with yielding (yielding occurs when a process is blocked on reading or writing). The buffer size of each FIFO channel is set to two tokens in this experiment. In this scenario, the process scheduling iterates in the following way. First, P_1 executes two times, until it blocks on writing because its output buffer is full. Then P_2 is scheduled. It completes two iterations, consuming the tokens created by P_1 and producing two tokens for P_3. It then blocks while reading its input FIFO which is empty by then. Similarly, P_3 and P_4 execute twice before blocking on read. This scheduling repeats until the end of the application execution if no migration is performed.

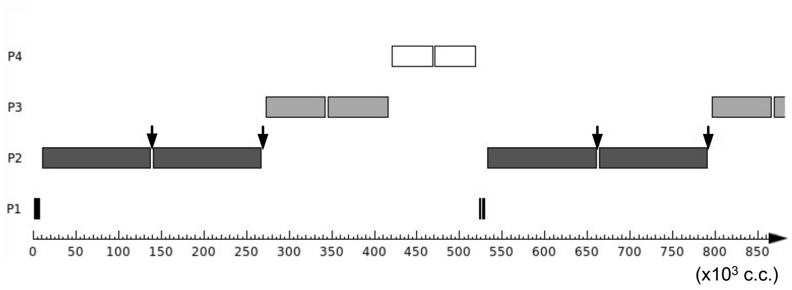


Figure 4.5: *M-JPEG process scheduling when running on a single tile. The scheduling policy is round-robin, with yielding when a process is blocked on reading or writing. The buffer size of each FIFO channel is two. These conditions lead to the periodic schedule (P_1, P_1, P_2, P_2, P_3, P_3, P_4, P_4), which continues indefinitely until the end of the application if no migration is performed.*

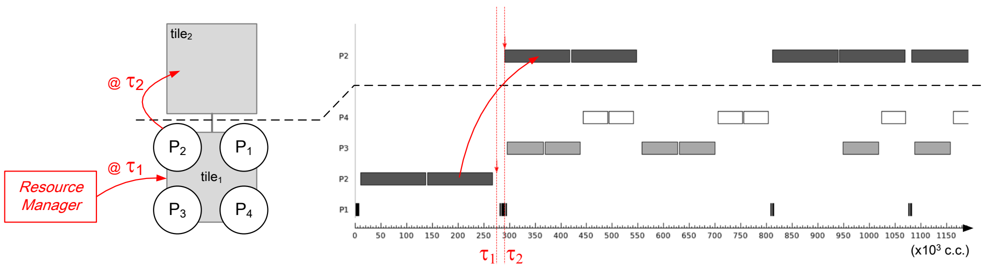


Figure 4.6: *M-JPEG process scheduling while migrating P_2 using the proposed migration mechanism. Until time τ_1, all processes are mapped on tile$_1$. At time τ_1, the resource manager requires process P_2 to migrate from tile$_1$ to tile$_2$. By using our interrupt-driven migration mechanism, the migration request is handled promptly. Process P_2 can be restarted on tile$_2$ within a predictable amount of time, in this case represented by the time interval ($\tau_2 - \tau_1$).*

4.6.1 Process migration benefits and overhead

System adaptivity requires the ability to change the process mapping at runtime in a predictable and efficient way. To illustrate the benefits of our migration approach presented in Section 4.5, we compare our proposed migration mechanism, driven by interrupt-generating control messages, with a migration approach based on fixed migration points.

In the latter case, process migration can take place only at fixed points in the code.

For instance, referring again to Figure 4.5, the arrows over the bars of process P_2 represent the start of an iteration of that process (for the sake of clarity, see line 4 in Figure 4.4). Assume that these points correspond to migration points, namely where the process checks if migration-messages have been sent by the resource manager. Given that the migration request can reach $Tile_1$ at any time, the latency of the actual process migration can vary. In the best case, the migration request reaches the tile right before the migration check point. In the worst case, the migration request arrives just after the migration check point, for instance the one which is reached around clock cycle 275,000 in Figure 4.5. The actual migration would not take place until the next migration check point of P_2, which happens to be after 2 executions of P_3, P_4 and P_1, and one execution of P_2. In this simple case, an upper bound of the process migration response time can be found, based on the process scheduling, which in turn depends on the workload of processes, the buffer sizes and the scheduling policy. In more complex cases, where the scheduling on one tile is affected by the scheduling on other tiles because of data dependencies, even finding an upper bound for the response time practically would not be possible.

By contrast, the interrupt-driven migration mechanism that we propose in Section 4.5 has a predictable behavior. As shown in Figure 4.6, the system has a faster response time to migration requests. At time τ_1, which is the worst case for the fixed point migration strategy discussed above, the resource manager sends a control message which triggers the migration of process P_2 to $Tile_2$. The process can be restarted on the destination tile within a predictable amount of time represented by the difference $(\tau_2 - \tau_1)$. This is the time it takes the source tile and the destination tile to execute the steps described in Section 4.5.2, such as the movement of the iteration vector of P_2 and the content of the FIFO connected to P_2, followed by the activation of P_2 on the destination tile. This migration overhead in time, $(\tau_2 - \tau_1)$, as shown in Figure 4.6, is much smaller than a single execution of the DCT function in process P_2. The migration procedure in this example actually takes less than 12% of a single execution of the DCT process.

Note that an upper bound of the migration procedure overhead can be derived for guaranteed throughput (GT) NoCs. In fact, the migration duration T_{mig} of a process $P \in \mathcal{P}$ can be split in two main components:

$$T_{mig}(P) = T_{stateMig}(stateSize(P)) + T_{procAct} \tag{4.3}$$

$T_{procAct}$ is a constant value which represents the time required to activate the migrated process using OS system calls, to update the middleware table, and complete all the actions described in Section 4.5.2 on the destination tile. $T_{stateMig}$ is the time it takes to transfer the state from the source to the destination tile. Its worst case, for GT NoCs, depends only on the state size. The largest state size occurs when both the input and output FIFO buffers connected to the migrating process P are full. This worst-case value can then be derived from the PPN topology and buffer sizes:

$$\max(stateSize(P)) = \sum_{ch \in IOC_P} size(B(ch)) \tag{4.4}$$

where $IOC_P = IC_P \cup OC_P$ as defined in Section 2.1.3, $size(B(ch))$ is the size of the buffer which represents the channel ch on the source tile. The value $size(B(ch))$ is obtained by multiplying the number of tokens of $B(ch)$ by the token size of a channel ch. An upper bound of the migration time T_{mig} of a process P can be calculated using $\max(stateSize(P))$ in Equation (4.3).

Our interrupt-driven migration mechanism incurs the worst-case overhead when a migration request arrives just before the end of a function execution in a process that has to be migrated. In this case, the migration still takes place in a predictable amount of time but the process execution has to roll back to the beginning of the interrupted iteration. In this scenario, all the time spent in the function execution is wasted.

4.7 Discussion

From the experimental results and analysis provided in Section 4.6.1 we can conclude that the migration mechanism proposed in this chapter complies with the requirements set in the problem statement of Section 4.1. In particular, our proposed migration mechanism possess the following properties:

- It is *predictable*, that is, when a migration is triggered, it will be completed within a certain time frame given by Equation (4.3);
- Thanks to the modified code structure of PPN processes proposed in Section 4.5.1, a migration can be triggered at any time during the process main body.
- Referring again to Section 4.5.1, the code needed to allow the proposed migration mechanism can be generated in a completely automated way.

Finally, note that the experimental results of Section 4.6.1 show that our proposed process migration mechanism is efficient. In fact, the overhead incurred to complete a process migration is experimentally shown to be negligible compared to the overall execution time of the application.