Study of task migration in a multi-tiled architecture. Automatic generation of an agent based solution

Ashraf Elantably

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nano Electronique et Nano Technologies (NENT)**

Arrêté ministériel : 7 août 2006

Présentée par

Ashraf ELANTABLY

Thèse dirigée par **Frédéric ROUSSEAU**

préparée au sein **Laboratoire TIMA**
et de **l'École Doctorale Electronique Electrotechnique Automatique et Traitement du signal EEATS**

Étude de la migration de tâches dans une architecture multi-tuile

Génération automatique d'une solution basée sur des agents

Thèse soutenue publiquement le **16 Décembre 2015**,
devant le jury composé de :

Monsieur, Jean François MEHAUT
Professeur - Université Grenoble Alpes, Président
Monsieur, Fabiano HESSEL
Maître de conférences - Université de Porto Alegre, Rapporteur
Monsieur, Gilles SASSATELLI
CR CNRS (HdR) LIRMM, Rapporteur
Monsieur, Florent DE DINECHIN
Professeur - INSA Lyon, Examinateur
Monsieur, Olivier GRUBER
Professeur - Université Grenoble Alpes, Examinateur
Monsieur, Frédéric ROUSSEAU
Professeur - Université Grenoble Alpes, Directeur de thèse

Acknowledgments

"Say: 'In the Bounty of Allâh, and in His Mercy; therein let them rejoice'. That is better than what they amass." **THE NOBLE QURAN**, Yunus 10:58.

I would like to sincerely thank my supervisor prof. Frédéric Rousseau for giving me the opportunity to work with him. I would like also to thank him for his guidance, friendliness, patience, kindness, and his continuous technical support during last four years and especially during writing this dissertation.

I would like to thank my thesis manuscript reviewers Dr. Fabiano Hessel and Dr. Gilles Sassatelli for their efforts, time taken to review this manuscript, and their constructive feedback and remarks. I would like also to thank prof. Jean-Francois Mehaut for presiding over my thesis defense committee, as well as the other members of the committee: prof. Olivier Gruber and prof. Florent de Dinechin for their questions and remarks.

I would like to thank my colleagues in SLS team members for their wonderful company especially Nicolas Fournel, Clément Deschamps and Etienne Ripert for their valuable contribution to the work of the project EURETILE. I would like to thank our partners in EURETILE project for their professionalism during the work in the project.

I would like to sincerely thank my family for their prayers, encouragement, and moral support during my studies. I would like as well to thank my wife for her support and my newly born baby for his delightful smiles and laughs that fill me with joy.

Ashraf Ahmed Mostafa El-Antably

January 2016

Abstrait

Les systèmes multiprocesseurs sur puce (MPSoC) mis en œuvre dans les architectures multi-tuiles fournissent des solutions prometteuses pour exécuter des applications sophistiquées et modernes. Une tuile contient au moins un processeur, une mémoire principale privée et des périphériques nécessaires associés à un dispositif chargé de la communication inter-tuile. Cependant, la fiabilité de ces systèmes est toujours un problème. Une réponse possible à ce problème est la migration de tâches. Le transfert de l'exécution d'une tâche d'une tuile à l'autre permet de garder une fiabilité acceptable de ces systèmes. Nous proposons dans ce travail une technique de migration de tâches basée sur des agents. Cette technique vise les applications de flot de données en cours d'exécution sur des architectures multi-tuiles. Une couche logicielle "middleware" est conçue pour supporter les agents de migration. Cette couche rend la solution transparente pour les programmeurs d'applications et facilite sa portabilité sur architectures multi-tuiles différentes. Afin que cette solution soit évolutive, une chaîne d'outils de génération automatique est conçue pour générer les agents de migration. Grâce à ces outils, ces informations sont extraites automatiquement des graphes de tâches et du placement optimisé sur les tuiles du système. L'algorithme de migration est aussi détaillé, en montrant les phases successives et les transferts d'information nécessaires. La chaîne d'outils est capable de générer du code pour les architectures ARM et x86. Cette technique de migration de tâche peut être déployée sur les systèmes d'exploitation qui ne supportent ni chargement dynamique ni unité de gestion mémoire MMU. Les résultats expérimentaux sur une plateforme x86 matérielle et une plateforme ARM de simulation montrent peu de surcoût en terme de mémoire et de performance, ce qui rend cette solution efficace.

Abstract

Fully distributed memory multi-processors (MPSoC) implemented in multi-tiled architectures are promising solutions to support modern sophisticated applications; however, reliability of such systems is always an issue. As a result, a system-level solution like task migration keeps its importance. Transferring the execution of a task from one tile to another helps keep acceptable reliability of such systems. A tile contains at least one processor, private main memory and associated peripherals with a communication device responsible for inter-tile communications. We propose in this work an agent based task migration technique that targets data-flow applications running on multi-tiled architectures. This technique uses a middleware layer that makes it transparent to application programmers and eases its portability over different multi-tiled architectures. In order for this solution to be scalable to systems with more tiles, an automatic generation tool-chain is designed to generate migration agents and provide them with necessary information enabling them to execute migration processes properly. Such information is extracted automatically from application(s) task graphs and mapping on the system tiles. We show how agents are placed with applications and how such necessary information is generated and linked with them. The tool-chain is capable of generating code for ARM and x86 architectures. This task migration technique can be deployed on small operating systems that support neither MMU nor dynamic loading for task code. We show that this technique is operational on x86 based real hardware platform as well as on an ARM based simulation platform. Experimental results show low overhead both in memory and performance. Performance overhead due to migration of a task in a typical small application where it has one predecessor and one successor is 18.25%.

Contents

Chapter 1

Résumé

Contents

Le but de ce résumé long en français est de synthétiser le travail qui est décrit en anglais dans les chapitres de ce mémoire. Ce résumé commence par une introduction qui explique le contexte de ce travail et le problème ciblé, ainsi que la solution proposée. L'état de l'art décrit des travaux qui concernent le même sujet. Nous détaillons ensuite notre solution et la méthodologique, puis nous décrivons les expériences qui sont réalisées pour valider notre solution. Nous finissons par une conclusion et quelques perspectives.

1.1 Introduction

Les systèmes embarqués électroniques existent dans presque tous les aspects de notre vie quotidienne. De nouveaux marchés sont ouverts presque tous les jours en raison de leurs capacités croissantes, de leur puissance de calcul et de leur ubiquité. Grâce aux progrès en cours dans le domaine de l'électronique, plus de transistors peuvent être intégrés sur une seule puce, et par conséquent, des systèmes plus performants peuvent être construits.

L'unité qui est responsable de réaliser toute la logique et les calculs arithmétiques s'appelle un microprocesseur. Un microprocesseur est un circuit intégré numérique qui est capable de lire les instructions stockées dans une mémoire et les exécuter sur les données stockées et/ou d'entrée(s) et enfin à stocker le(s) résultat(s) dans une mémoire. C'est un dispositif programmable. Les circuits intégrés (ou puces) sont composés de dispositifs semi-conducteurs (transistors). Un circuit intégré a besoin de nombreux transistors pour avoir plus de fonctionnalités et de capacités de traitement, et la technologie permet l'intégration de toujours plus de transistors.

Le rythme avec lequel le nombre de transistors intégrés a augmenté sur une seule puce, jusqu'à présent, avait été observé par Gordon Moore en 1965 [1]. Il a remarqué que le nombre de transistors double tous les deux ans. Ceci, conduit à la possibilité d'avoir des architectures de microprocesseurs qui peuvent fournir des puissances de calcul de plus en plus élevées. Par conséquence, le domaine des systèmes embarqués s'est enrichi. Un système embarqué est un système qui est capable d'effectuer une (des) fonction(s) dédiée(s) nécessitant une puissance de calcul limitée.

Dans la section suivante, nous donnons un aperçu des architectures de microprocesseurs, examinons brièvement les questions liées à la technologie, mentionnons la migration des tâches et les définitions de base et pourquoi la migration de tâches semble une bonne solution à la problématique de cette thèse.

1.2 Problématique

Pour bien comprendre certaines hypothèses et le raisonnement derrière les choix de conception, il est nécessaire de mentionner que ce travail a été réalisé dans le cadre d'un projet européen FP7 EURETILE. Ce travail est donc très lié à celui de certains partenaires, et des décisions communes ont été prises pour développer un démonstrateur.

Les calculs effectués par les unités de traitement des tuiles peuvent entraîner une variation thermique importante et affecter l'intégrité des composants. La migration de tâches pourrait être une solution à cette contrainte thermique. Mais d'autres raisons pourraient bénéficier du support de la migration de tâches : équilibrage de

charge entre tuiles, détection des problèmes de communication (congestion, rupture de liens) et adaptation en déplaçant des traitements sur une autre tuile, adaptation dynamique en fonction des applications pour une basse consommation (alimentation réduite de certaines tuiles).

1.2.1 L'environnement logiciel

Dans le cadre du projet EURETILE, c'est l'aspect fiabilité, notamment par la détection des problèmes de communication (un composant matériel permettait de détecter ces problèmes), qui nous intéressait. La détection d'un problème entraînait la migration d'un nombre réduit de tâches pour assurer la continuité de traitement pour de grosses applications de calcul.

Les principales phases de l'environnement logiciel du projet sont illustrées sur la figure 1.1. La première étape est le développement d'applications. Les applications sont développées en langage C et peuvent s'exécuter en parallèle. Chaque application est divisée en un certain nombre de tâches. Ces applications entrent dans un processus de co-conception dans laquelle les spécifications de niveau système sont analysées. Les applications sont simulées en couvrant plusieurs scénarios pour trouver la meilleure réalisation en fonction des objectifs de conception spécifiés. Dans cette phase, un modèle de programmation avancé est introduit. Il est appelé couche d'application distribuée (DAL). DAL permet la programmation des applications dynamiques à grande échelle, l'introduction d'un modèle à deux niveaux de réseau de processus, des mécanismes de contrôle, et une API complète. Tous ces modèles sont raffinés pour produire du logiciel, et notre participation au projet EURETILE s'est focalisée dans la couche logicielle de bas niveau (logiciel dépendant du matériel ou HdS). Ainsi, DAL présente les premiers niveaux des mécanismes de tolérance aux pannes qui nécessitent une prise en charge complète de la plateforme et du logiciel de bas niveau.

1.2.2 Hypothéses pour une solution de migration de tâches

Plusieurs hypothèses sont envisagées pour la mise en œuvre de la migration de tâches :

1. Les tâches qui migrent ne sont pas redémarrées à destination, mais reprises, ce qui implique le transfert d'état de la tâche. Il faut donc garantir la mise en pause d'une tâche sur la tuile source et sa reprise sur la tuile destination.

2. L'architecture du système doit être prise en considération. Les systèmes multi-tuiles n'ont généralement pas d'espace d'adressage accessible dans une mémoire partagée. Cela rend le transfert de l'état de la tâche effectué par communication explicite entre la tuile.

3. La migration de tâche doit être capable de fonctionner sur des systèmes d'exploitation légers, c'est à dire ne nécessitant pas des fonctionnalités spéci-

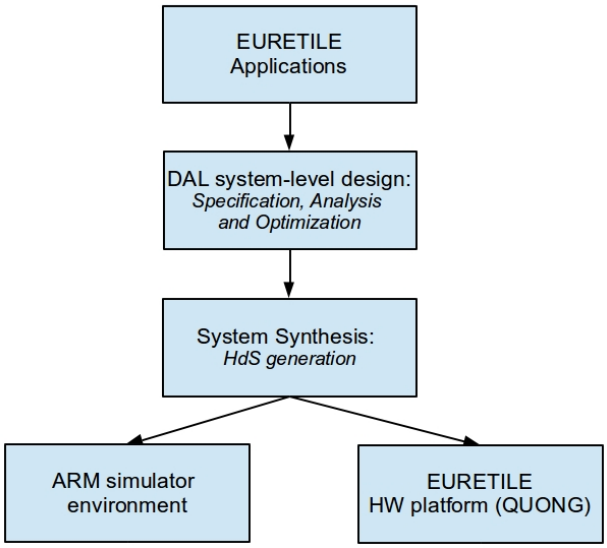

Figure 1.1: Vue globale de l'environnement logiciel du projet EURETILE.

fiques qui existent dans des systèmes d'exploitation plus sophistiques (MMU ou chargement dynamique).

4. La solution de migration devrait être compatible avec le flot de conception de DAL.

1.3 État de l'art

La migration de tâches a été mise en œuvre de manières différentes et dans des couches logicielles différentes. Les techniques d'implémentation varient selon des paramètres tels que l'architecture du système et les capacités du système d'exploitation. En plus, il y a plusieurs difficultés qui rendent l'implémentation difficile. Le chapitre 4 est dédié à la description des implémentations pour des architectures différentes et nous mentionnons les travaux de recherche correspondants. Cependant, dans cette section, nous nous focalisons sur les solutions qui concernent la migration de tâche dans des architectures multi-tuiles. Nous étudions ces solutions dans la section suivante en montrant comment elles se différencient de celle proposée dans ce travail.

1.3.1 Migration dans des architectures multi-tuiles

Les architectures des systèmes multi-tuiles sont des architectures NORMA, c'est à dire qu'un processeur (élément de traitement) d'une tuile ne peut pas accéder à la mémoire d'une autre tuile. Il existe des solutions récentes de migration de tâches qui sont proposées dans [24, 25, 26, 27, 29]. Elles visent des architectures multi-tuiles à base de NoC.

Dans [24], un système d'exploitation est utilisé sur lequel un chargeur dynamique est conçu pour soutenir le transfert du code de la tâche qui doit migrer. Toutefois, la migration du contexte de tâche (l'état de la tâche) n'est pas pris en charge. Par conséquence, cette technique est efficace seulement pour les tâches qui n'ont pas d'état ou si l'application ou la tâche redémarre sur la tuile destination. La migration de tâches est appliquée dans le contexte de la reconfiguration dynamique et pour une meilleure performance. Il est affirmé que le coût de la migration est amorti par le gain de performances à cause du redéploiement de la tâche. Il n'est pas mentionné, cependant, comment la solution est transparente vis à vis de l'application. Contrairement à cette solution, notre solution supporte les applications avec des tâches qui ont un état, c'est à dire un ensemble d'information qui permet la poursuite du traitement sur la tâche destination. De plus, notre solution fonctionne sur du matériel réel. [28] propose une solution assez similaire, mais sans transfert de code.

Dans [25], les auteurs déploient une technique de migration de tâches ciblant des MPSoCs basés sur une NoC qui ne comprend pas de transfert de code. En ce qui concerne la communication, un message MPSoC Passing Interface MMPI est utilisé. C'est un modèle de programmation parallèle qui rend le programme indépendant du placement optimisé de la tâche. La solution est basée sur les agents (tâches) maître (master) et esclaves (slaves) qui gèrent le processus de migration. Le maître réside dans un élément de traitement (Processing element PE) séparé et fonctionne sur le système d'exploitation Linux. Les esclaves sont exécutés sur MicroC/OS-II. Il est indiqué que cette technique fonctionne sur une plateforme de simulation.

Dans [26], ce travail se préoccupe principalement de fournir des plates-formes pour simuler la migration de tâches dans des architectures différentes. Par conséquent, le processus de migration n'est pas représenté du point de vue algorithmique ou méthodologique. Les plates-formes de simulation supportées sont des architectures à accès mémoire uniforme et des clusters (UMA et NORMA). Pour obtenir les résultats, des expériences de migration sont exécutés sur ces plates-formes. Ils utilisent une couche logicielle «middleware» qui fournit des APIs pour supporter la migration. Ils appliquent la réplication de tâches de sorte que toutes les tâches pouvant migrer, un réplicat est déployé sur toutes les tâches. L'impact pour répliquer toutes les tâches sur tous les éléments de traitement PE n'est pas représenté et limite l'évolutivité de la solution. La migration de communication est gérée par un composant qu'ils ont ajouté à la plateforme de simulation. Il s'agit d'un ordonnanceur runtime de ressource RTR «RunTime Resource Scheduler». Ce composant est responsable de la gestion de toutes les ressources matérielles et toutes les communications. Les résultats expérimentaux sont présentés d'une manière relative entre les différentes techniques de décision de déploiement. Des applications multimédias différentes sont exécutées. Les résultats montrent seulement l'impact du redéploiement en ce qui concerne, par exemple, l'amélioration de la communica-

tion sur le NoC. Ce qui peut expliquer pourquoi il n'y a pas beaucoup de différence entre certains cas où les décisions de déploiement peuvent conduire aux mêmes résultats.

Dans [27], les auteurs abordent le problème d'incohérence de la communication de la migration de tâches. Ils utilisent une solution basée sur des agents pour gérer la communication avant, pendant et après le processus de migration. Ils utilisent la méthode de commutation de protection «protection switching method» pour résoudre le problème d'incohérence, via la retransmission sans arrêter la communication. Contrairement à la solution proposée dans ce travail, aucun système d'exploitation n'est utilisé dans ce travail. Cela signifie que l'ensemble des logiciels de bas niveau est évidemment fait sur mesure comme notamment les pilotes de communication qui supportent des fonctions spécifiques. La cohérence de la communication est abordée ici, mais ce modèle ne semble pas pouvoir être générique pour s'adapter à différentes plateformes. Aucun détail n'est donné sur la transparence d'une telle solution vis à vis du programmeur.

Les travaux proposés dans [29] utilisent la migration de tâches pour améliorer l'adaptabilité du système MPSoC. La solution cible des applications décrites dans le modèle de calcul polyhédrique (Polyhedral Process Network PPN). Le modèle PPN est un cas particulier des réseaux de Khan KPN). Chaque tâche est basée sur une boucle avec deux compteurs. Les tâches ne peuvent pas avoir d'état, et par conséquent, la migration de tâches peut être vue comme une migration de communication, c'est à dire un processus de transfert qui concerne le contenus des files d'entrée et de sortie, et les deux compteurs de la boucle. La solution est basée sur une couche logicielle intermédiaire «middleware» qui fournit les APIs. La communication est initiée coté récepteur. Quand les données d'entrée sont reçues, elles sont stockées dans les files d'entrée correspondantes, et restent présentes même après leur consommation par la tâche. Elles ne sont supprimées que quand les données de sortie sont produites. Cette technique est utilisée car elle est simple à mettre en œuvre. L'interface réseau (NI) est étendue pour produire une interruption à la réception d'un message particulier, ce qui déclenche l'envoi d'un message par le gestionnaire d'exécution vers la tuile source de la migration et interrompt la tâche. La tuile source envoie ensuite aux tuiles voisines un message pour la migration. L'interface réseau nécessite donc d'être modifiée, contrairement à notre solution. Il n'est pas clairement indiqué dans l'article si une tâche peut avoir plusieurs canaux en entrée ou en sortie, et ce qui se passe quand les tâches voisines (prédécesseur ou successeur) sont sur la même tuile, ce qui suppose la mise en œuvre de communication inter-tuile. De plus, les opérations de lecture et écriture n'étant pas atomiques, il n'est pas précisé comment on garantit de ne pas perdre de données quand une interruption se produit pendant une lecture.

À notre connaissance, il n'y a pas de travail équivalent au notre dans la littérature avec la mise en œuvre d'une solution complète, évolutive et qui génère

automatiquement les ajouts (couche logicielle) pour une exécution sur une plateforme multi-tuiles existantes. En effet, presque toutes les implémentations et leurs validations sont effectuées soit sur des plateformes de simulation ou des plateformes ad-hoc basées sur des FPGA. Les mesures sont aussi données de manière relative et sont rarement représentées de manière absolue. Cela rend les surcoûts pas très clairs et une analyse comparative très difficile. Des questions telles que la transparence pour les programmeurs d'applications et la portabilité sur différentes architectures de base sont rarement expliquées, ce qui pose la question sur la façon dont la solution est applicable à des applications existantes ou comment adapter la solution de migration à différentes architectures. Enfin, il n'y a aucune technique qui étudie l'influence de performance de migration en fonction du nombre de voisins dans une architecture distribuée. Notre solution est expliquée avec des détails à la fois méthodologiques et algorithmiques. Notre méthode est déployée dans un outil de génération automatique, ce qui rend possible son utilisation pour différentes architectures et pour différentes applications.

1.4 Méthodologie de la solution

L'objectif de cette section est d'expliquer comment les problèmes de migration de tâches sont gérés. Les détails qui sont données dans cette section répondent à la question : *Comment le processus de la migration de tâches a lieu ?*
Les solutions sont expliquées selon les deux perspectives méthodologiques et algorithmiques.

Ce travail décrit le processus de migration de la décision de migration de la tâche jusqu'à sa reprise sur la tuile destination en se focalisant aussi sur l'aspect communication pour ne pas perdre de données. Les algorithmes utilisés pour prendre la décision de migrer une tâche ne font pas partie de ce travail. Dans la spécification initiale, on laisse le concepteur système indiquer les tâches critiques, qui pourront subir une migration, ainsi que les tuiles qui seront en charge d'exécuter la solution de déploiement de tâche initiale et la tuile destination qui accueillera la tâche après migration.

Les architectures distribuées sont intrinsèquement décentralisées. La décentralisation est attribuée au fait que chaque tuile exécute sa propre copie du système d'exploitation. C'est pourquoi, la solution de migration de tâches est basée sur des agents répartis sur les tuiles de manière à décider et exécuter la migration. Afin de permettre à ces agents d'exercer leurs rôles dans la migration, une couche logicielle intermédiaire «middleware» (appelée Multi-Processing FrameWork MProcFW) est développée pour fournir des APIs nécessaires qui facilitent l'exécution de la migration. Ces API sont invoquées par les agents, et font le lien entre le système d'exploitation et la couche logicielle applicative. Par conséquent, cette caractéristique offre une bonne portabilité sur différentes architectures et une transparence

vis à vis des applications. La solution réside dans deux couches logicielles : la couche application où les agents résident et la couche «middleware» où MProcFW fournit les API nécessaires.

1.4.1 MProcFW

La couche MProcFW est conçue pour fournir des API nécessaires pas seulement pour faciliter la migration, mais aussi pour permettre le contrôle des tâches des applications. MProcFW se situe au-dessus de la couche de système d'exploitation dans l'architecture logicielle (figure 1.2). Les APIs de MProcFW peuvent être classés en deux catégories comme suit :

- Un catégorie qui contient toutes les APIs qui contrôlent les tâches, c'est à dire création, démarrage, pause, arrêt ou migration. Ces APIs sont applelées dans la couche d'application par des tâches spécifiques (ou agents).
- Un catégorie qui contient toutes les APIs qui contrôlent les canaux de communication en créant, l'ouverture, l'envoi, la réception et la fermeture des canaux. Ces APIs sont appelées par la couche d'application.

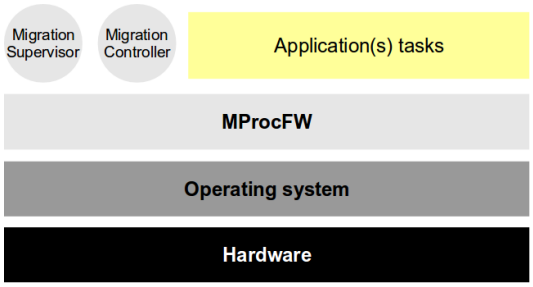

Figure 1.2: L'architecture logicielle du système. La couche MprocFW réside directement au-dessus le système d'exploitation.

1.4.2 Transparence des points de migration

Notre technique de migration de tâches est basée sur des points de migration [9, 10, 32, 33]. Un point de migration est un point prédéfini dans le code applicatif où une requête de migration est traitée. La tâche concernée doit être arrêtée en toute sécurité et son exécution reprise à partir de ce point de migration. MProcFW spécifie un modèle de tâches en boucle, et toutes les tâches doivent être conformes à ce modèle.

Le modèle de tâche est présenté dans le code 1.1. La procédure `INIT` est principalement responsable de l'allocation de mémoire pour l'état de la tâche et de l'initialisation des variables nécessaires. L'initialisation est exécutée une seule fois

au démarrage de chaque tâche de l'application. L'exécution principale est composée d'itérations de la procédure `FIRE`. La procédure `FIRE` est composée de trois parties pour la lecture des données (jetons) d'entrée, le traitement et l'écriture des données en sortie. A la fin de l'application, la procédure FINISH est appelée pour nettoyer la mémoire allouée par la procédure `INIT`. La communication via les canaux utilise les procédures `read` et `write`.

```
procedure INIT(TaskStructure *t)  // initialization
  initialize();
end procedure
procedure FIRE(TaskStructure *t)  // execution
  Ch_FIFO->read(buf, size);     // read i/p from fifo
  process();                      // processing data
  Ch_FIFO->write(buf, size);    // write o/p to fifo
end procedure
procedure FINISH(TaskStructure *t) // cleanup
  cleanup();
end procedure
```

Listing 1.1: Le modèle d'une tâche de flux de données (Data-flow task model). `TaskStructure` est une structure de données qui contient tous les paramètres d'une tâche.

1.4.3 Gestion du problème d'incohérence de la communication

L'incohérence de communication provient du fait que les tâches qui migrent changent de tuile d'exécution. Cela rend la reprise de la communication entre la tâche qui vient de migrer et ses voisins impossible sans modification des canaux qui les relient. En conséquence, ces canaux doivent être changés pour que la reprise de l'exécution se fasse de façon correcte. Les tâches voisines doivent évidement être informées de la migration. De plus, l'incohérence de communication peut aussi être attribuée à la possibilité d'avoir des données non-traitées qui résident dans les files d'entrée de la tâche qui migre et qui ne sont pas encore consommées. La reprise d'exécution doit se faire en conservant ces données. Afin d'effectuer une migration avec succès, il faut transférer toutes les données qui restent dans les files d'entrées à la tuile source à la tuile de destination, mais, ce transfert doit être fait de manière transparente vis à vis des développeurs d'applications et si possible sans utiliser des services ou des dispositifs spécifiques de communication.

Nous proposons deux solutions pour résoudre les deux problèmes mentionnés précédemment. La première concerne la modification des canaux d'une manière transparente. Pour cela, nous proposons un nouveau type de canal appelé «canal reconfigurable». Un canal configurable relie une tâche qui peut migrer et une tâche voisine. Un tel canal possède trois extrémités dépendant du sens du canal : un canal d'entrée de la tâche à migrer possède une seule extrémité pour recevoir des données

et deux extrémités pour lire les données. Un canal de sortie de la tâche à migrer possède deux extrémités d'entrée et une extrémité de sortie. Un canal configurable est composé de deux branches, chaque branche étant un canal ordinaire mais un seul canal est activé à la fois. Ceci facilite la connexion entre une tâche qui peut migrer et son réplicat d'un côté, et un voisin d'un autre. Lorsque il y a une migration, tous les canaux configurables sont mis à jour par la commutation entre leurs branches. Ce processus de mise à jour est fait de manière transparente pour les tâches qui ne détectent rien. En effet, le numéro de port (d'extrémité) reste fixe mais l'identifiant du canal change. Ce changement est effectué par l'invocation de l'API spécifique (`Mproc_channel_update`) pour faire la commutation entre les deux branches. La figure 1.3 illustre ce principe avec un canal d'entrée de la tache qui peut migrer et un canal de sortie (figure 1.3.a). On peut observer les deux branches de chaque canal reconfigurable. La figure 1.3.b généralise ce concept par montrer les branches de deux canaux configurables dans le cas où les deux tâches reliées peuvent migrer.

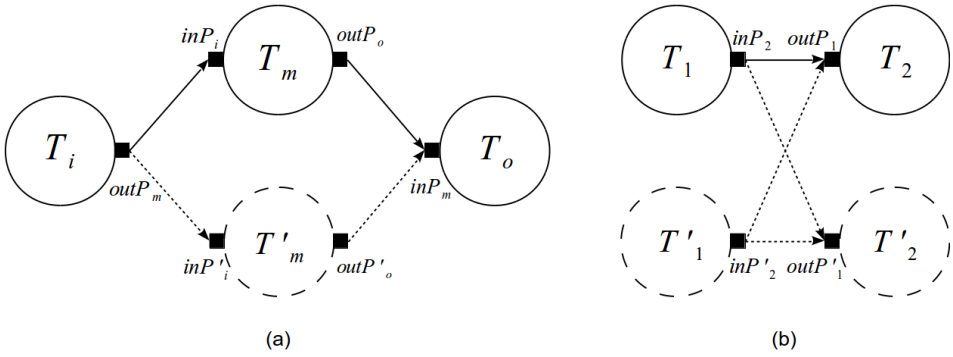

Figure 1.3: Canaux configurables. T_m est une tâche qui peut migrer, T_i et T_o sont respectivement les prédécesseur et successeur. T'_m est le réplicat. (a) Seul T_m peut migrer. (b) T_1 et T_2 peuvent migrer.

La deuxième solution cible le problème des données non traitées dans une FIFO liée à la tâche qui peut migrer. Le but est de proposer une méthode qui permet le transfert des données après migration des données non consommées au début de la migration. Cette solution doit être transparente pour le programmeur c'est à dire sans devoir modifier le code de l'application. Nous ajoutons alors un protocole pour la communication entre les tâches de l'application : chaque tâche qui transmet les données à une autre, sauvegarde un copie des données transmises dans une file qui s'appelle une file de copie (copy buffer CB). Côté réception, toutes les données reçues par une tâche sont stockées dans une file qui s'appelle une file de réception (receive buffer RB) (figure 1.4). Dès qu'une tâche consomme une donnée de la file de réception, elle transmet au coté émetteur l'information de consommation, ce qui permet supprimer la donnée correspondante dans CB. Ce protocole permet d'avoir toujours une copie de toutes les données non-traitées, par conséquence, lorsque la migration est effectuée, les copies sauvegardées coté émetteur peuvent

être retransmises à la tuile destination pour synchroniser les nouvelles files de réception. Cette synchronisation est effectuées par l'invocation de l'API de MprocFW (`MProc_channel_synchronize` décrite dans l'annexe A).

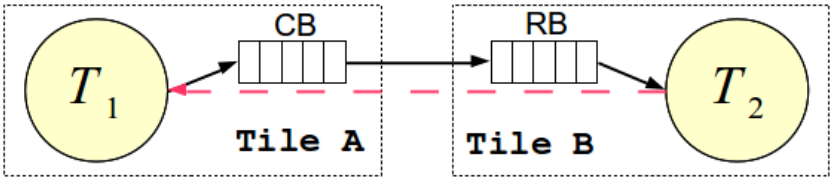

Figure 1.4: Protocole "Write-with-copy"

1.4.4 Génération des agents de migration

Le nombre et le rôle des agents de migration sont déterminés en fonction de l'approche choisie : entièrement distribuée, entièrement centralisée, semi-distribuée.

Entièrement distribuée : tous les agents ont le même rôle qui inclut à la fois la décision de migration et son exécution. Cette approche correspond bien à la nature de l'architecture dans le projet EURETILE. Cependant, cette approche nécessite une conception complexe des agents de migration. Un autre inconvénient de cette approche est que la décision de migration peut être prise par un certain nombre de différents agents dans le même temps, qui peut conduire à de l'instabilité ou un blocage si deux tâches adjacentes dans le même réseau de processus migrent en même temps. En effet, l'incohérence des données et des canaux peut entraîner un blocage avant la reprise.

Entièrement centralisée : Il existe un agent pour l'ensemble du système qui est responsable à la fois de prendre la décision de migration et son exécution. Cela évite la complexité de l'approche entièrement distribuée, car le développement d'un agent centralisé est relativement facile. Mais cela peut aussi entraîner un nombre important de communications et des difficultés de synchronisation.

Semi-distribué : deux types d'agents sont conçus avec des responsabilités différentes. Un agent de migration est centralisé et est chargé de prendre la décision de migration et un autre agent est responsable de l'exécution des commandes de migration. La complexité du développement de cette approche est intermédiaire entre les deux approches précédentes. Cette solution correspond bien à la nature décentralisée de l'architecture du projet EURETILE par groupe de tuiles, constituant un cluster. La complexité de la conception des agents est relativement atténuée, mais au détriment de la communication nécessaire afin de coordonner tous les agents nécessaire à la migration.

La troisième approche est adoptée en raison de ses avantages par rapport aux deux autres. Elle fournit une solution générique qui correspond à une architecture multi-tuile distribuée. Avec une telle hiérarchie distribuée, les agents ont différentes responsabilités. Un des agents (le maître) gère un groupe d'agents esclaves, ce qui rend la migration a deux niveaux : la prise de décision est différent du niveau d'exécution de la décision. Par conséquence, il existe deux types d'agents selon le rôle de chacun d'eux :

i **MigSup**, C'est l'agent superviseur de la migration qui est chargé de prendre les décisions de migration. Il initie la migration via l'envoi de commandes de la tuile source à une tuile destination.

ii **MigCtrl**, C'est l'agent contrôleur de la migration qui est responsable de l'exécution des commandes de migration envoyées par MigSup, en transférant état de la tâche de sa tuile source vers la tuile destination, en prenant en charge les communications avec les voisins, et en assurant la reprise d'exécution sur la tuile destination. MigCtrl est capable d'exécuter des commandes de migration sur la tuile en raison de sa capacité à accéder à la mémoire principale privée.

Le système est divisé en groupes de tuiles (clusters). La migration ne peut se dérouler qu'à l'intérieur d'un même cluster. Il existe un seul agent MigSup par cluster tandis qu'un MigCtrl existe sur chaque tuile. Les API fournies par MProcFW ne sont suffisantes pour la gestion de la migration, il faut aussi que les agents disposent des informations sur le placement des tâches sur les tuiles. Cette information est maintenant disponible sous la forme de tables, ce qui permet aux agents de connaître les tâches voisines et leur placement (déploiement) ainsi que les destinations de migrations pour les taches concernées. Deux tables sont nécessaires :

1. "Global View Table" **GVT** contient des informations sur toutes les tâches qui peuvent migrer dans un cluster. GVT est composée d'un certain nombre d'enregistrements correspondants au nombre de tuiles du cluster. Dans un enregistrement, on retrouve la liste des tâches qui peuvent migrer, leur tuile source et leur tuile destination. GVT est utilisée par MigSup, par conséquence, une seule GVT existe par cluster.

2. "Destination look-up table" **DLT** contient plus de détails sur les tâches qui peuvent migrer tels que les emplacements de tous les voisins, et tous les identifiants de canaux reliés à cette même tâche. Ce tableau permet à MigCtrl d'avoir toutes les informations pour gérer la suspension et la reprise correcte de l'exécution après migration.

1.4.5 Principe de la migration

L'algorithme de migration est conçu en deux parties distinctes :

i La partie qui concerne la prise de la décision de migration.

ii La partie qui concerne l'exécution de la migration.

La première partie, la prise de décision est faite par l'agent MigSup. Les critères de décision, la méthode de décision, les valeurs ou paramètres des informations permettant cette prise de décision ne font pas partie du travail de thèse. Néanmoins à titre d'exemple, dans le contexte du projet EURETILE, un composant spécifique permettait de récupérer des informations sur l'état du matériel de l'architecture multi-tuile (température des unités de traitement, état des liens - opérationnel , défaillant - entre les routeurs du réseau sur puce connectant les tuiles au sein d'un cluster), ce qui permettait de décider ou non d'initier une migration. La responsabilité de l'exécution de la migration est donnée à l'agent MigCtrl. MigSup initie la migration en informant les agents MigCtrl des tuiles concernées, ce qui permet de démarrer le processus de migration.

1.5 Expérimentations et résultats

La solution proposée dans ce travail a été implémentée, c'est à dire que le flot de production de code initial a été modifié pour ajouter les agents, et la couche d'API MprocFW. Le code ainsi obtenu peut alors être compilé pour obtenir un ensemble de fichiers binaires à exécuter sur les différentes tuiles de l'architecture. Deux types d'architectures distinctes sont visées : le domaine de l'embarqué avec des cœurs de calcul ARM v7 dans une plateforme de simulation, et le domaine du calcul haute performance avec une plateforme existante composée de 16 tuiles, chaque tuile intégrant un cœur x86 (Xéon). L'exécution d'une application mettant en œuvre la migration sur ces deux plateformes permet alors d'une part de valider la solution, mais aussi de mesurer les coûts et les performances de la solution pour une analyse de la méthode. Par exemple, nous étudierons l'impact sur la mémoire, sur le temps d'exécution en cas de migration, mais aussi l'influence du nombre de canaux sur les performances.

L'application de démonstration est une petite application, avec trois tâches. En fait, cette application est dupliquée pour obtenir un ensemble d'applications à exécuter. L'intérêt d'une petite application est d'une part sa facilité de mise en œuvre, d'autre part, les résultats obtenus permettront de valider la méthode sur une application défavorable. Le temps d'exécution ou la taille mémoire de l'application sont petits, et le surcoût mémoire ou la perte de performance est plus significatif.

Les résultats les plus difficiles à obtenir sont ceux provenant des expérimentations faites sur l'architecture multi-tuile matérielle. En effet, l'accès (distant) à la machine a été limité à quelques semaines dans le cadre du projet EURETILE, mais aussi aucune solution de deboggage n'était disponible, car c'est le chargement des binaires autour de DNA-OS n'offre pas cette possibilité. Les mises au point ont toujours été faite dans un premier temps à l'aide de la plateforme de simulation ARM

pour la vérification de la fonctionnalité (application et algorithme de migration). La mise au point pour la plateforme matérielle a été plus délicate.

1.5.1 Description des plateformes

Si les deux plateformes matérielles diffèrent, le système d'exploitation utilisé pour toutes les expérimentations est le même. Il s'agit de DNA-OS [14], un système d'exploitation développé au TIMA et porté sur les deux architectures (en 32 bits). DNA-OS est léger, et son mécanisme de génération n'inclut que les fonctionnalités nécessaires à l'application. Il supporte les architectures SMP, possède un ordonnanceur coopératif, et la communication est basée sur un système de fichiers virtuel. La `libc` est largement supportée. Néanmoins, il ne possède ni support MMU, ni chargement dynamique.

1.5.1.1 Domaine de l'embarqué : plateforme de simulation ARM

La plateforme de simulation a été développée au TIMA en SystemC TLM [45]. Elle est composée de tous les composants ou périphériques nécessaires tels que `TTY`, afficheurs.... Chaque cœur ARM cortex A9 utilise un émulateur QEMU. Chaque tuile contient une mémoire privée, et un composant spécifique (composant de communication) permettant l'interface avec le NoC. Ce composant spécifique est proche du composant physique utilisé dans la seconde plateforme. Néanmoins, il a nécessité une adaptation du pilote. L'instanciation de plusieurs tuiles permet de créer un cluster à l'aide de scripts. Pour un temps de simulation raisonnable, on est aujourd'hui limité à 128 tuiles. Chaque tuile exécute sa propre copie du système d'exploitation.

1.5.1.2 Domaine du calcul haute performance : plateforme matérielle x86

La plateforme matérielle, développée par l'INFN Rome (Italie) - coordinateur du projet EURETILE, est basée sur les processeurs Intel (Xéon SMP) et fournit un environnement pour du calcul scientifique haute performance [46]. Plusieurs mécanismes spécifiques et performants permettent les communications entre tuiles (infiniband, APEnet+). La plateforme est composée de 16 tuiles (figure 1.5). Tous les composants à l'intérieur d'une tuile sont connectés via un bus PCIe. Une machine serveur est installée pour faciliter les connexions à distance et le chargement des binaires sur les tuiles.

1.5.2 Surcoût de performance

Le surcoût de performance à cause de la migration est le temps additionnel sur le temps d'exécution d'une demande en raison de la migration. Le temps d'exécution mesuré est le temps total écoulé depuis de départ de l'application jusqu'à la fin, y

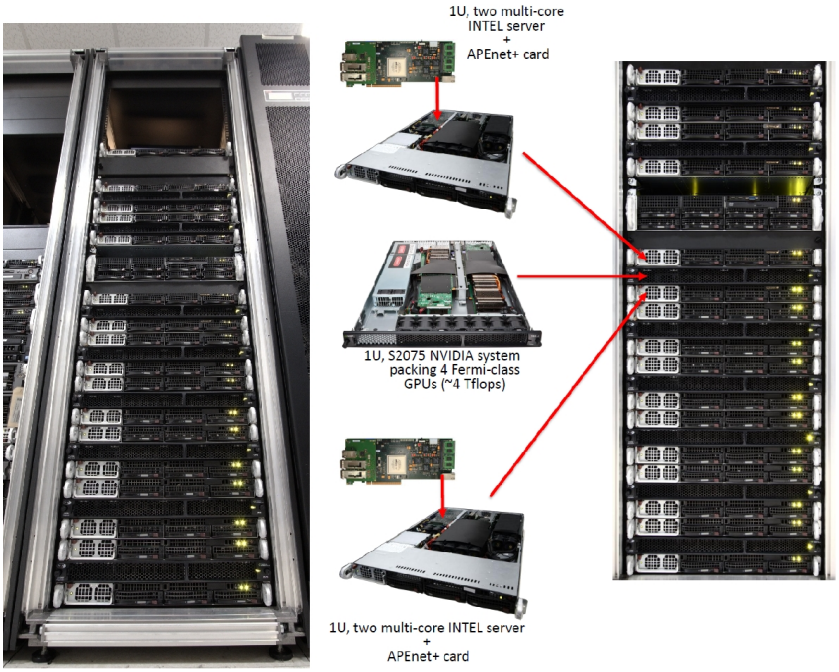

Figure 1.5: Plateforme matérielle 16 tuiles pour le calcul haute performance.

compris la communication inter-tuile. De part notre solution basée sur des agents, il y a deux composantes à ce surcoût :

a Le temps d'exécution normal des agents de migration.

b Le temps écoulé à cause de la migration.

Pour évaluer le surcoût de migration, nous avons effectuer la mesure dans trois cas différents pour une même application :

1. Cas 1 (C1) : Aucun agent de migration existe. Le temps d'exécution de l'application est mesurée.

2. Cas 2 (C2) : Les agents de migration existent et s'exécutent mais aucune migration n'a lieu. Ce cas est évaluer pour les mesurer le surcoût introduit par les agents de migration par rapport à cas 1.

3. Cas 3 (C3) : Les agents de migration s'exécutent et une migration est effectuée à un moment arbitraire. Ce cas est conçu pour les mesurer le surcoût introduit par les deux agents de migration et le processus de migration par rapport à des cas 1 et 2.

Quatre tuiles sont utilisés sur lesquelles deux applications identiques s'exécutent, chaque application contient trois tâches. Les tâches sont : la tâche du générateur,

Table 1.1: Les temps d'exécutions et le surcoût à cause de migration.

Cas	temps de départ (μs)	temps de fin (μs)	temp d'exécution (μs)	Surcoût (%)
Application 1				
1	65 027 461	67 247 538	2 220 077	N/A
2	28 401 948	30 623 281	2 221 332	**0.06**
3	21 754 757	24 379 930	2 625 173	**18.25**
Application 2				
1	33 163 108	35 387 392	2 224 284	N/A
2	28 687 143	30 911 856	2 224 713	**0.02**
3	40 811 884	43 358 217	2 546 333	**14.49**

la tâche de traitement et la tâche du consommateur. Toutes les tâches ont des états. Le placement des agents et de l'application est représenté dans la figure 1.6. Les résultats sont donnés dans tableau 1.1.

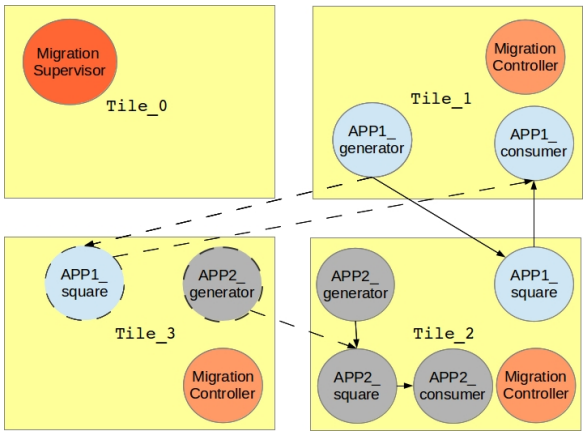

Figure 1.6: Le placement des agents et les applications sur les quatre tuiles.

On remarque que les agents de migration ajoutent un surcoût négligeable car l'augmentation du temps d'exécution pour les deux applications est de l'ordre de 0,02% - 0,06%. Sachant que dans les situations typiques, une migration n'est pas un événement ordinaire qui a lieu régulièrement, il n'y a presque aucun coût de déploiement des agents dans les applications.

Le surcoût du à la migration est d'environ 18% du temps d'exécution par rapport aux cas 1 de l'application 1, mais seulement de 14% pour l'application 2. La différence vient de plusieurs facteurs : la tâche qui peut migrer n'est pas la même, et le nombres de canaux connectés est différent. Pour l'application 1, `APP1_square`

a deux voisins s'exécutant sur des tuiles différentes, ce qui augmente le temps notamment pour la gestion des canaux, alors que l'application 2 où `APP1_generator` est la tâche qui migre a un seul voisin et un seul canal (de sortie).

1.5.3 Le surcoût de la migration par rapport à la taille de l'état de la tâche

L'objectif de cette série d'expériences est d'étudier le surcoût causé par la taille de l'état de la tâche. L'impact est mesuré en faisant varier la taille de l'état (taille d'un tableau). Les expériences sont effectuées sur la même plateforme 4-tuiles avec la même application et le même placement donné dans la figure 1.6. La taille de l'état de `APP1_square` est augmentée et le temps d'exécution incluant une migration est mesuré. La charge utile "payload" de données dans chaque paquet du réseau est limitée à 4 ko, c'est à dire que les tailles de l'état d'une tâche qui sont inférieures ou égalent à 4 ko sont envoyés dans un seul paquet. Les paquets de taille supérieure à 4 ko sont divisés en plusieurs paquets de 4 ko. Le tableau 1.2 donne les différentes tailles et les surcoûts de migration correspondants. La relation entre le surcoût et la taille est montrée dans la figure 1.7. On observe un loi d'évolution linéaire.

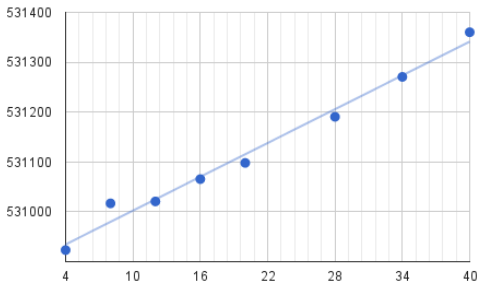

Figure 1.7: Le surcoût de la migration (μs) vs. la taille de l'état de la tâche (ko).

Table 1.2: Taille de l'état de la tâche vs. le surcoût de migration.

Taille (ko)	Surcoût (μs)	Pourcentage d'augmentation (%)
4	530 923	-
8	531 017	0.0176
12	531 021	0.0184
16	531 066	0.0268
20	531 099	0.033
28	531 191	0.0504
34	531 271	0.0654
40	531 360	0.0823

1.5.4 Variation du surcoût d'exécution de la migration

L'objectif de cette série d'expériences est d'étudier si le temps écoulé par la migration varie quand elle est répétée dans les mêmes conditions. Ceci permet d'évaluer le déterminisme de l'algorithme de la migration. Avec cette information, nous pourrons prédire le temps de migration et offrir une certaine qualité de service.

Table 1.3: Variation du surcoût de migration en modifiant la date de migration.

Expérience	Temps de départ (μs)	Temps de fin (μs)	Temps d'exécution (μs)	Surcoût Cas 1 (μs)
1	27 801 647	30 425 029	2 623 382	402 057
2	45 614 455	48 237 843	2 623 387	402 062
3	78 243 189	80 866 640	2 623 450	402 125
4	36 361 808	38 987 310	2 625 503	404 177
5	121 649 077	124 273 322	2 624 246	402 920
6	24 258 600	26 884 037	2 625 437	404 111
7	56 529 629	59 154 948	2 625 319	403 994
8	69 221 222	71 844 467	2 623 245	401 920
9	31 294 232	33 919 547	2 625 315	403 990
10	88 779 483	91 402 913	2 623 430	402 105
Temps moyen (μs)				402 946
Temps maximum (μs)				404 177
Temps minimum (μs)				401 920
Écart-type (μs)				1 004
Différence (Max - Min) (μs)				2 258
Pourcentage de (Max-Min) à temps moyen				0.56%

Plusieurs expériences de migration sont répétées, avec chaque fois la même tâche qui migre, mais à une date arbitraire. A chaque expérience, tout le système est réinitialisé, la machine doit redémarrer pour éviter l'influence de la mémoire cache. La taille de l'état de la tâche est fixée dans toutes les expériences et ne dépasse pas 4 ko. Les résultats sont donnés dans le tableau 1.3. La gamme de variation des surcoûts mesurés (différence entre maximum et minimum) est de 2.25 ms, comme illustré dans la figure 1.8. Les chiffres montrent que la variation du surcoût due à la migration est limitée (de l'ordre de 0,6%) avec un écart type très faible (environ 1 ms). Cela contribue à une bonne estimation du surcoût de la migration et surtout rend les coûts de migration déterministes.

1.5.5 Le surcoût de migration par rapport au nombre de canaux

La méthode de migration proposée dans notre travail, est basée sur des agents déployés sur une architecture distribuée, ce qui implique un échange de commandes

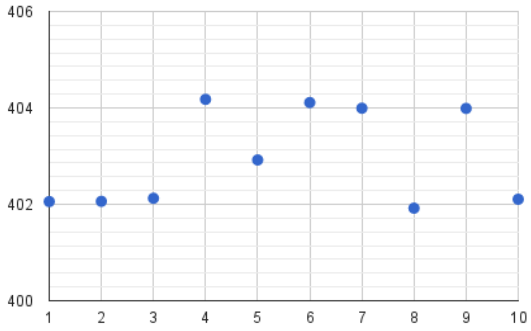

Figure 1.8: Variation du surcot de migration.

entre les différents agents et les tuiles impliquées dans la migration. Pour cette raison, le temps de migration dépend du nombre de tuiles voisines impliquées, notamment dans les communications avec la tâche qui doit migrer. L'objectif des expériences décrites dans cette sous-section est d'étudier l'influence du nombre de canaux sur le surcoût de migration. Le principe de l'application de calcul du carré est reprise mais en modifiant le nombre de générateurs pour faire varier le nombre de canaux.

Le graphe de tâches de l'application 1 a été modifié pour produire les deux exemples suivants :

i Quatre générateurs comme indiqué (Figure 1.9.(a)). Ils sont répartis sur `tile_1` et `tile_2`.

ii Dix générateurs (Figure 1.9.(b)). Ils sont répartis sur les tuiles `tile_1`, `tile_2` et `tile_3`.

Les temps d'exécution sont mesurés dans le cas 2 (avec agents sans migration) et 3 (avec agents et migration). Les résultats sont présentés dans le tableau 1.4.

Table 1.4: Surcoûts de la migration dans les cas de quatre générateurs et dix générateurs.

Cas	Temps de départ (μs)	Temps de fin (μs)	Temps d'exécution (μs)	Surcoût Cas 2 (%)
Quatre générateurs				
2	17 482 080	19 565 805	2 083 724	N/A
3	24 940 435	27 628 979	2 688 544	**29**
Dix générateurs				
2	30 300 077	32 434 774	2 134 697	N/A
3	33 822 369	37 231 950	3 409 580	**59.7**

Nous pouvons aussi étudier la relation entre le nombre de canaux et les surcoûts

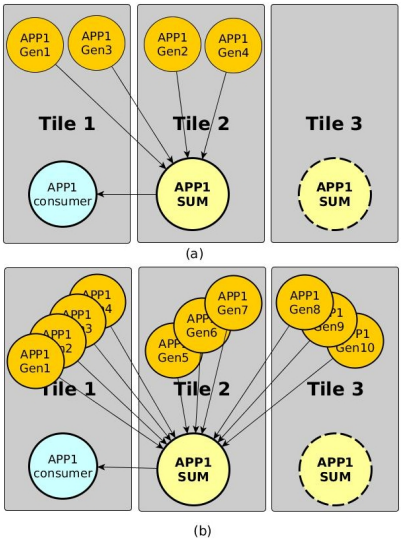

Figure 1.9: Placement optimisé de l'application modifiée sur un plateforme de trois tuiles. (a) Quatre générateurs. (b) Dix générateurs.

de performance due à la migration par la consolidation de toutes les mesures de toutes les expériences présentées dans les sections précédentes. Nous avons listé dans le tableau 1.5 toutes les expériences avec le nombre de canaux et de leurs surcoûts de migration correspondants. La relation est linéaire entre le nombre de canaux et le surcoût de performance comme présenté dans la figure 1.10.

Table 1.5: Nombre de canaux par rapport les surcoût de performance.

Expérience	Nombre de canaux	Surcot cause de migration (%)
(App2) in table 1.1	1	14.5
(App1) in table 1.1	2	18.3
In table 1.4	4	29
In table 1.4	10	59.7

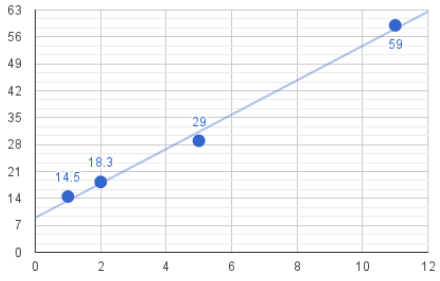

Figure 1.10: Le surcoût de la migration (%) vs. nombre de canaux.

1.6 Conclusion

Nous avons proposé une solution de migration de tâche basée sur des agents. Cette solution cible les architectures multi-tuiles, et présente un faible impact en terme de surcout de performance. La solution est expliquée à la fois de point de vue méthodologique et algorithmique avec les détails de mise en œuvre. Un flot de génération a été développé dans le cadre du projet européen FP7 EURETILE, ce qui a permis de nombreuses expérimentation à la fois sur une architecture ARM en simulation, mais aussi sur une architecture x86 existante en exécution.

La solution proposée présente plusieurs avantages. Elle est transparente pour le programmeur d'applications. La solution est basée sur une couche logicielle intermédiaire conforme à POSIX. Elle est donc portable sur d'autres architectures et d'autres systèmes d'exploitation. Aucun service spécifique d'un système d'exploitation n'est nécessaire (pas d'utilisation de la mémoire virtuelle ou de chargement dynamique), et même un système d'exploitation léger supporte la méthode. Dans ce contexte de système d'exploitation léger, on a montré que le cot était déterministe.

Chapter 2

Introduction

Contents

As computers are getting more and more pervasive, there is hardly an aspect in our lives nowadays in which they do not exist. New markets are being opened to computers almost everyday due to their increasing capabilities, computation power and ubiquitousness. Their versatile functionalities are ranging from simple management of automatic garden watering to sophisticated robots that discover distant planets' surfaces. Thanks to ongoing and outstanding advancements in electronics, more transistors can be integrated on a single chip, hence, better computers can be built.

The unit that is responsible for undergoing all the logic and arithmetic computations and incorporates all functions of Central Processing Unit CPU inside a computer is called a microprocessor. A microprocessor is a digital integrated circuit that is capable of reading instructions stored in a memory and executing them on stored and/or input data and finally storing the outcome on a memory, i.e. it is a re-programmable device. Integrated circuits (or chips) are made of semiconductor devices (called transistors). The more features and capabilities an integrated circuit has, the more transistor it needs. As a result, technology has been being more and more developed and advanced to house bigger numbers of transistors.

The pace with which the number of transistors is being increased on a single chip until now had been observed by Gordon Moore in 1965 [1]. He observed that the number of transistors doubles every almost two years, as depicted in Fig 2.1. This, consequently, leads to the possibility of better and more fledged microprocessor architectures that can deliver higher and higher computation powers. This, also

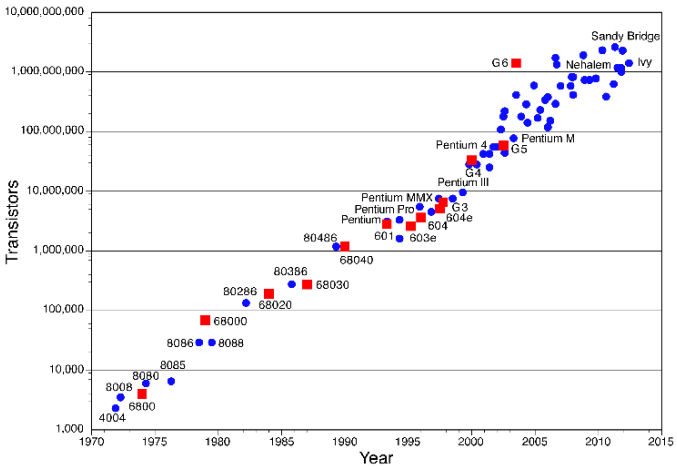

Figure 2.1: Moore's law, number of transistors doubles every two years. The source of the photo is in [2]

leads to computers invading wider and wider aspects.

Computers, in time, not only do not need much space to exist but also they are becoming more and more portable enriching the embedded systems field. An embedded system is a system that is capable of performing dedicated function(s) for specific purpose(s) with limited supply of power. These systems are classified according to the manner with which they deliver their outputs; some must deliver the output of their processing in predefined hard timed-deadlines as in *hard real-time* systems, others are not constrained stringently by time, i.e. their outputs can be accepted within a relatively acceptable delay from their deadlines like in *soft real-time* systems.

In the following section, we give an overview of microprocessors architectures, look briefly at technology-related issues, mention task migration basic definition and how it proposes a good system-level solution for some of these issues and finally mention how this work is planned.

2.1 Microprocessor architectures

A microprocessor is the unit that is responsible for performing computations and processing digital data. This unit incorporates the functions of a Central Processing Unit CPU. It is built on a single chip that sometimes houses other peripherals beside the CPU rendering it to be micro-controller.

2.1.1 What is architecture?

Architecture is an assembly of disciplines that describe the design and implementation of a microprocessor. It covers three aspects of microprocessor design:

i *Instruction-set*, it is the set of instructions visible to the programmer (compiler). They are the basic instructions that only can be executed by the real hardware. They stand in the middle between software and hardware, i.e. whatever the type of the software; it must be entirely converted to these instructions.

ii *Organization*, it includes high-level aspects of the design like the memory system, the memory interconnect, and the design of the internal processor (like arithmetic, logic, branching, and data transfer are implemented).

iii *Hardware design*, it refers to the specifics of the microprocessors including logic design, clock rates, transistor and packaging technology.

2.1.2 Towards parallel architectures

Microprocessor architectures have different impacts on the computer systems. They affect performance, power consumption and how it can be programmed. Although architectures keep getting more sophisticated, this cannot continue indefinitely. This is due to the fact that higher attained speeds of microprocessors can no longer increase the throughput of the whole computer systems that require not only CPUs but also memories, networks and power. Because of the fact that developing all the components (processor, memory, interconnect ... etc.) in a computer system cannot result in the same speedup, latency of some components can dominate the speedup of another. As a result, ongoing speed enhancements in processors are blocked by the latencies of other components like memory. This makes further architecture development in uniprocessor systems infeasible due to the diminishing returns. This can be briefly explained as follows:

- *Hitting memory wall*: speed of memory access is not increasing with the same rate as that of microprocessor. Consequently, ongoing increase in microprocessor speed requires larger caches. Larger caches solution is not considered to be free, i.e. its cost diminishes the benefit of the bigger size.

- *Wire delay*: wire delay does not scale well with transistor performance, i.e. while transistors are getting smaller and more efficient, wire delays are becoming dominant. This makes wire delay a limitation in designing computers.

- *Hitting power wall*: since transistors are becoming more and more numerous because they are smaller, power needs to be brought to and distributed among the chip gates. Heat results from power distribution anomalies in a filled area with transistors. That is why distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges. Unlike in the past when area made the most important limitation in designing integrated circuits in general, power now makes the major limitation to using transistors.

That is why the trend now is to have multiple microprocessors (or cores) in single chip (or multiple processor system-on-chip MPSoC) and they are all there for executing different parallelized applications where an application (it is, sometimes, called *job*) is split into a number of processes and a process has a number of threads. As a result parallel architectures emerged where every processor executes a thread so that a number of threads are executed in a simultaneous manner. All microprocessor architectures are categorized according to the taxonomy that was put by Michael J. Flynn [3] in 1966. They are categorized such that an architecture must be one of the following four ones:

1. *Single Instruction Single Data* SISD: This is the category in which a uniprocessor is used to execute instructions one by one.

2. *Single Instruction Multiple Data* SIMD: Same instruction is executed by a number of a processors (or co-processors) all managed by a single control unit such that every processor has its own data memory; however, only one instruction memory is accessible by all of them. All processors access multiple items of data in parallel. That is why SIMD computers exploit *data-level parallelism*. Vector architecture is the largest class of SIMD architectures.

3. *Multiple Instruction Single Data* MISD: A number of processors are supposed to perform different operation on the same data. Not many instances of this architecture exist, as MIMD and SIMD are often more appropriate for common data parallel techniques, hence, no commercial multi-processor system of this kind has ever been built.

4. *Multiple Instruction Multiple Data* MIMD: Each processor has its own instruction and data memories; therefore, each one fetches its own instructions and operates on its own data. As a result MIMD computers exploit *thread-level parallelism*, since multiple threads operate in parallel. In general, thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable.

We classify MIMD multiprocessors according to their memory access times to two categories as follows:

I *Uniform Memory Access UMA architectures*
The time the processors need to access a central the main memory is the same. This is due to the fact that the main memory is central and shared among all processors. This architecture is called also Shared memory architecture or Symmetric multiprocessing SMP or centralized memory architecture, they are expanded in chapter 3, section 3.1.1.

II *Non Uniform Memory Access NUMA architectures*
Every processor needs different time to access the main memory. The main memory does not exist in a central entity; however, it is distributed among all processors with different general accessible portions and private portions.

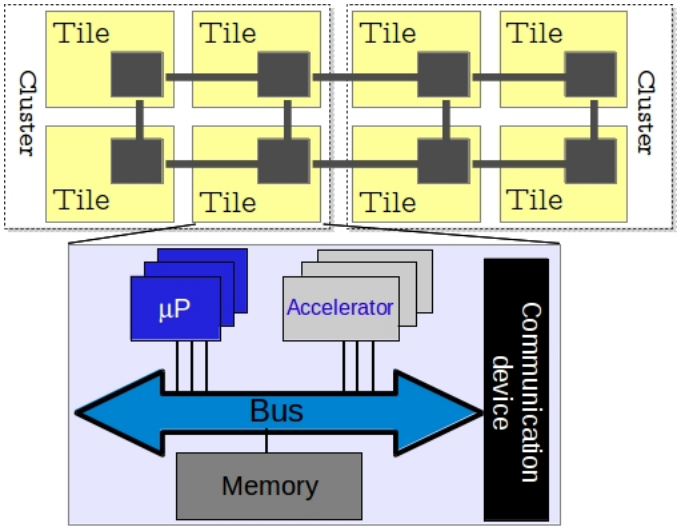

Figure 2.2: Multi-tiled architecture.

They are also called distributed memory architecture, they are expanded in chapter 3, section 3.1.2.

2.1.3 Multi-tiled architecture

Although centralized UMA architectures are able to house parallel applications and show valid and satisfying parallelism levels, they do not show high degrees of scalability. They cannot be getting bigger without solving the problem of having the central shared memory as a bottleneck. That is why, distributed architectures are being examined to be further applied in embedded systems as they have been being used since considerable period of time in cluster computing. This is because they have much potential to be scalable and support parallel applications.

Multi-tiled architecture combines a core or more with necessary peripherals and a communication device in a modular element called a tile as depicted in Fig 2.2. It is a distributed memory architecture where it is not allowed for a core in a tile to access memory in another tile, hence, No Remote Memory Access NORMA architecture. A multi-tiled architecture consists of a number of tiles connected by a communication medium. A communication medium with high throughput and scalability is required for such architectures. That is why, communication is usually performed via Network-on-Chip NoC which is better compared to previous communication architectures. MPSoCs with multi-tiled architectures are more capable to accommodate parallel data-flow applications.

In this work, we focus on MPSoCs where there are still issues in their design and operation. Reliability is one of those issues which still accompany deep sub-micron technologies and makes a lot of concerns. Other issues come from the nature of

applications themselves; they tend to exhibit time changing workloads which make mapping decisions sub-optimal in number of scenarios. This is due to the large variety of use cases which is, of course, directly proportional to the complexity of the required applications and features. Offline approaches of tasks mapping paradigms are no longer sufficient to cope with such dynamic change in workloads. This, in fact, imposes a challenge of task re-mapping or re-distribution. This re-distribution is usually demanded in a dynamic fashion i.e. task relocation/migration between the cores has to be supported.

2.2 Task migration

A *process* is a key concept in operating systems [4]. It consists of data, a stack, register contents, and the state specific to the underlying Operating System **OS**, such as parameters related to process, memory, and file management as depicted in Fig 2.3.(a). A process can also be referred to as a task and it can have one thread or more. Thread is also called light-weight process and has its own stack and register contents; however, it shares the process address space and some of the OS signals as illustrated in Fig 2.3.(b). Every process, in short, has its own *context*. The context of a task or a thread can be defined as the minimal set of data used by this task/thread 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 (after returning from interrupt service routine **ISR** or after migration process is completed).

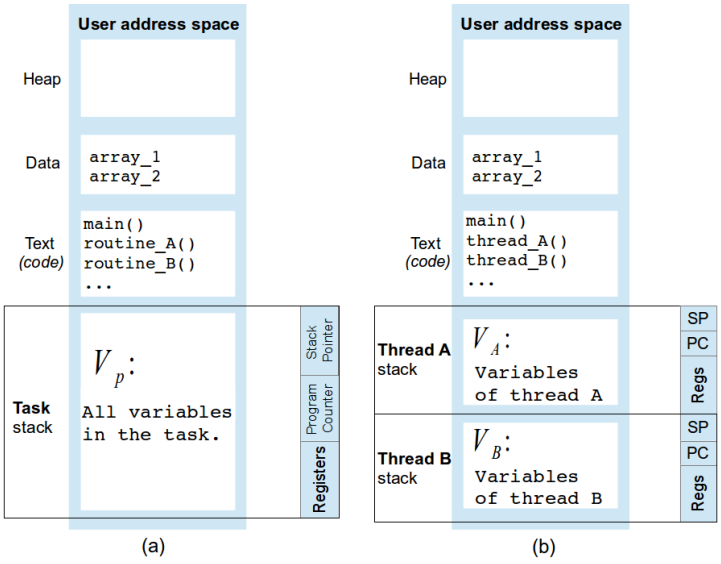

Figure 2.3: Multi-thread process model

The main idea of task (*process*) migration is the transfer of a process state plus its address space from the source core (referred to also as home or host node) to the

destination one (referred to also as destination node). Its significant cost is coming mainly from the process of transferring the address space. The address space is usually composed of the task stack and heap. However, this is not the only task attribute to be transferred. The whole task state including the address space and the CPU registers, opened files and ports, has to be transferred to the destination core to be eventually resumed/restarted properly.

The size of that address space varies from task/thread to another according to several parameters, local variables size is one of them, for instance. Consequently, the cost of the task migration becomes variant and hence the latency of the migration. The size of the address space is not only depending on the variables size but also on the instant the task has to be migrated at which in turn makes the size of variables and even their number depend on the execution path of the thread.

2.2.1 Motivation

There are different motivations that constitute the rationale behind adopting task migration solution. The importance of such solution stems from the fact that it can function either pro-actively or reactively to systems errors; it can prevent the existence of thermal spots before their formation or depending on the architecture, it can transfer tasks away from faulty links via other alternative ones, have they existed by design in the architecture. In the following, we list different motivations of using task migration:

2.2.1.1 Load Balancing

Substituting the workload among all cores evenly contribute to reducing the peak temperature of the system. If the cores become hotter, this leads to increased leakage power, transient errors that eventually lead to permanent physical damages in the system.

2.2.1.2 Reliability

Increasing the reliability leads to increasing the durability of the whole system. Measures must be taken according to the overall system condition detection like runtime error checking, and on-chip communication fault statistics. These measures are simply summarized in avoiding excessive use of certain cores. In order to respond to such situations, tasks may need to get transferred between cores.

2.2.1.3 Power consumption

Task migration is a valid solution that is being adopted to reduce power consumption by coupling dynamic voltage and frequency scaling DVFS as in [5]. In heterogeneous architectures, sometimes it is better to move away tasks even from not so

hot cores to other ones just to avoid wasting much power by entering their power down mode.

2.2.1.4 System adaptivity

As mentioned earlier, in more and more complex applications not all possible scenarios can be handled statically, therefore, task remapping is sometimes inevitable to avoid links contention, use tiles with hardware accelerators or cores with powerful co-processors according to the system overall state. Task remapping demands task migration facility supported.

2.2.1.5 Variability

As electronics technology gets more and more evolved and scaling is going deeper and deeper, variability issue appears. In deep sub-micron (<22nm) integrated circuits of the same functionality on the same die show variable performances. This variability is on the same die and pretty considerable to the extent that it can be compared to die-to-die variability. Run-time adaptivity as shown before is quite beneficial to reduce the problem of decreased predictability of exact functional, electronic and thermal behavior of a system in pre-silicon phase [6]

2.3 Outline

This work is planned as follows: chapter 3 shows a necessary background before it explains the problem statement of this work. Then chapter 4 comes to mention related work in literature showing how they are different from this work, as well as, showing main contributions. Chapter 5 explains the solution thoroughly from both methodological and algorithmic points of view, it shows the solution design in details and its practical application. Chapter 6 expands how the proposed task migration capability is plugged into a working tool-chain in order to automatically generate codes for multi-tiled systems. Then chapter 6 comes to show experimental results after explaining the real hardware platform as well as the simulation platform used.

Chapter 3

Background and problem statement

Contents

Basic background is always necessary before explaining task migration technique in details. This chapter explains task migration techniques in different architectures, expands the general approaches of task migration and shows the nature of parallel applications and the adopted programming model. In addition to the background this chapter ends with the explanation of the problem statement, as well. Details about the nature of the programming model and applications used are given.

3.1 Hardware/Software architecture

Task migration came with the appearance of multi-processing. Its research had started long time ago regarding workstations and servers. However, now it has recently regained high importance in the field of Multi-Processor System on Chip **MPSoC**. The reason behind this is the fact that implementation of task migration

is not an easily achievable goal, this is because the running OS are not usually full-fledged like how they are on workstations and servers. Another important reason is the constraints entailed by different types of the architectures of multi-processing systems.

There are different architectures of multi-processors according to memory organization and interconnect strategy. This is because the number of cores (similar processors) is varying according to the memory organization and interconnect. There are two groups [7], centralized shared-memory architecture and distributed shared memory **DSM**. The main motivation behind this difference in the architecture is the number of cores versus the memory bandwidth. It is already well known that the memory in case of centralized shared memory architecture as shown in Fig 3.1, is the bottleneck, for that the memory bandwidth has to be enlarged to enable more accesses simultaneously. On the other side, the DSM architecture integrates higher number of cores. This integration, in turn, makes the memory partitioned to be local to every core for speeding the computation with the ability to have a shared memory for communications purpose. Sometimes the shared memory does not exist and no core can access the private memory of another, this variant of that kind of architecture is called No Remote Memory Access **NORMA**.

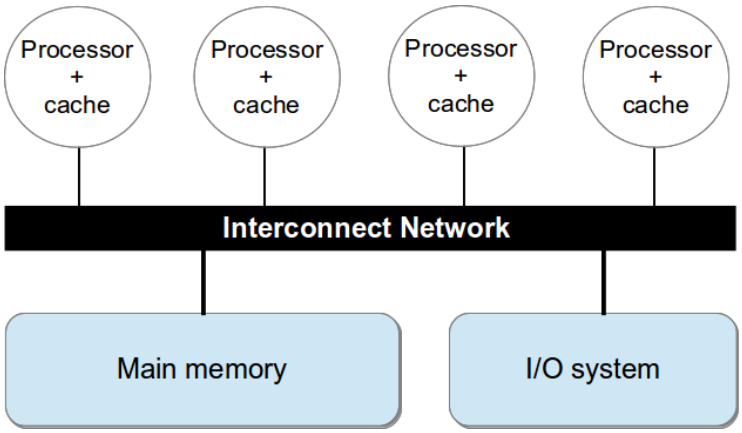

Figure 3.1: Centralized shared memory architecture

3.1.1 Centralized shared memory architecture

The main memory is shared among all the processors as shown in fig 3.1. The cost of accessing the memory by all the processors is the same, for that this architecture can be called also uniform memory access **UMA**. A common communication medium links memory modules and cores (computational modules) having caches. The communication between the cores is commonly undergone through memory modules via reads and writes to the shared data stored in the shared memory. This architecture is also called tightly coupled shared memory systems (also called sym-

metric multi-processor **SMP**). In which all processors run the same copy of the OS which also resides in the main memory. Synchronization takes place by cache-coherent low latency shared-memory. Also the address spaces for the processes all reside in the main memory. The context transfer is not a complex process since only the registers values have to be transferred from the source core to the destination one to resume the task execution. In fact, it shall be a lot easier if the task is to be restarted in the destination core.

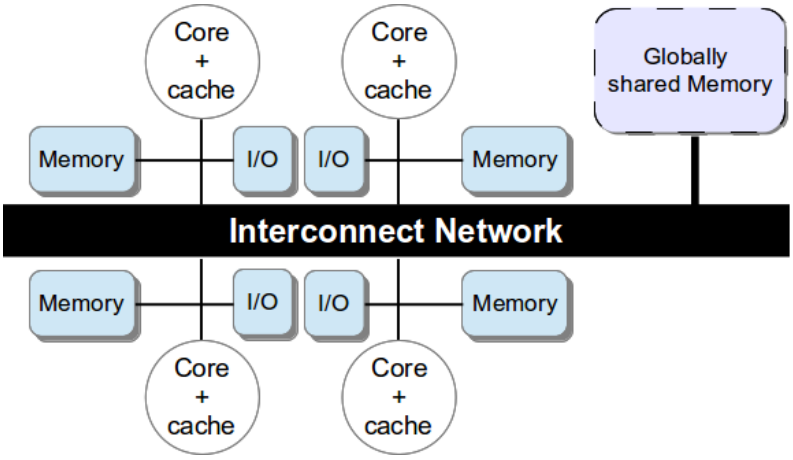

Figure 3.2: Distributed memory architecture, the shared memory is margined in dotted line to show that it is optional. In case of its absence the architecture is called NORMA

3.1.2 Distributed memory architecture

As obvious as its name, it is the architecture in which every core has got its own main memory attached to it with the optional existence of cache. The memory is distributed among all cores as illustrated in Fig 3.2. Sometimes called non-uniform memory access **NUMA**. Access of all the memories could be granted to all cores, the memory access cost is no more equal among the cores. That's why it is called non-uniform memory access. Every core runs its own copy of the OS. Communication in such architecture can take place by two methods; The first is like that previously mentioned in the case of SMP which is by using distributed shared memories. The second method of communication occurs through a shared address space, the physically separate memories can all be accessed as one logically shared address space. This means that the address is translated by the core to a physical memory location address to access the correct memory module. In short the data communication is done via load and store instructions.

If the address spaces are completely private to the cores which means that every core is attached to its own local memory and no granted access to any other core

and the optional shared memory is absent (NORMA case), the shared memory method is no more useful, this is illustrated in Fig 3.2 but without the optional dotted shared memory. Which leads us to the second method of communication which has been being used in cluster computing **CC**, it is called message-passing interface **MPI**. In which data (in our case data would be the task state in addition to task address space) is transferred by messages between cores, this message-passing multiprocessor is built on the fact that the processors share nothing.

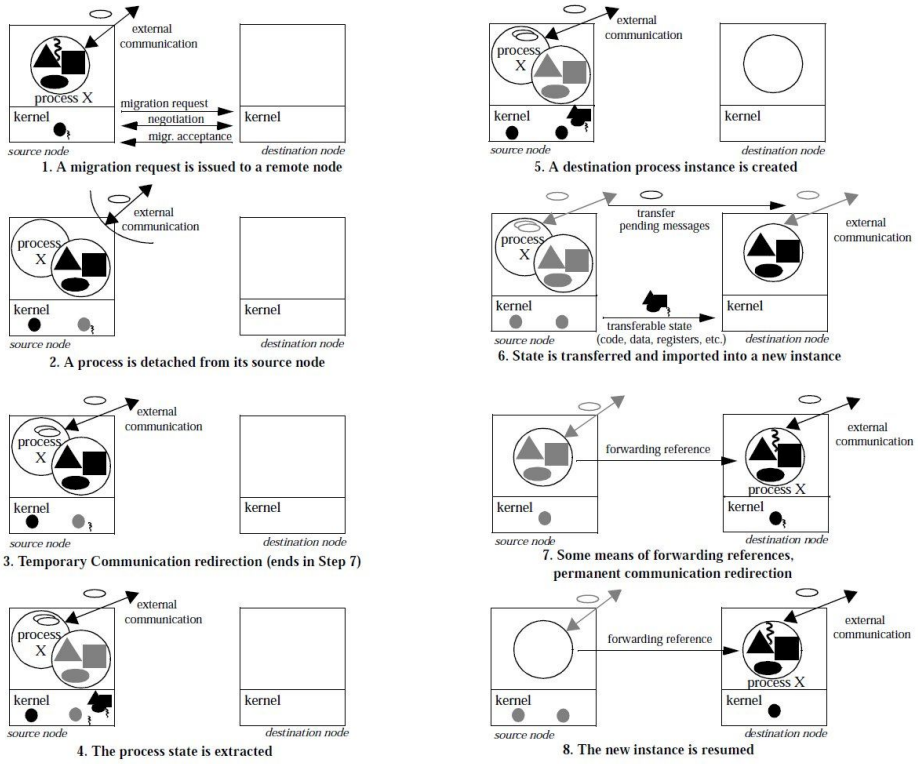

Figure 3.3: Migration algorithm

3.2 General migration algorithm

There are a lot of implementations and designs. However, the main steps of task migration all can be summarized in the following algorithm made in [3]. The algorithm means all the steps included in the process of migration in addition to the protocol-like communication between the source core and the destination one regarding the transfer of the task attributes. For the sake of further clarification, all the steps are depicted in Fig 3.3 and listed as follows:

(A) *Migration request*
The migration request is sent by the source node to the destination node after

negotiation had been taking place between them. This negotiation can differ from implementation to another according to the requirements.

(B) *Process detachment*
At the source node, the process execution is suspended declaring it to be in migrating state as a preliminary step for redirecting its communication as shown in next step.

(C) *Temporary communication redirection*
All messages arriving to the suspended migrating task shall be queued and sent after migration ends. This step remains being executed in the background in parallel with next steps 4, 5 and 6. After the end of step 7 the migration will be known to all nodes.

(D) *Task state extraction*
Task state includes:

1. Processor state: register contents
2. Communication state: message channels and opened files
3. kernel state

(E) *Destination task instance creation*
A task (process) shall be created in the destination but deactivated and in idle mode waiting for enough state to come until resumption.

(F) *Task state transfer*
Transfer and importing task state in the newly created destination idle one as a preliminary step to start resumption.

(G) *Forwarding references*
In this step all the queued messages in the source (if any) shall be all forwarded to the destination. This shall be done in addition to modifying the channels to be with the destination node instead of the source one.

(H) *New instance resumption*
When sufficient state has been transferred and imported. With this step, process migration completes. Once all of the state has been transferred from the original instance, it may be deleted on the source node.

3.3 Address space transfer strategies

One of the main contributors in the communication cost in task migration is the transfer of the address space of the task from source node to the destination. This, in turn, impacts directly the performance. That is due to the fact that address space size is variable not only among the tasks but also regarding one task. That's because not only the size changes according to the execution path of the task itself

and even in the same path but in time course. This means that the address space size depends on several parameters which in brief, are the task itself, execution path and the instant of execution. This makes a difficulty in predicting task migration performance or in other words, the time of migration. This issue, in fact, had been tackled extensively before in the domain of servers so it is worth mentioning the strategies devised for task address space transfer. This is, of course, due to the importance of opting for the most optimum one to be adopted in the embedded domain which intrinsically imposes time constraints and suffers from limited resources. The strategies, expanded in [8], are as follows:

(A) Eager (all)
It is a simple strategy which is based on copying all the contents of address space and simply sending the whole address space. This strategy has a considerable high initial cost due to the amount of memory but no run-time cost. Initial cost is high obviously because of the time elapsed in transferring all the task address space and no run-time cost is also attained because the transferred task will not suffer delays and can be resumed as soon as it is copied in the memory. This strategy is implemented by Checkpoint[1] /Restart.

(B) Eager (dirty)
This strategy can be deployed if there is remote paging[2] support. This is a variant of the eager (all) strategy that transfers only modified (dirty) pages because they will be very likely the latest accessed and used part of the address space according to temporal locality. Unmodified pages are paged in on request from a backing store. They shall be lazily copied back on-demand. This variant provides significantly better initial cost especially with tasks of large address spaces but higher run-time costs. It also does not help much in case the task migration is needed because the source core is getting faulty because it does depend on the remaining of unmodified stored in the source node and retrieved back on-demand.

(C) Copy-on-Reference (CoR)
This strategy also needs remote paging. It transfers pages on demand. It is very much the same in performance compared to the previous; however, with lower initial cost but much higher run-time one.

(D) Pre-Copy
This strategy needs remote paging support, as well. It is designed to reduce

[1] *Checkpointing* means that there are predefined task positions for migration, in other words tasks cannot just be migrated at any instant of execution. Sometimes whole task states at these predefined points are stored for prominent migrations.

[2] *Memory paging* is originally adopted in servers and workstations as it requires secondary storage media (hard disk). However, it is actually called demand paging in embedded domain and it only applies to code execution. It is the act of dynamically loading code pages from NAND flash memory into main memory then onto instruction cache if existed. Demand paging in embedded is also considered like prefetching.

the "freeze" time of the migrating task. This can be achieved by letting the task to continue execution while sending all its address space to the destination core. Sending will take place so long as the number of dirty pages is smaller than a fixed limit. Pages dirtied during pre-copy have to be re-sent after migration.

3.4 Migration approaches

There are different layers [8] in which migration can be implemented, user application layer or kernel layer are examples of these layers. The layer of implementation does; however, impact some parameters like complexity, performance, transparency and re-usability. Migration implemented in *user application* does provide the application programmer a degree of freedom to derive better policies since migration knowledge is provided. However, the one implemented in *kernel* space on contrary does provide better performance and transparency. There are, of course, disadvantages for each. The first option lacks performance while the second suffers from relatively high complexity; the different layers are viewed in Fig 3.4. In embedded systems, it has been preferred to implement task migration facility in *middleware* like in [9, 10]. A scheme of different SW abstraction layers is depicted in Fig 3.4. That is mainly because one of the crucial constraints in embedded is the performance. That is in addition to reducing as much as possible development time to cope with diminishing time-to-market which mandates high re-usability.

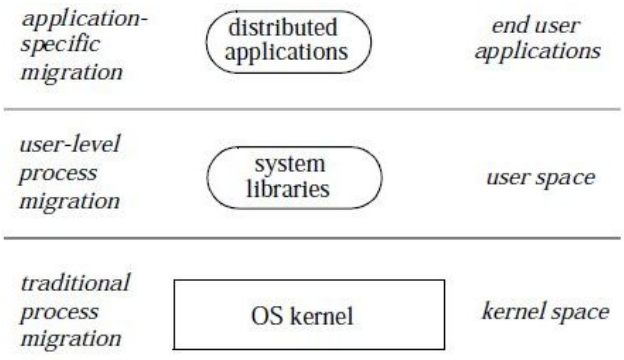

Figure 3.4: Migration levels differ in implementation complexity, performance, transparency and re-usability.

Task migration can also be classified to two different methodologies according to the way of implementation. They are task replication and task recreation. These two methods are going to be explained in the next two subsections and then an overview of different methodologies in implementing migration solutions taking into consideration the architecture of the platform for each case.

3.4.1 Task replication

It is considered the easiest way of task migration implementation. Its idea is about having replicas of all tasks (that can be migrated from or to in the future) in the system on all processors. There shall be a stack allocated in the main memory of every processor while kernel-level task-related information are allocated by each OS in the process control block **PCB** (i.e. an array of pointers). A task shall be active and running on only one processor while reside in the suspended tasks queue in all other processors. When a migration is needed, the task is suspended in its source processor and resumed in the destination processor by bringing it from the suspended queue to the ready one. The rationale behind adopting such alternative is that transferring the whole task code and stack will imply allocating new memory space for its stack in the main memory which in turn requires dynamic loading which is not available in the majority of embedded operating systems.

3.4.2 Task recreation

This mechanism kills the process on its source processor and recreates it on the destination processor. This mechanism requires the OS to be supporting dynamic loading. Also if the destination processor does not have memory management unit **MMU** which preforms address translations between physical and virtual memory, a position independent code **PIC**[3] will be needed as well.

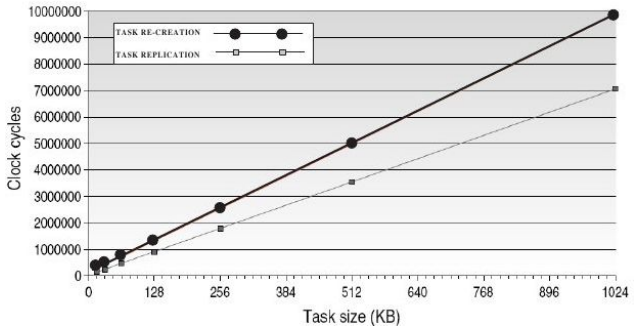

Figure 3.5: Migration levels differ in implementation complexity, performance, transparency, and re-usability.

3.4.3 Cost

In each method, there is, of course, a cost. However, the costs of both methods are not equal. Time (in processor cycle) is here taken as the cost of the methodology.

[3]PIC is a body of machine code that is stored in the main memory. It is executed properly regardless of its absolute address. PIC is commonly used for shared libraries, so that the same library code can be loaded in a location in each task (process) address space where it will not overlap with any other uses of memory (e.g. other shared libraries)

In both cases, there is a contribution due to the amount of data transferred through shared memory. Moreover, regarding task recreation, an additional overhead is always required due to the time elapsed to reload the task in the file system, which gives the task replication a benefit over task recreation in terms of time but with data loss due to replication. The cost for both methods of task migration versus the task size is depicted in the Fig 3.5. The larger slope of the task recreation hence explains that the reloading time increases linearly with the task size.

It is worth noting that the memory overhead in case of task replication is getting quite considerable in time with increasing number of threads in massively parallel architectures. This puts a limitation for the simplicity of having this solution in future platforms with increasing number of nodes.

3.5 Parallel applications and dataflow programming model

As architectures are developed to be parallelized, programming models advance to support such parallelism so that applications can be executed in the most efficient manner. In order to benefit from the fact that multiple cores are running simultaneously, an application has to be split into different activities that are able to be concurrently executed. This makes, as a result, the elapsed time to finish the same application shorter compared to the case of its ordinary sequential execution. This concurrency must not incur large overhead so as not to reduce the desired efficiency. Splitting application into smaller concurrent activities results in an issue which is the coordination of these concurrent activities and/or synchronization between them. Such coordination leads to sequencing of the parallelized application activities, hence, diminishes the gain of the concurrency. In order to alleviate this, speculative execution is adopted with the risk of misprediction that leads to not only longer execution time but also more power consumption.

Dataflow model of computation presents a natural choice for achieving concurrency, synchronization and speculation. Activities in such model are enabled once all necessary input data is available. Thus, all concurrent activities can be executed concurrently and synchronization among them is undergone via the flow of data. Dataflow model of computation is based on the use of graphs to represent computations and flow of information among the computations.

Dataflow applications are described in directed acyclic graphs DAGs that explicitly state the connections between nodes via arcs. Nodes (or activities) represent units of computation that receive input data (or tokens) and process them and produce output data. A *token* is a term that denotes data without any regard to its value or type. Tokens are transferred via arcs. An *arc* is a unidirectional path between two nodes through which tokens are transmitted. Every arc has one writer and one reader. It can store data until the size of data reaches its maximum, this

is called *arcf capacity*. Tokens are stored in first-in first-out FIFO buffers; hence, each arc has a FIFO queue. A node is triggered to work when all required input tokens are available in the input FIFOs.

3.5.0.1 KPN model

Before discussing Kahn Process network, it is worth mentioning the definition of a process (task) network; it is the way by which an application is described in flow based programming paradigm. Applications -in that paradigm- are referred to as data factory where data travel through a series of sequential operations like black boxes, each one is a process (task). This network of processes is called process network.

Kahn Process Network KPN is one of the earliest and the most famous dataflow models. An application in this model is composed of a set of processes (called nodes) and channels (called arcs). Fig 3.6 depicts an example of an application in KPN and its corresponding graphical representation whereas equations 3.1 represent the data processing functions of each and every node. In KPN model the arc capacity is infinite. Also, the tokens are not timed; i.e., no time stamps are maintained. It is explicitly required that communication latency is finite.

When a node reads from a queue, the read operation is blocking. If the queue is empty, then the process suspends until the availability of data. While reading operation is blocking, write operation is a non-blocking because the size of FIFO queue theoretically is infinite.

There is no finite duration of time within which KPN applications run as it is designed so as to work whenever input tokens are available. That is why for a KPN applications, it will be undecidable whether or not a Process Network terminates. It is also undecidable whether or not a Process Network will require bounded memory. However, since we are interested in Process Networks that run in infinite time, we could decide to let the scheduler find a bounded memory solution using infinite time.

KPN programs are determinate; i.e., the history of tokens produced on the communication channels do not depend on the execution order. They may be executed sequentially or in parallel with the same outcome. These systems support recurrence and recursion. Termination of KPN program does not depend on the execution order, only on the program. It occurs when all processes are blocked waiting for input tokens.

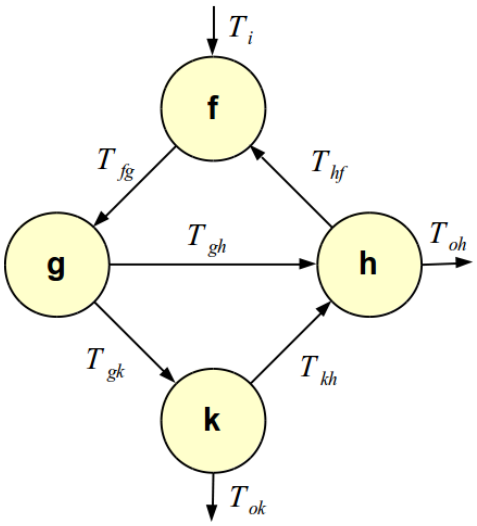

Figure 3.6: Kahn Process Network, T_i: *input* tokens, T_{xy}: tokens that are produced by node x and sent to node y, T_{ox}: tokens output by node x.

$$\begin{aligned} T_{fg} &= \mathbf{f}(T_i, T_{hf}) \\ T_{gk} &= \mathbf{g_1}(T_{fg}) \\ T_{gh} &= \mathbf{g_2}(T_{fg}) \\ T_{kh} &= \mathbf{k_1}(T_{gk}) \\ T_{ok} &= \mathbf{k_2}(T_{gk}) \\ T_{oh} &= \mathbf{h_2}(T_{gh}, T_{kh}) \\ T_{hf} &= \mathbf{h_1}(T_{gh}, T_{kh}) \end{aligned} \tag{3.1}$$

3.6 Problem statement

A brief explanation is necessarily given about the project. This is in order to understand correctly the rationale behind the chosen design choices of task migration solution. This work has been held under the banner of EURETILE project, EURETILE stands for European Reference Tiled Architecture Experiment. It is funded by the European Community through the Seventh Framework Program FP7[4].

The main aim of EURETILE project is to create a reference experimental platform for brain inspired many-tile architectures. The main objectives of this project are as follows:

[4]EURETILE project is under the Grant Agreement number 247846

- Provide a hardware prototype integrating at a number of tiles.
- Provide a many-tile programming environment, where applications can be expressed as dynamic network of processes and can be dynamically and mapped and controlled in an efficient and scalable manner on brain inspired architecture.
- Implement many-tile fault tolerance at system-level and include the support of all software and architecture layers.

Brain inspired architecture is a homogeneous distributed one. The processing cores in this distributed architecture run as an application a model of neural networks that is designed to run on embedded systems efficiently. This neural network model[5] is an accurate and an accurate and computationally efficient model that simulates human brain behavior. It is worth mentioning that details about the brain model adopted to be used in EURETILE are out of the scope of this work. In the following, project software environment is expanded, then, problem statement is explained.

3.6.1 Software environment

The main phases of the software environment of the project are illustrated in Fig 3.7. The first stage is the development of EURETILE applications that are parallel applications. Applications are developed in C languages. Every application is split into a number of tasks. These applications, as well as, their process networks are then input to hardware/software co-design process in which system level specification, analysis and optimization take place. Applications tasks are simulated covering several scenarios to find the best design point according to specified design objectives. In this phase, an extended programming model is introduced. It is called distributed application layer (DAL). DAL supports programming of large-scale dynamic applications, introducing a two-level process network model, control mechanisms, and a complete API that are refined in the lower software layer of hardware dependent software HdS. As well, DAL introduces the first levels of fault-tolerance mechanisms that require full support of hardware, hardware-dependent software, and simulation platform. This phase results in the mapping of applications tasks on the available number of tiles in the system.

3.6.2 Distributed Application Layer (DAL)

Many-tile hardware architecture offers to application programmers a high potential to exploit pipelining, parallelism, and concurrency in their applications at different degrees of granularity. Targeting such a powerful architecture becomes mandatory for new applications from the domain of high performance computing, high

[5]The benchmark of Izhikevich model of Polychronous Neural Networks [11] is selected in this project.

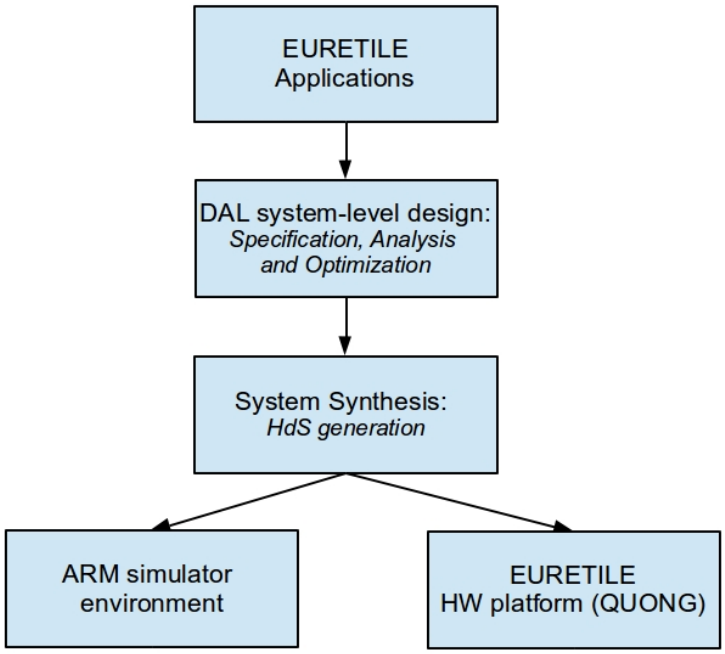

Figure 3.7: Overview over EURETILE software environment.

performance digital signal processing, and control, which have highly demanding computational needs, express a large degree of parallelism, and are increasingly dynamic. In order to execute efficiently these applications on a multi-layer complex platform, an innovative programming model is being investigated that directly represents the three levels of concurrency and parallelism present in the brain-inspired many-core paradigm.

DAL allows the application programmer to specify the concurrent execution of several dynamically instantiated applications on the same hardware platform, while still guaranteeing real-time constraints. Then a software tool-chain shall be supplied. This tool-chain is supposed to integrate and support DAL as specification and coordination language.

3.6.2.1 Basic conception

DAL is the more sophisticated version of distributed operation layer DOL, the latter is a programming model that is introduced in [12]. The difference between them is that DOL only considers programming and mapping of a single and static application on multi-core architecture. Mapping decisions are made at compile time and the mapping never changes during run-time. In order to extend this concept towards dynamic and large-scale multiple applications which is the scope of DAL unlike DOL, we need to capture the possible dynamic behavior in the application specification, as well as control mechanisms at system-level, all these concepts cor-

responding to the third level of parallelism.

For the sake of clarification, dynamic behavior needs to be into classified into two classes as follows:

- *Behavioral* dynamics only affects the behavior of the application. Introducing new tasks to the system, as well as, destroying existent running tasks in the system changes the actual behavior of the application(s). However, from architectural viewpoint, neither introduction of new tasks nor destruction of existent ones affects the architectural characteristics of the system.

- *Architectural* changes at run-time may necessitate adaptive task remapping (or migration). Maximum temperature constraint violation in a computation core or permanent failure of some hardware can be seen as examples of architectural dynamics.

A scenario is defined as a system state in which a predefined set of applications is running, being paused, or being stopped simultaneously. Then the dynamic application behavior can be seen as a transition between scenarios. Transitions between scenarios model the dynamics of applications behavior, i.e., addition/deletion or pausing/resuming applications. Such transitions between scenarios are described in a finite state machine. They are triggered by run-time events that are emitted by controlling tasks. These tasks are added to the number of desired applications in order to control their execution.

An example of four applications (Application 1, Application 2, Application 3, and Application 4) and their process networks are illustrated in Fig 3.8. Their execution is described by three simple execution scenarios illustrated in Fig 3.9. There are three possible operation scenarios (Scenario 1, Scenario 2, and Scenario 3). Application 4 is the interface that issues the events that, in turn, trigger the transitions. All events are listed in table 3.1 whereas table 3.2 denotes the condition and the desired action for each transition.

Table 3.1: Actions for each event.

Event	Description
e1	run App1
e2	run App2
e3	run App3
e4	stop App1
e5	stop App2
e6	stop App3

In full behavioral dynamic, any application can be added or removed in any system state without restrictions. Full behavioral dynamics can be allowed instead

Table 3.2: Transition and events interaction.

Transition	Condition	Action
t1	e4	stop App1
t2	e2	start App1
t3	e3 & e4 & e5	start App3, pause App1, pause App2
t4	e6 & e1 e2	stop App3, resume App1, resume App2
t5	e3 & e2	stop App3, resume App2
t6	e6 & e5	pause App2, start App3

(a) Application 1

(b) Application 2

(c) Application 3

(d) Application 4 (Interface)

Figure 3.8: Example.

of predefining operation scenarios at design. This is a preferred alternative for more complicated systems that have a large number of applications. However, the static mapping is nearly impossible since the number of scenarios that must be considered at design time becomes infeasible. But typically unfeasible combinations of applications are pruned to alleviate the burden paid in design time and to provide a basis for predictability in design like the example given in Fig 3.9.

3.6.2.2 DAL-level control mechanism and implementation

Since transitions between scenarios are triggered by events that are issued by tasks like interface task in the last example illustrated in Fig 3.8. Such interface task is supposed to have information about the scenarios FSM. It also represents the controlling task that is used in DAL programming model to control multiple applications execution. The control mechanisms are discussed here briefly so as to know how DAL controllers are supposed to work.

There are different approaches that are used to deploy DAL controllers. They are briefly expanded in the following:

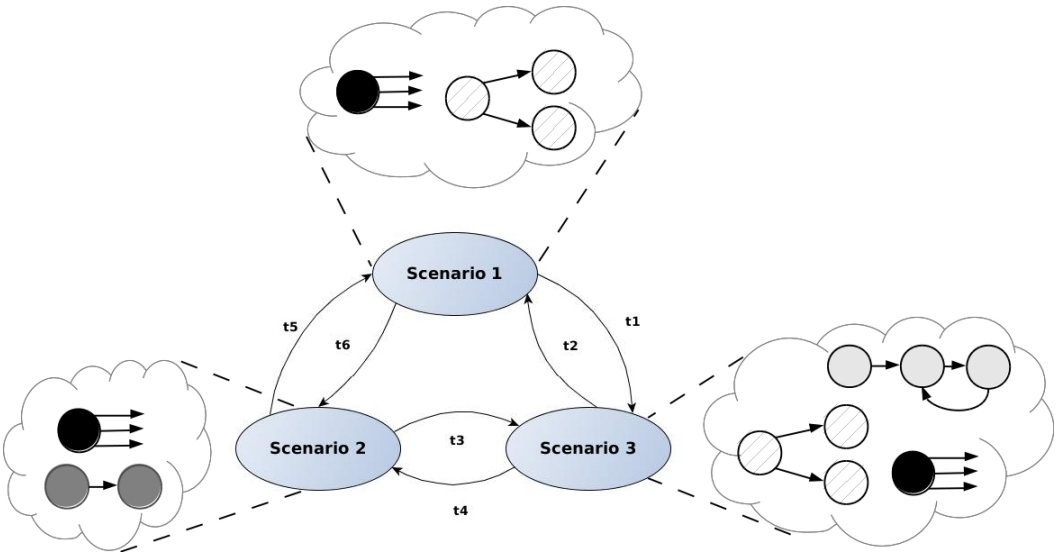

Figure 3.9: FSM of 3 scenarios combining 3 applications.

Fully-centralized: This approach is based on the existence of a central controller. The role of such controller is to detect behavioral or architectural dynamics and performs accordingly what is necessary to change the system scenario. This is undergone totally by such single central controller all over the system. The main advantage of the fully-centralized solution is that it is relatively simple to implement. There should be a system management connection[6] between such controller and every other tile. The controller should have global knowledge about all the existing tiles and their relevant corresponding states. The main disadvantage of this approach is its centralized nature which may impose a performance bottleneck.

Fully-distributed: In this approach, each tile in the system is autonomous and establishes system management connections to a subset of the tiles only. In this case, if a tile fails the system does not break necessarily. One main disadvantage of this policy is the expected complexity of the system control architecture. Another considerable disadvantage is the number or management connections between tiles, such number is very likely to be very large as its relation with the number of tiles is not linear but exponential.

Semi-distributed: This approach is a hybrid solution that contains the previous two. Its control policy is hierarchically centralized as there is a master DAL controller and in a lower control level slave controller. One master controller is responsible of issuing control commands to slave controllers. As a result one slave controller exists in every tile while a single dedicated tile houses the

[6] An abstract reference to a means by which control commands can be exchanged. It can be a channel when implemented.

Table 3.3: DAL application control

Action	Description
Start application	Stop each task in the application and delete any context (Context includes local state of tasks as well as data stored in the connected FIFO queues).
Stop application	Execute init method of each task in the application and execute fire method repeatedly
Pause application	Stop calling fire method (in the scheduler) of each task in the application
Resume application	Execute fire method repeatedly (without calling init method)

master controller. This approach actually takes into account system scalability to arbitrarily large systems without any concern about increasing number of management connection as in fully distributed approach.

The semi-distributed approach is adopted for EURETILE, hence DAL controller are placed accordingly throughout the system. Because the mapping is dynamically managed by the control FSM, all events should be delivered to the control FSM. Events originate from controller tasks. Additional DAL Application Programming Interface APIs for this purpose need to be defined for such purpose as listed in table 3.3. Once an event is delivered to the controller, the proper action should be taken to relocate the tasks in the system according to the selected state transition.

3.6.3 Fault-avoidance solution description

It is required that the system shall be equipped with task migration capability. This is to enable the system to be responsive to thermal hot spots by reducing the number of tasks running on hot tiles, hence, avoid thermal stresses. Several issues have to be resolved by the design of task migration solution, they are as follows:

I Migrating tasks should not be restarted on their destinations but resumed; this implies the transfer of task state guaranteeing the safe and correct pausing and resumption of a migrating task on its source and its destination, respectively.

II System architecture has to be taken into consideration such that moving tasks shall not depend on a globally accessible address space in a shared memory. This makes the transfer of task state undergone by explicit inter-tile communication.

III Task migration solution should work on embedded operating systems without the need of any of features that exist in full-fledged operating systems. Such embedded operating systems tend to small and basic to have a small memory footprint. They most probably do not support memory management unit MMU or have dynamic loading.

IV Applications code should not be modified to account for migration. In other words, application developers should not care while they are writing their codes about how tasks can be migrated dynamically in run-time.

V Task migration solution should fit DAL design flow.

Chapter 4

Sate of the art

Contents

Task migration has been implemented in different ways and on different levels. Not only do the implementation techniques vary but also there are several challenges that do not make the implementation job as easy as it might be expected. This chapter is dedicated for expanding such issues and stating the related work in circumventing them. In addition to that, it lists the state-of-the-art in implementing task migration in the literature.

4.1 Migration implementation

As explained in section 3.4, there are different levels of implementation of task migration. Application, OS, and kernel level are all possible levels in which the solution could reside. The implementation trade-off also has been explained.

In this section, some solutions will be expanded mentioning the architecture of the platform, the level of the solution and whether the solution is for embedded systems or servers.

4.1.1 Migration using shared memory

Implementing task migration in shared memory SMP is easy thanks to shared memory that provides quite easy communication means. It results in no need of any data (task address space) transfer or task code. This is because of the fact that

all processors are entitled to access every location in the shared memory, migrating a task comes down to electing a different processor for execution. There are several operating systems that support that such as Windows and Linux [13] and also DNA-OS [14] which is chosen to be our OS for the rest of this work.

Task migration has been explored for MPSoCs as in [15] in which task migration is based on locality consideration for decreasing communication overhead or power consumption. In [16], a migration case study that relies on the μClinux and checkpointing mechanism had been presented. The system uses the MPARM framework [17]. Also the whole system supports data coherency through the shared memory.

In [18], scalable shared memory many-core architecture with global cache coherence implementation is presented. The architecture is built around 4096 cores which makes use of a logically shared memory but physically distributed with cache coherence enforced by hardware using a directory based protocol.

4.1.2 Migration using distributed memory

In the case of migration in distributed memory MPSoCs (NUMA architecture), task state must be migrated from source node (processor) private memory to the destination one with the possibility of migrating the code too in case of not using task replication as shown before in section 3.4.1. The manner of communication and synchronization is determined by the architecture. Pure message passing interface **MPI** takes place in case of NORMA, while exchange of messages could take place via global shared memory in case of NUMA with shared memory.

There are several experiments done on distributed memory MPSoCs in literature which use shared memory like [16, 10, 9, 19]. This is due to the fact that architecture of embedded MPSoCs often uses memory organization of the type NUMA as mentioned before. There are different solutions according to the provided NUMA architecture.

In [10, 9], dynamic task migration has been implemented by checkpointing mechanism. A master daemon running on master processor is responsible of dispatching the task to be migrated while one of slave daemons running on all other slave processors receives the migrating task. This solution adopts task replication to avoid the need of dynamic loading or PIC. It also provides task migration facility in middleware layer. Shared memory is used as a communication means between processors; however, every task resides in local memory. The architecture is sort of similar to that depicted Fig 3.2. In [16], a migration case study is presented and for MPSoCs with μClinux OS and checkpointing mechanism. In [20] MMU does not exist, therefore PIC was used to implement dynamic task migration with checkpointing. Another solution in [21] includes MMU-less homogeneous bus-based

multi-processor system with three 32 bit RISC cores with a powerPC and shared memory. In [22] task are frozen and recreated in the destination processor in a bus-based with MMU-less multi-processor. Almost all the solutions proposed use task replication due to the lack of dynamic loading to the majority of small embedded operating systems.

In [19] another policy of migration is presented, it exploits the temperature as well as workload information of streaming applications to define suitable run-time thermal migration patterns. The approach depends on task replication too. The source processor stops execution of the migrating task and the destination one starts running its own replica without the migration of the task state. The processors communicate with each other via the shared memory.

In [23] dynamic task allocation strategy is proposed. In other words, migration takes place at run-time in a manner that is based on bin-packing algorithms in the context of MPSoCs. The whole context (code, data, stack, and contents of internal registers) is migrated and there is no task execution during the transfer. The inter-processor communication is based on MPI (send/receive primitives). However, in this work neither explanation about the task migration protocol nor the impact in term of performance of such mechanism is given.

It is worth noting that task replication is always simpler to be adopted but it has considerable limitation due to memory bloating with increasing number of threads and nodes as mentioned in section 3.4.3.

4.1.3 Migration in NORMA architecture

In the case of NORMA architectures, there are recent task migration solutions that are proposed in [24, 25, 26, 27, 28, 29]. They are targeted to NoC-based distributed memory architectures MPSoC. We investigate these solutions showing how they are different from the one proposed in this work.

In [24], an OS is used on which a dynamic loader is developed to support task code migration. However, task context migration is not supported. Consequently, such technique is effective only in state-less tasks or the application would have to be restarted from the beginning in the destination. Migration was applied in the context of dynamic remapping for better performance; hence, it is not shown the impact of the migration on performance. It is stated, instead, that migration cost is amortized by performance gain due to remapping, this is premised on remapping is always performed for better performance. It is not mentioned; however, how transparent the solution is to the application. Unlike this solution, ours supports state-machined applications as task state is migrated, runs on real hardware. Performance overhead of migration is measured along with memory usage. In [28], almost the same solution is proposed as in [24] but without code transfer.

In [25], they deploy task migration technique aiming at NoC based MPSoC that does not include code transfer. Regarding communication, MPSoC Message Passing Interface MMPI is used. It is a parallel programming pattern which makes the parallel program independent of task mapping. The solution is based on master and slaves tasks that manage the migration process. The master resides in a separate processing element PE and runs on Linux OS while slaves run on microC/OS-II. It is stated that this technique runs on a simulation platform.

In [26], research work is done mainly to provide simulation platforms of different architectures for simulating task migration; therefore, migration is not shown from either the algorithmic or the methodological point of view. Simulation platforms for both UMA and NORMA architectures are described. To show results migration experiments are run on those platforms. They use a middleware layer that provides APIs to support migration. They apply task replication so that all tasks are migrate-able and their replicas are everywhere. The impact on scalability of replicating all tasks codes on all PEs is not shown. Communication migration is handled by a component they added in the simulation platform called run-time resource scheduler RRS. It is responsible of managing all HW resources and all communication. Experimental results are shown in a relative manner between different remapping decision techniques. They show the performance overhead of applying random task remapping RTR versus applying energy-aware mapping versus no migration at all. Different multimedia applications are run. The results show the time of processing per frame; however, jitter time which is introduced by the migration is not shown but rather the impact of the remapping regarding the improvement in the communication over NoC as an example. That is why there is not much difference between some cases where mapping decisions might lead to the same results.

In [27], they address communication inconsistency problem of task migration. They use agent based solution to manage communication before, during and after migration. They used protection switching method to resolve the inconsistency problem. This is via forwarding without stopping the communication. Unlike the solution proposed in this work, no OS is used; instead a bare metal run-time system that handles the management of tasks is used. This means that the whole low level software is customized for specific HW including communication drivers that should allow the forwarding of messages. Although communication inconsistency is addressed in their work, it may not be generic to fit different platforms. No details are given on transparency or portability of such solution.

In [29], the proposed task migration solution targets polyhedral process network PPN model of computation on a multi-tiled architecture. Task migration is used for system adaptivity. PPN is a special case of KPN. The nature of applications used are static affine nested-loop programs SANLPs [30], they are nested loop-based task models, each has two iterators where its loop boundaries, condi-

tions, and variable indexing functions are affine functions of the loop iterators and parameters. They use *pn* compiler [31] to convert SANLPs to parallel PPN specifications and determine the buffer size that guarantees deadlock-free execution. This conversion imposes some restrictions on the specification of input application. Unlike the adopted KPN model in this work that allows a task to have a state, PPN model requires a stateless task. As a result, their task migration solution is almost communication migration since the state of a PPN task is represented only by:

- Content of input and output FIFOs
- Task iterators.

Their task migration technique uses run-time task re-mapping. They use a middleware based solution where this middleware layer provides migration aiding APIs. In this layer, a request-driven (R) communication approach is also implemented. This communication approach is the one used due to its simple and easy implementation as it requires less synchronization points. In this approach, communication is initiated by the receptor where it sends asking for data tokens to the sender and once the sender receives this request, it sends required tokens to the receptor. Input tokens remain in input FIFOs even if they got consumed, FIFOs are released later after writing output ones. This also applies to writing tokens, hence, reading and writing tokens, each is split into three actions (listening to the channel, copying from HW FIFO to SW one or the contrary depending which action is being performed, and consumed tokens release). They use an interrupt-based task migration where a run-time manager sends a message to source tile that interrupts the migrating task. In order to be able to interrupt a migrating task, the network interface NI is extended to be able to generate interrupt once it receives a certain type of messages, this makes it a special NI. Migration is disabled in some parts of the used PPN task model, these parts are where task iterators get updated and in the parts where old written or read tokens get deleted. Run-time manager sends to source tile migration request which interrupts the migrating task and then source tile sends to other related tiles which are tiles that have successor and predecessor tasks. However, it is not clear in the work whether there can be more than one successor or predecessor. It is not clear either if this migration technique can function when a tile can run more than one task using context switching. There is a corner communication case where a migration interrupt can occur right after the migrating task sends a request to read input token(s) from its predecessor task so that the predecessor task sends the token(s) to an interrupted migrating task, this would lead to the loss of these input tokens. They cover this case by making the predecessor tile able to sending another interrupt to source tile to re-direct all tokens to destination tile. This, however, does not clarify what would happen if the input FIFO on the source tiles was deleted before receiving tokens coming from predecessor tile or what would happen if due to some congestion in the NoC, these tokens are delayed while the migrating task already started running on the destination tile and sent for input tokens from predecessor tile, this would lead to a possible out-of-order reception of tokens which

may lead to abnormal outputs. There are also cases where a migration interrupt can be generated right after the calculation of output tokens before writing. This leads to the repetition of this iteration again at the destination lowering the benefit of using an interrupt based migration.

To our best knowledge, there is no work in the literature that implemented a full, scalable and automatically generated task migration solution to run on real hardware with multi-tiled architecture. Almost all the implementations and their validations are performed either on simulation platforms or FPGA based custom HW platform. The measurements are usually performed in a relative manner and are seldom shown in absolute manner. This makes migration overheads not very clear to application execution time. Issues like transparency to application programmers and portability to different core architectures are rarely explained, this poses question about how applicable the solution to legacy applications or how possible the solution can adapt to different core architectures. Finally, there is no technique that investigates how migration performance overhead changes regarding the number of neighbors in a distributed architecture. Our solution is expanded with details both from methodological and algorithmic points of views. It is plugged into automatic generation tool-flow, this makes both generation for different core architectures and use with different application easy thanks to its portability and transparency, respectively.

4.1.4 Migration points

Some methodologies of implementing migration require that the process/thread has some *predefined* points for migration, i.e. it is not allowed to migrate at any arbitrary instant while executing the code. The process/thread proceeds execution till one predefined migration checkpoint in the code comes, at which the process/thread checks if it is required to be migrated or not; in case of no requirement received the execution proceeds normally till the next migration checkpoint and so on.

In [32, 33, 10, 9] migration points are used. In [32], for instance, the task code looks like listing 4.1 in which the pseudo code illustrates the way the code is checkpointed. In the same work, task replication which was expanded in section 3.4.1, had been implemented in a NUMA with shared memory, i.e. offline code is deployed on all the nodes CPUs and the migrating task shall be resumed just on the destination ones. Source node suspends the migrating task T_m after saving its context and then it writes the saved context in the shared memory. Communication must take place prior to task migration between source and destination nodes. After this communication the destination node reads the context from the shared memory and resumes the execution of the task without the need of code transfer. This is further depicted in Fig 4.1.

```
for (;;){
  ...
  if (migr_point == FIRST_MIGR_POINT){
     //-------------------------------------------
     // User's code
     //-------------------------------------------
     migr_point = SECOND_MIGR_POINT;
     if (request_migration){
        save_context();    //important data
        tell Sys_T_data_is_ready();
     }
     if (migr_point == SECOND_MIGR_POINT){
     ...
     }
  }
}
```

Listing 4.1: Task migration checkpoints.

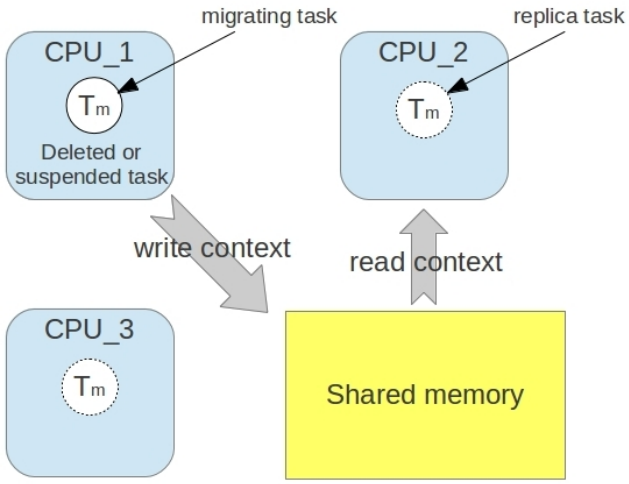

Figure 4.1: Task migration by replicas.

4.1.5 Dynamic loading

As mentioned before in section 3.4, task recreation is a task migration methodology that requires the transfer of the task code. Task transfer requires, in turn, the OS supporting dynamic loading which is a feature that is not easily found in embedded operating systems since they are always designed to be as light as possible with small memory footprint. There are a few embedded operating systems that support dynamic loading that is required by their domains in which they are used like Contiki [34]. Task code relocation is a research topic and found in the literature like in [24, 35]. However, such solutions are mostly custom ones depending on some parameters like the OS, the compiler used, and the linkers.

4.2 Issues of migration

There are some challenges which hamper functioning of the task migration. They are expanded in the following sub-sections. These challenges depend whether the migration be among threads *thread migration* only or among whole processes *task migration*.

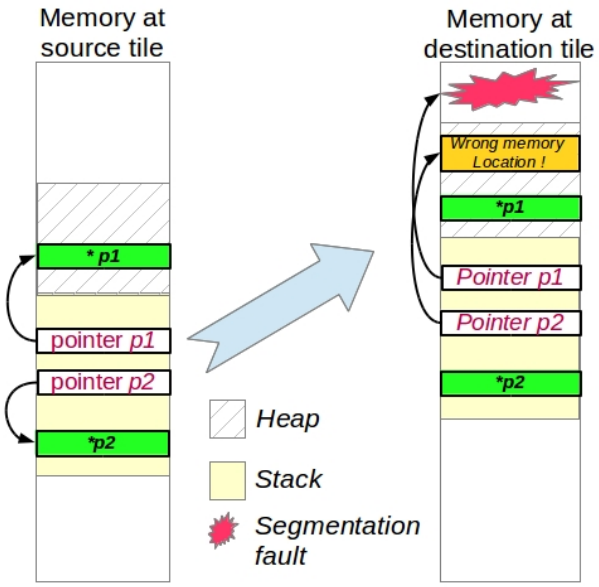

Figure 4.2: Migration of threads with pointers

4.2.1 Address collision

This problem appears in both cases of migration, i.e. during thread migration and task migration. It can be best explained as follows:

In case the migrating task (or thread) has pointer(s) pointing whether to locally declared variables or to dynamically allocated variables (i.e. stack or heap), it will never be able to resume execution correctly in the destination core. This is obviously because that the addresses values of the pointers have been updated prior to resumption at the destination node. In fact, the memory part allocated for the migrating task and/or thread is not guaranteed to reside in the same exact range as it was in the source node memory range. Consequently, the pointer values (which are the addresses of other variables and/or functions ... etc.) which remain the same even after migration will be pointing to illegal memory locations[1] as depicted in Fig 4.2, in which `p1` and `p2` are pointing to a dynamically allocated memory location residing in the heap and a local memory location residing in the

[1] Places in the memory that do not belong to the legal range of the process and/or thread.

stack. They both point to different places in the memory after migration when the whole stack and heap reside in different address range in the whole address space. Therefore, without prior proper preparation to the migration, there is no guarantee that the addresses will keep their consistency for proper continuation of execution.

This is also explained in the listing 4.2. In which the pointer `ptr` points to a local variable `x`. When resuming execution after migration, any attempt to change the value pointed to by the pointer `ptr` creates `segmentation fault` error.

```
void thread_function(){
  int x;                        /*local variable*/
  int *ptr = &x;                /*local pointer*/
  x = 1;
  printf("x=%d\n",x);
  *ptr ++;                      /*causes segmentation fault*/
  printf("x=%d\n",*ptr);
}
Execution before migration (on source node):
x = 1
x = 2
Execution after migration (on destination node):
x = 1
segmentation fault
```

Listing 4.2: Thread migration in the presence of pointers.

There are solutions to circumvent address collision problem which can be categorized according to their approaches to three approaches as in [36]. The first approach requires language and compiler support to maintain adequate type information and identify pointers as in [37]. This is, of course to overcome the problem of C language as it is not strictly type-safe[2] language. This approach suffers poor portability which is a quite drawback.

The second approach [38] scans the stack at run-time, i.e. dynamically after compilation, to detect and consequently, translates and updates all pointer values to fit the newly allocated memory portion. However, it is quite possible that some pointer cannot be detected and resumed execution might go wrong and suffers from any possible `segmentation fault` as seen before.

The third approach mandates the partitioning of the address space and reservation of unique virtual address for the stack of each thread so that the internal

[2]**Type-Safety** is the extent to which a programming language can prevent or even warn type errors. It is also meant by type error, the undesirable behavior of a program due to a discrepancy between different data types of constants, variables and functions of that program, e.g. treating integer values as floating point ones. C language, in fact, is type-safe in limited contexts; for example, compilation errors pops up when any attempt is made to convert a pointer pointing to structure to point to another one with explicit casting

pointers remain the same values. A common solution adopted is to preallocate memory space for threads on all machines and restrict each thread to migrate only on its corresponding address on other machines. This is also called *iso-address* methodology. This will be expanded in a separate following section.

In the following sections, two examples of the previous approaches are to be expanded. The solutions are iso-address as an example of the third approach and user-level migration support libraries as an example of the first approach. It is believed in this work that the second approach is not completely feasible for the previously mentioned reasons as well as the uncertainty in updating the *implicit* pointers[3].

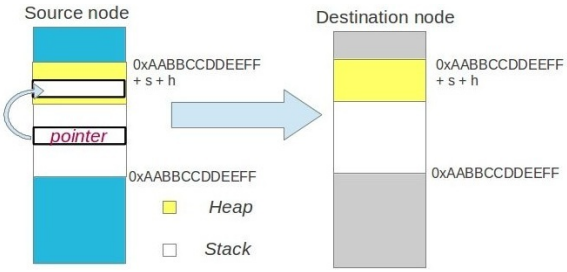

Figure 4.3: Migration of threads stack + heap with pointers to exact same address range in the destination node address space

4.2.1.1 Iso-address

One way to resolve this problem is by using iso-address method as in [39, 40, 41]. This approach had been used on cluster computing like the Parallel Multithreaded Machine **PM2** which adheres to single program multiple data **SPMD** programming model. In SPMD, the code to be executed by the thread is replicated on each node and is not part of the thread. This methodology does not deal with thread migration with shared data which is another issue that will be discussed in section 4.2.2. The main idea of this methodology is that every node allocates storage area in a system-wide *locally* (in its own private memory) which is *globally* consistent. The allocation mechanism must guarantee that each range of virtual addresses at which memory has been mapped at some node is kept free on all the other nodes. In other words, the memory portion allocated on one memory belonging to some node must be the same in the destination node memory in the sense that the task stack must reside in the same address range so as to avoid any post-migration update of the pointers as shown in Fig 4.3.

[3]pointers that are created implicitly by the compiler sometimes in order to chain the stack frames, while *explicit* pointers are those which are declared explicitly in the application by the programmer

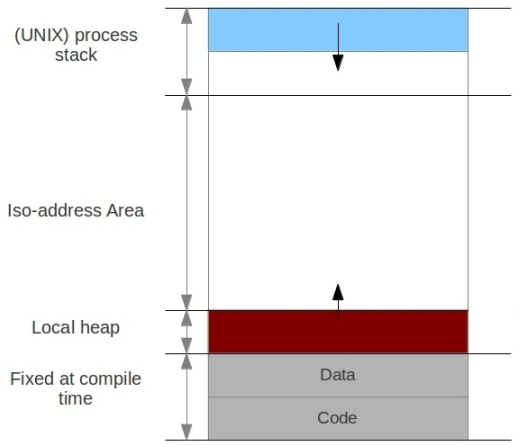

Figure 4.4: All the nodes have the same memory mapping

The implementation of this solution is mainly by devising new memory allocation mechanism by `isomalloc` (instead of the ordinary `malloc`) which circumvents the problem of dynamic memory allocation. This allocation mechanism relies on the architecture as well as some rules. This is to ensure that every node may use its globally reserved memory without having to inform the other nodes. This is to avoid the synchronization among the nodes. Those rules can be expanded as follows:

I The physical execution environment in PM2 is *homogeneous* with the same copy of OS run by all cores. Moreover, all nodes have the same memory mapping: the same binary code is loaded on each of them (main characteristic of SPMD as previously mentioned) at the same virtual address[4]. The (unique) process stack is also located at the same virtual address on all nodes and this will be further explained and shown later.

II All iso-address allocations take place within a special address range called iso-address area, this is the case for every node. We have located it between the process stack and the heap as shown in Fig 4.4. This zone corresponds to the same virtual range on all nodes.

III The virtual iso-address range is chopped to equal partitions (slots). These slots are *globally reserved* on each node. This is to ensure that only one node uses a single slot at a time.

IV Any allocation in the memory is undergone *locally*, i.e. local memory to the node.

Given the previously listed items which describe the environment and the rules, the implementation approach is done by developing *slot layer*. This is for allocating

[4]No task and/or thread code needs to be transferred to the destination node during migration.

and managing memory slots as previously explained to tasks/threads on the nodes. This is to be done in global consistent way so as to avoid nodes synchronization. The iso-address part shown in Fig 4.4 is divided to equal portions (slots) as explained before. Not all the slots per local memory can be used freely by the owning node, there will always be reserved slots for the other node. In an example of two nodes, slots can be divided to even and odd slots in each node such that even slots in one node is reserved for (owned by) the other node and exactly the inverse holds true for the other node. In Fig 4.5, an example of slot management is depicted. Thread A is created in Node 1/2 and acquires a slot owned by the node locally to store its stack as in Step 1 quadrant. The thread data expands, hence acquires other slots from the local node (it is noticed that they are not contiguous) for extra data storage as in Step 2 quadrant, also another thread B is created on node 2 and it is obviously clear that its slot reside in the first odd slot in the address space leaving the even 0 slot reserved for probable future migration of thread A from node 1 to 2 as it will follow. Thread A then migrates along with its slots from node 1 to node 2 as in Step 3 quadrant, it is quite obvious that they rest in exactly the same place in node 2 address space and these slots have become owned by node 1 thread A. The threads die and their slots are acquired back by the corresponding nodes as in Step 4 quadrant.

This solution has some advantages as stated in [39]. They are as follows:

Simplicity
: The migration mechanism is simplified, because no post-migration pointer update is necessary any longer.

Transparency
: Applications may make free use of pointers without having to take into account possible problems related to thread migration. User-level pointers are always guaranteed to be safe.

Portability
: No compiler knowledge about the thread stack structure is required, since the stack contents remains exactly the same after migration. In particular, compiler-generated pointers are migration-safe, too. Consequently, any compiler may be used and compiler optimizations are allowed.

Preemptive-ness
: Preemptive migration is possible, given that no assumption is made about the thread state at migration time.

However, there are, of course, other disadvantages of this solution which can be summarized in poor scalability and large address space requirement for implementation. To further explain that a simple equation must hold true, it is $ns = w$ where n is the number threads (or slots given that every thread requires only one slot), s is the address space of the slot (allowed for one thread) and w is the whole global

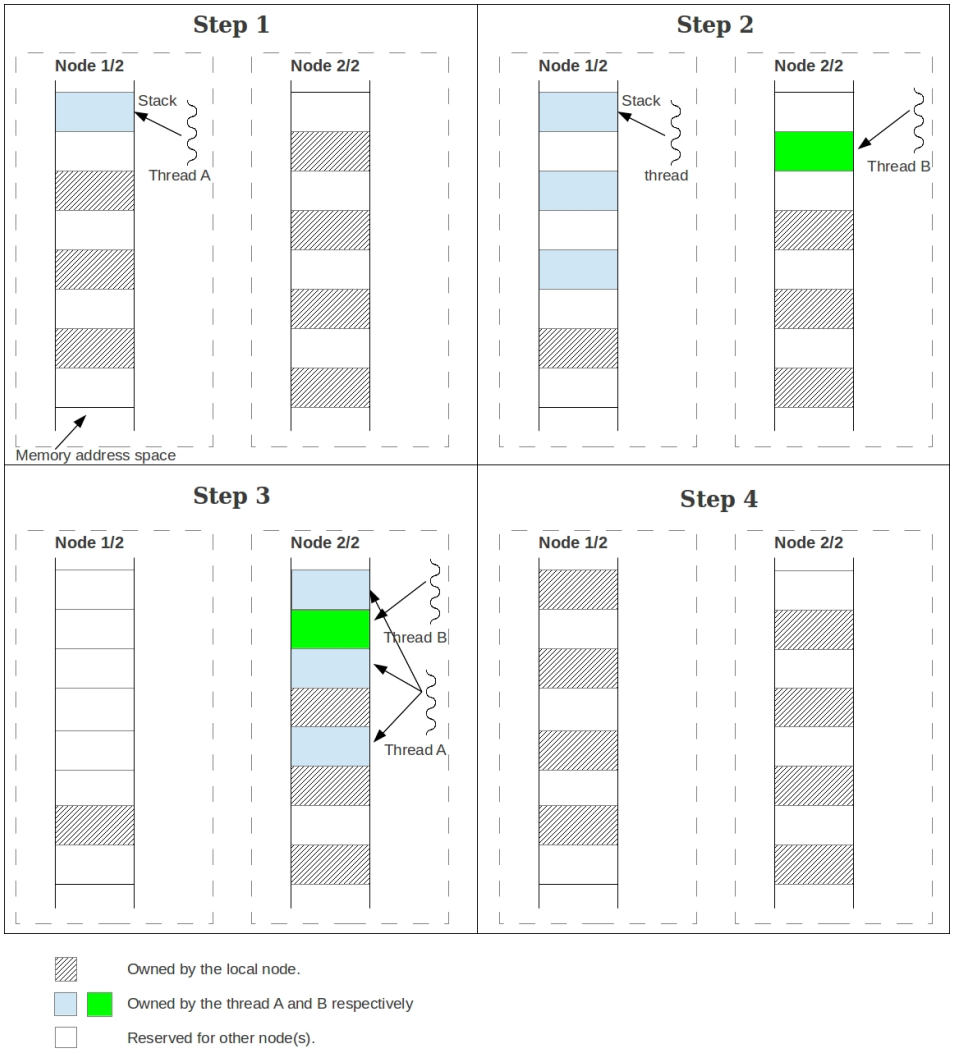

Figure 4.5: Example of slot management in this approach.

address space. More restrictions are to be added to the equation when reserving *spare* slots for probable future migration. In brief, this solution consumes a lot of memory.

4.2.1.2 User-level

As shown in the last section how the address collision was resolved by implementing some sort of OS support for iso-address management, in this section a completely user-level solution is expanded. In [36] application level libraries are to be developed to circumvent the problem of address collision by avoiding the conditions in which C is not type-safe. In other words, this solution tries to make C language as type-safe as possible. This is simply by inserting all variable and pointers (whether locally pointing or pointing to a dynamically allocated memory location) in special

```
void foo()
{
  //local variable
  int         a;
  //local variable
  double      b;
  //local pointer
  int        *c;
  //local pointer to pointer
  double   **d;
  .
  .
  .
}
```

Listing 4.3: Original function

```
void Mth_foo() {
  struct MThV_t {
     void       *MThP;
     int         stepno;
     int         a;
     double      b;
  }MThV;
  struct MThP_t {
     int        *c;
     double   **d;
  }MThP;
  MThV.MThP=(void *)&MThP;
  \\Rest of code ...
}
```

Listing 4.4: Transformed function

structures.

The solution in [36] proposes user level stack/heap management package called `MigThread`. It does not depend on special OS libraries, unlike the last solution `isomalloc`. `MigThread` package consists of two parts: a preprocessor (which carries out source code transformation) and a run-time support module. The preprocessor is designed to transform the user-level (application) source code into a format from which the run-time support module can construct the computation state precisely and efficiently so as to be able to resume execution on the destination node.

The core of migration/checkpointing aimed in [36] is to be able to store and reconstruct process state. This is done majorly in pre-compile time via transforming the code written by the programmer. The main reason why pre-compilation source code transformation is done, is the recuperation of all information related to stack variables, functions parameters, program counter and dynamically allocated memory regions. This information is collected into certain predefined data structure as in [42] which relates to the same context of [36].

An example is taken to illustrate the methodology applied in this solution. In the following two listings 4.3 and 4.4 the collection of data in predefined structures are shown. In listing 4.3 a function `foo` is defined with four local variables. The function is transformed by `MigThread` to be called `MTh_foo` instead and to look as in listing 4.4 putting local variables in `MthV_t` structure and the pointers in `MThP_t`. Within the latter there is an item `MThV.MThP` which is the only pointer to the second structure `MThP` which may or may not exist. In stacks, each function activation frame contains `MThV` and `MThP` to record the current function's computation status. The overall stack status can be obtained by collecting all of these `MThV` and `MThP` data structures spread in activation frames. The program counter **PC** is the memory address of the current execution point within a program. It indicates

the starting point after migration/checkpointing. When the PC is moved up to the language level, it should be represented in a portable form. We represent the PC as a series of integer values declared as MThV.stepno in each affected function, as shown in listing 4.4. Since all possible adaptation points[5] have been detected at compile-time, different integer values of `MThV.stepno` correspond to different adaptation points. In the transformed code, a `switch` statement is inserted to dispatch execution to each labeled point according to the value of `MThV.stepno`, and executed after the function initialization. The `switch` and `goto` statements help control jump to resumption points quickly.

It is worth noting that `MigThread` framework does propose a solution to reduce the problem of address space dependence that will be expanded in next section 4.2.2 by introducing special data structure to deal with global variables but in the case of whole process migration not just one thread.

There are also other application process interfaces **API**s for dynamic memory allocation. Also there are more details of state restoration and data conversion prior to restoration and resumption. However, for the sake of brevity, it is sufficient to clarify the idea about source transformation without expanding all the details of construction and resumption. It is also noted that this solution suffers from complexity and poor support in case of complex application. This is in addition to the fact that this solution does not deal with *implicit* pointers generated by the compiler.

4.2.2 Address space dependence

This problem appears only in the case of thread migration. Threads inside one process share a single address space as shown earlier in Fig 2.3.(a), especially global variables. Thus when one of the threads in the process migrates to another node, the moment it gets resumed, it inevitably results in inconsistency due to the lack of the global variables. However, even the migration of global variables will not help. Furthermore, an application programmer cannot use the global variables as he expects because they are actually shared with other threads in the same process. Note that the application programmer cannot even know which threads are in the same process because the framework should migrate the threads transparently.

As previously mentioned some frameworks tried to reduce the severity of the problem like the previously mentioned `MigThread` framework in [42, 36, 43]. This is also the case in PM2 architecture [39, 41] previously mentioned in this work at section 4.2.1.1 as well as in [37]. All these frameworks do not resolve the problem of the use of global variables in case of thread migration.

[5] Migration cannot occur in any instant, there are certain predefined states at which migration can be executed otherwise execution proceeds till an appropriate state comes which is called adaptation point.

In short, the use of global variables is not allowed in case of possibilities of thread migration. Therefore, the solution is to avoid the use of them in applications, i.e. coding guidelines do not allow them.

4.3 Conclusion

After mentioning different recent research work, we can conclude several points. The aim of this section is to list these points.

There is still an interest in task migration as a research topic among different research teams. Exploring different task migration solutions targeting different MP-SoC architectures still requires some research work in order to assess their costs in performance and memory.

The majority of the research works listed in this chapter is validated on simulation platforms except few ones. Not enough details are always given about the accuracy of measurements performed on such simulation platforms.

It is not easy to find numbers of performance overhead that are not relative, i.e. we cannot find the increase in execution time added by task migration process. Overheads in performance are always presented in a manner that depends on mapping as well as different parameters. Sometimes this increase is given as the extra number of cycles added to the execution time but without showing how much the application overall execution time is increased due to migration. Another parameters that contribute to impacting performance overhead are rarely mentioned like the impact of task state size.

In this chapter, the listed task migration solutions that target NORMA architectures rarely explain how communication inconsistency problem that results from task migration process is resolved. Few research solutions are dedicated to resolve such problem but such solutions are simplified, i.e. they do not address other issues like the transfer of task state or functioning on operating systems. However, in the other solutions, the resolution of the communication inconsistency problem is either not detailed or depending on specific HW or SW components.

In this work, we are interested only in the solutions that target NORMA architectures. This is because such solutions fit the NORMA nature of the multi-tiled architecture of EURETILE project. In the following chapter, we expand the methodology of our solution.

Chapter 5

Task migration methodology

Contents

The task migration solution is proposed in this work to fit distributed multi-tiled MPSoCs. In order for this solution to be operational, different implementation issues have to be resolved. In this chapter, we expand the adopted methodology of the proposed solution. We show also how such issues are circumvented and show rationales behind the design choices. We start by describing the overall solution. Then we go into details of the solution.

5.1 Solution overall description

The solution can be best described by answering the three main questions that accompany each and every task migration: `what, where to and how`

`What` to migrate?
Most suitable tasks should be migrated first, once reasons to migrate exist. Suitability, in fact, is determined according to parameters that contribute to producing heat like consumed CPU load.

`Where` shall the migrating task be migrated to?
The aim of this question is to determine the destination of the migrating task.

`How` to migrate?

This question is mainly interested in the way migration is executed and how its design is implemented after all its issues are circumvented.

This work is concerned with the way task migration is implemented from the moment of migration decision till the safe resumption of migrating tasks at its destination. Algorithms that are incorporated in order to take the decision to migrate a task are not in the scope of this work. As a result, the question of *when to migrate* is excluded from the list of the tackled questions. However, it is worth noting that the system is supposed to migrate tasks from their *hot* source to another tile so as to avoid the formation of hot spots in the system.

Answers of the first two questions are known in the hardware/software co-design stage where spare tiles are determined in an off-line manner. Since all feasible scenarios are tested and simulated, designers know which task is better be migrated when necessary and to where they shall be migrated to. The last question is answered in details in this chapter and the next one, as well.

Distributed architectures are intrinsically decentralized. Its decentralization stems from the fact that each tile runs its own copy of the OS. That is why task migration solution is based on agents spread out on the tiles so as to decide and execute migration.

In order to enable those agents to perform their roles in migration, a middleware layer (called Multi-Processing FrameWork `MProcFW`) is developed to provide necessary APIs that facilitate execution of task migration. Such APIs are invoked by those agents while a migration process is taking place. Being middleware places this layer between OS and applications layers. As a result, this characterizes the solution by portability to different architectures as well as transparency to the applications. More details about these characteristics will be expanded later in this section. The solution lies in two layers: **Application layer** where the agents reside and function, **Middleware layer** where `MProcFW` provides necessary APIs.

5.2 MProcFW layer

MProcFW layer is designed to provide necessary APIs not just for migration purpose but also to enable controlling application(s) tasks. Fig 5.1 shows where MProcFW resides in the SW architecture of the system. MProcFW APIs can be categorized into two categories, they are as follows:

- *Task-related* category that contains all APIs that control tasks by creating, starting, pausing, stopping, resuming or migrating them. These APIs are invoked in ***Application*** layer by application controller task.
- *Communication-related* category that contains all APIs that control communication channels by creating, opening, sending, receiving and closing them. These APIs are invoked by ***Application*** layer.

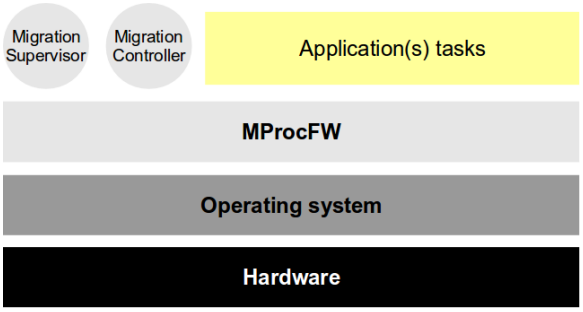

Figure 5.1: MProcFW layer is placed right above OS layer and over which resides applications layer in addition to migration agents.

Controlling applications and implementing states of an overall FSM for the system is possible by invoking necessary MProcFW APIs on its own without the necessity to add other modules and/or layers. The *Task-related* category contains all APIs that create, start or destroy tasks. The *communication-related* category provides APIs that handle inter-task communications (e.g. `open`, `close`, `read`, `write`), these APIs are compatible with Portable Operating System Interface POSIX. Both categories are expanded in Appendix A.

We show in the following, how MProcFW design makes tasks able to be migrated without changing application codes and resolves communication inconsistency issue. This is expanded in the following sections that explain how a migrate-able task is stopped for migration when a migration request is issued and how inter-task communication is migrated too.

5.2.1 Transparent migration points

Our task migration technique uses *migration points* [10, 9, 32, 33] to migrate tasks. A migration point is a predefined point at the code where tasks check if a migration

request exists. A task can be safely stopped and resumed from a migration point. MProcFW specifies a data-flow loop-based task model, i.e. every task in every application has to be compliant with this model.

The task model is shown in Listing 5.1. The INIT procedure is mainly responsible for allocating memory for task state and initializing necessary variables. The initialization is executed once at the startup of every application task. Afterwards, core execution part of a task comes; it is split into individual executions of the FIRE procedure, which is repeatedly invoked. Once an application task is stopped, the FINISH procedure is called for cleanup. Communication is enabled by calling high-level read and write procedures.

```
procedure INIT(TaskStructure *t)   // initialization
  initialize();
end procedure
procedure FIRE(TaskStructure *t)   // execution
  Ch_FIFO->read(buf, size);      // read i/p from fifo
  process();                       // processing data
  Ch_FIFO->write(buf, size);     // write o/p to fifo
end procedure
procedure FINISH(TaskStructure *t) // cleanup
  cleanup();
end procedure
```

Listing 5.1: Data-flow task model. TaskStructure contains data that identifies every task.

Application tasks are supposed to be compatible with such model via coding these procedures only. Then, the model is put into action under an API called MProc_task_bootstrap, its C-like code is shown in Listing 5.2. A task is created by filling a data structure called TaskStructure with its corresponding data like task ID, pointers to INIT, FIRE and FINISH and task state. Then TaskStructure is fed to MProc_task_bootstrap which acts as the task handler so that either OS or system scheduler shall call.

There are commands that can affect tasks. Such commands modify flags that are checked every iteration in MProc_task_bootstrap. One of these checks is the migration point, at which tasks choose which branch they shall continue execution in. In case of migration, the task does nothing waiting for new commands as it is supposed to be destroyed shortly by $MigCtrl_{src}$. *MigCtrl* is responsible for issuing migration requests to tasks.

5.2.2 Overcoming communication inconsistency issue

Communication inconsistency arises from the fact that migrating tasks change their locations in distributed architectures. This makes the resumption of communication between a migrating task and its neighbors impossible without changing these channels. As a result, such channels that connect the migrating task with its neigh-

```
void MProc_task_bootstrap (TaskStructure *t)  {
  t->INIT();                          // INIT procedure
  while (t->cmd != STOP) {
    if (t->cmd == RUN) {
      t->CURRENT_STATE=RUNNING;       // normal exec
      t->FIRE();                      // FIRE procedure
    }
    else if (p->cmd == MIG_REQUEST) {
      t->CURRENT_STATE=MIGRATING;     // migrating
      wait_new_cmd();
    }
  }
  t->FINISH();                        // FINISH procedure
}
```

Listing 5.2: C-like implementation of `MProc_task_bootstrap` that shows the migration point every iteration.

boring tasks have to be changed and in order to resume the migrating task at its destination properly, neighboring tasks have to be informed with the new channels. This change and informing procedures have to be done in a transparent manner to both the task and its neighbors.

Not only is channel change an issue, but also there exists another issue regarding the left unprocessed tokens in FIFOs at the source tile where a migrating task used to run. Without checking if there are left unprocessed tokens after a migrating task is stopped for migration, consistent resumption of communication after migration is impossible due to loss of data. This transfer must be done in a transparent manner to the application and at the same time without requiring special services from communication drivers.

In the following, we show in details how every issue is circumvented in our design. We expand communication channel update issue as well as the problem of the left unprocessed tokens.

5.2.2.1 Communication channel update

Inter-tile communications are undergone through unidirectional FIFO channels through which messages are sent/received by communication drivers, such channels are referred to as *external* channels. Every unidirectional KPN channel has two ends; one is for the input side and the other is for the output side. Every external channel is distinguished by a unique ID. Since a scenario-based design flow is used where application(s) mapping is done statically, all channels IDs are known in the pre-compilation phase. As a result, if a channel is changed due to migration, then a new ID has to be used to create another one connecting the migrating task in its new tile with the same neighboring task.

We introduce the *configurable* channel that is used to enable changing the chan-

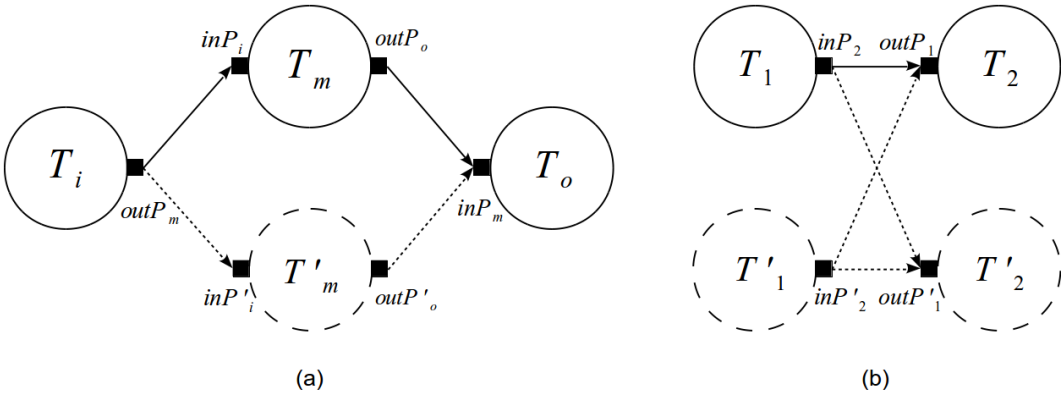

Figure 5.2: Configurable channels. T_m, T_i and T_o are migrating, predecessor and successor tasks, respectively. T'_m refers to the replica. `inp`, `outp` are input and output ports, respectively. Port *inp*: input port $inp_i = inp'_i$ and $outp_o = outp'_o$. (a) T_m is the migrate-able task. (b) Both T_1 and T_2 are migrate-able.

nels during migration transparently to application tasks. A configurable channel, as shown in different forms in Fig 5.2, is a two-branched channel that is composed in reality of two unidirectional channels, i.e. it has three ends and every branch is a unidirectional channel with a unique ID. Only one branch is activated at a time as there is one branch connected to the running version of the migrating task and the other one is connected to its code replica which is created to be a task only due to migration. These two branches that result in three ends are as follows:

i A *configurable output* channel: there is one end at the input side and two ends at the output ones. This is due to the fact that there are two receiving sides only one of them is active and working. In Fig 5.2.(a), a configurable output channel connects T_i from the sending side to both T_m (active) and its replica T'_m (inactive). The dashed arrow (T_i to T'_m) is the alternative branch to the current working on in this configurable channel.

ii A *configurable input* channel: there is one end at the output side and two ends at the input ones. This is due to the fact that there are two sending sides only one of them is active and working. In Fig 5.2.(b), a configurable input channel connects both T_m (active) and its replica T'_m (inactive) on the sending side to T_o. The dashed arrow (T'_m to T_o)is the alternative branch to the current working in this configurable channel.

Changing the channel takes place during migration. The change process is just the switching between the branches transparently to application tasks (migrating and neighbors). We call this changing process *communication channel update*. This update is undergone by the proper migration agent that is responsible for implementing migration process, such agent invokes `MProc_channel_update` function (listed in section A.2) with the proper argument. This function accesses the structure of the corresponding configurable channel where IDs of the branches exist,

closes the current branch, releases its memory (like FIFO buffers) and opens the alternative branch. Port numbers are kept the same in both sides the migrating task side and its neighbor(s) side; therefore, channel change is transparent to applications.

Channels, from spatial perspective, are classified to two types: *external* channel that connects two tasks in two different tiles and *internal* channel that connects two tasks on the same tile via a buffer allocated in the private memory. On one side, *external* channels, on the hardware side, is managed by the communication device while, on the software side, it is managed by communication driver, hence, it has a unique ID to be distinguished. *Internal* channel, on the other side, does not need communication devices; hence, it does not have any ID.

During migration process, the branches inside configurable channels may be switched from one type to another, the possible different scenarios are as follows:

1. Changing an *internal* channel to an *external* one. When a migrating task migrates from the same tile where its neighboring task resides to another tile, therefore, the channel between these tasks changes from internal to external.

2. Changing an *external* channel to an *internal* one. This scenario is the opposite of the previous one, a migrating task migrates to a tile where its neighboring resides, therefore, the channel connecting them changes from external to internal.

3. Changing an *external* channel to an *external* one. This takes place a migrating task migrates from a tile to another, its neighboring task resides in none of them.

Since `MProcFW` layer uses POSIX interface, external channels IDs are used when calling `open` and sometimes `ioctl`, as well, to create the channel and return the file descriptor fd. Every application task uses a port number to communicate with its neighbors in the process network. There is obviously a port number dedicated for communicating with each neighboring task. To application side, port number remains the same even when a migration takes place. This is due to the channel update process takes place only between IDs[1]. IDs are determined in the pre-compilation phase and they are stored in channel structures related to the corresponding channels as there is a structure holding all information about both ordinary channels and configurable ones in the system. Such structure is shown in listing 5.3 written in C.

```
struct config_ch_str{
  bool     cfg;            /*true if configurable*/
  int16_t  ID[N];          /*N: number of branch IDs*/
```

[1]We give all internal channels a single ID equals to -1 to distinguish them from the external ones.

```
  int16_t  tile_indexes[N];/*N: tile indexes*/
  char     ch_name[NAME];  /*channel name*/
  uint16_t crnt_br_id;     /*ID of current branch*/
  int16_t  fd[N];          /*N file descriptors*/
  buffer_t * p_buf;    /*pointer to CB or RB*/
};
```

Listing 5.3: Data structure of a configurable channel.

5.2.2.2 Unprocessed left tokens

The second root of communication inconsistency is the data loss after task migration. The possibility for this data loss stems from the considerable probability of having unprocessed tokens left in one or more of the input buffers of the migrating task. Consequently, resumption of the computation of such migrating task is impossible. Such unconsumed tokens must be transferred to the destination tile and be stored in the newly allocated input FIFO(s) of the migrating task.

The unprocessed tokens have to be transferred to the destination tile without having to modify application code, this is to have eventually a transparent task migration solution. `MProcFW` layer is responsible for allocating input and output FIFOs, hence, `MProcFW` holds control over them and their contents. As a result, we can avoid the modification of communication drivers and implement a protocol that helps circumventing this problem without needing to leave `MProcFW` layer. This is to make the layer self-contained and able to work with POSIX interface without the need of special drivers.

The protocol is called *write-with-copy protocol* and is shown in Fig 5.3, it is applied in reading and writing. On one side, every sender keeps a copy of what it sends in a local buffer called *copy buffer* CB. On the other side, a receiver sends an acknowledgment ACK to the sender for every token consumption. All incoming tokens to a receiver are stored in a local FIFO buffer called *receiving buffer* RB, hence, receiving sides have to consume tokens from RB. Since a copy of unprocessed tokens is still stored in the CBs of predecessors of a migrating task, they are all resent to newly created task at its destination tile. This enables consistent and transparent resumption of communication after migration of tasks state.

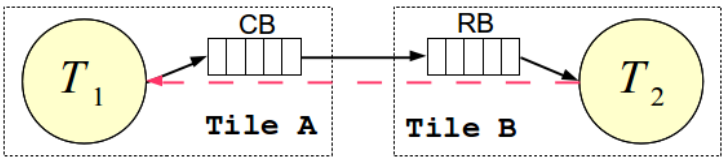

Figure 5.3: *Write-with-copy* protocol. T_1 is the sending side while T_2 is the receiving one.

In the following, we expand the protocol in details implemented in the high level communication primitives `read` and `write`.

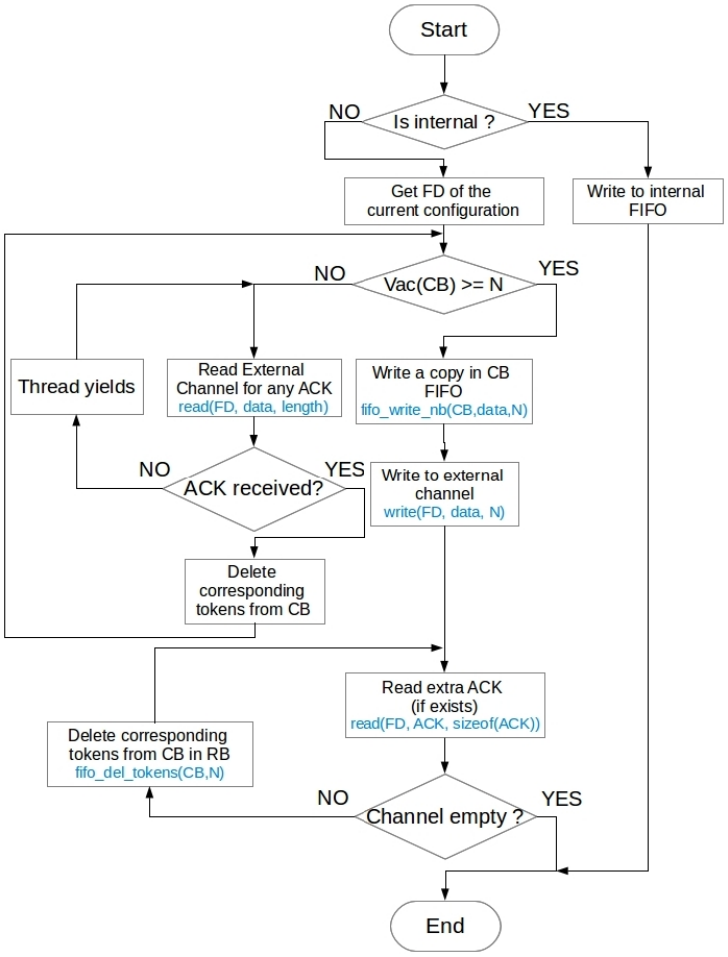

Figure 5.4: `write` flowchart.

Algorithm in `write` function: T_1 is the sending side in Fig 5.3 and exists in Tile A which sends to T_2 in tile B. Every time T_1 invokes `write` to send something to T_2, `write` function runs through the algorithm whose flowchart is shown in Fig 5.4.

After `write` function resolves the port number to the corresponding configurable channel, it starts the algorithm. Firstly, it checks if the channel is internal or external, if it is internal, then token is stored in the FIFO without the need for using communication driver. If it is an external channel then it accesses the corresponding channel structure shown in listing 5.3 to determine the current branch ID and the file descriptor fd. Secondly, it tries directly to write to the external channel but in order for this to be accomplished, a check for vacant locations in CB is performed first to store extra tokens. If there is room in CB, a copy of these tokens are stored

in CB and written to the external channel then another check is performed for ACKs sent from the receiver. If there are any, then the corresponding tokens are deleted from CB in order to have room for future tokens.

When there is no enough room for storing tokens in CB, `write` function listens on the external channel waiting for ACKs and if there is not any, it yields control to the OS scheduler to elect another task to utilize the CPU. This avoids hanging the system waiting for ACKs hindering other tasks from getting executed. This behavior continues until enough room in CB is ready to store extra tokens, this makes the sending side cope with the rate of consumption of the receiving side.

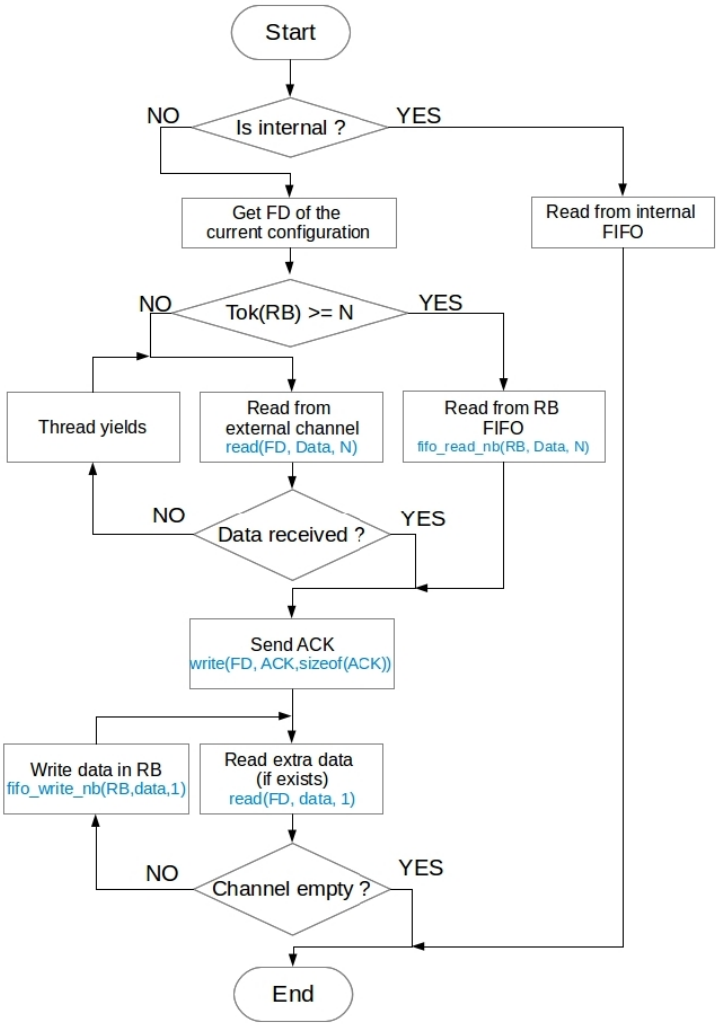

Figure 5.5: `read` function flowchart.

Algorithm in `read` function: T_2 is the receiving side in Fig 5.3 and exists in Tile B which receives tokens from T_1 in tile A. Every time T_2 invokes `write` to

receive input from T_1, `read` function runs through the algorithm whose flowchart is shown in Fig 5.5.

After `read` function resolves the port number to the corresponding configurable channel, it starts its algorithm. Firstly, it checks if the channel is internal or external, if it is internal, then token(s) are read from the FIFO without the need for using communication driver. The function checks RB first if it has unconsumed tokens, they are read and consumed from the buffer and a corresponding number of ACKs are sent to the sending side. Then, a check is made listening to the external channel for new tokens. The newly received tokens by the communication driver need to be stored in RB.

If no new tokens were stored in RB to be consumed directly by `read` function, it listens to the external channel for new tokens. If there is still no tokens received, `read` function yields the control from this task to the scheduler in the OS to elect another task to take control. This takes place until tokens are received from the sending side.

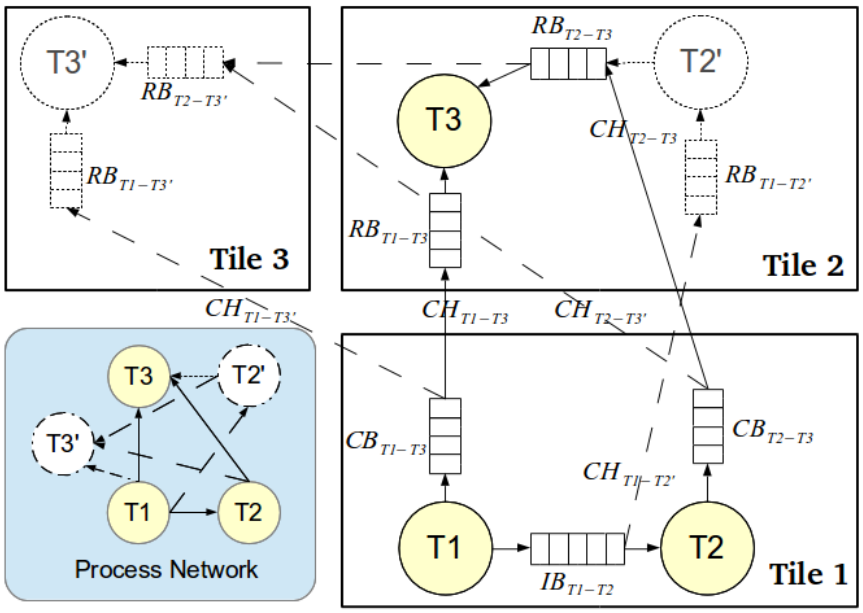

Figure 5.6: On the bottom left corner exists the process network of a small application (3 tasks), the rest of the figure show the mapping of this application on 3 tiles. Both $T2$ and $T3$ are migrate-able and their code replicas reside in tiles 2 and 3, respectively. Continuous line channels are the current working ones while dashed ones work after migration takes place.

5.2.3 Whole example of communication consistency preservation in a migration

In Fig 5.6, a process network of an application containing three tasks {$T1$, $T2$, $T3$} along with their mapping on 3-tile system. There are two migrate-able tasks {$T2$, $T3$} whose replicas reside in `Tile 2` and `Tile 3`, respectively. We use this example to show different configurable channels cases along with the buffers used for write-with-copy protocol that lead to almost all scenarios of switching between the branches of a configurable channel.

Every external channel is denoted by a symbol that represents its unique ID. Every symbol has two parts; the first has the prefix $CH\_$ and the second has an ordered hyphen separated pair of source and destination, e.g. CH_{T2-T3} is the ID of the external channels that connects $T2$ in `Tile 1` with $T3$ in `Tile 2`. Tasks, as well as, buffers in dashed line have not been allocated until there is a migration.

Configurable channels shown in Fig 5.6 are listed in the following:

1. Output channel connecting between $T1$ and {$T2$, $T2'$}. The two branches are:

- Internal buffer IB_{T1-T2} in `Tile 1`.
- External branch $CH_{T1-T2'}$ between `Tile 1` to `Tile 2`.

2. Output channel connecting between $T1$ and $\{T3, T3'\}$. The two branches are:
   - External branch CH_{T1-T3} connecting between `Tile 1` to `Tile 2`.
   - External branch $CH_{T1-T3'}$ connecting between `Tile 1` to `Tile 3`.
3. Input channel connecting $T2$ and $\{T3, T3'\}$. The two branches are:
   - External branch CH_{T2-T3} connecting between `Tile 1` to `Tile 2`.
   - External branch $CH_{T2-T3'}$ connecting between `Tile 1` to `Tile 3`.
4. Input channel connecting $T2'$ and $\{T3, T3'\}$. The two branches are:
   - Internal branch $IB_{T2'-T3}$ in `Tile 2`.
   - External branch $CH_{T2'-T3'}$ connecting between `Tile 2` to `Tile 3`.
5. Input channel connecting between $T3$ and $\{T2, T2'\}$. The two branches are:
   - Internal buffer $IB_{T2'-T3}$ in `Tile 2`.
   - External branch $CH_{T2'-T3'}$ connecting between `Tile 2` to `Tile 3`.
6. Input channel connecting between $T3'$ and $\{T2, T2'\}$. The two branches are:
   - External branch $CH_{T2-T3'}$ connecting between `Tile 1` to `Tile 3`.
   - Internal buffer $IB_{T2'-T3'}$ in `Tile 2`.

Suppose that we study the scenario in which $T2$ migrates from `Tile 1` to `Tile 2` then $T3$ migrates from `Tile 2` to `Tile 3`, all configurable channels have to be updated in all possible cases. Firstly, in order for $T2$ to resume communication with $T1$ properly, external channel $CH_{T1-T2'}$ has to be opened while buffer IB_{T1-T2} is kept to work as CB for $T1$ which is the sending side. This keeps the unconsumed tokens stored and ready to be sent to `Tile 2` when $RB_{T1-T2'}$ is allocated in its memory. Secondly, external channel CH_{T2-T3} is closed and CB_{T2-T3} is freed from the memory without caring about the unconsumed tokens. At `Tile 2`, $RB_{T2'-T3}$ which has a copy of all unconsumed token in CB_{T2-T3} is kept as it is and continues working as an internal channel between $T2'$ and $T3$. In short, $RB_{T2'-T3}$ transforms to internal buffer $IB_{T2'-T3}$.

When $T3$ migrates from `Tile 2` to `Tile 3`, firstly, external channel CH_{T1-T3} is closed and RB_{T1-T3} is freed from memory while the other branch of the configurable channel is the external one $CH_{T1-T3'}$ which is opened, while on `Tile 3` $RB_{T1-T3'}$ is allocated. Before communication is resumed normally between $T1$ and $T3'$, all stored tokens in $CB_{T1-T3'}$ are sent to `Tile 3` and by this token loss is

avoided. Secondly, internal buffer $IB_{T2'-T3}$ acts as $CB_{T2'-T3'}$ keeping all unconsumed tokens for $T2'$ and external channel $CH_{T2'-T3'}$ is opened to connect $T2'$ and $T3'$. RB is allocated in `Tile 3` for $T3'$ which is $RB_{T2'-T3'}$ that is supposed to store all unconsumed tokens once sent from $Tile2$ to $Tile3$ to enable consistent communication resumption.

5.3 Solution agents layer

Number and roles of migration agents are determined by the adopted approach in the system. There are different approaches, they are listed in the following:

Fully distributed approach: agents are all performing the same role which includes both migration decision taking and its execution. This approach fits well the nature of EURETILE architecture, this is considered to be an advantage. However, such approach requires a complex design of the migration agents. Another disadvantage of this approach is that a migration decision can be taken by a number of different agents at the same time, this may lead to stop the system if two adjacent tasks in the same process network migrate at the same time. This is because migration of one task is done without informing agents that migrate its neighboring one, this makes communication inconsistency between them after migration is almost inevitable.

Fully centralized approach: one agent exists for the whole system that is responsible of both taking migration decision and its execution. This avoids the complexity of the fully distributed approach, i.e. development of such central agent is relatively easy on one side. On the other side, such single centralized agent is not able to execute commands on remote tiles like accessing a task state stored in the private memory in a remote tile. This is due to the fact that EURETILE architecture is NORMA.

Semi-distributed approach: two types of agents are devised with different responsibilities. One migration agent is responsible for taking the migration decision and another agent is responsible for executing migration commands. Development complexity of this approach is neither as low like in the full centralized approach nor as high like in the fully distributed one. It fits the decentralized nature of the architecture of EURETILE and also avoids other approaches disadvantages. The design complexity of agents is relatively alleviated, however, there still some complexity in the inter-agent communication needed in order to make all of them working in a coordinated manner to perform the migration.

The third approach is adopted due to its advantages over the other two approaches. It still provides a scalable solution that fits distributed multi-tiled architectures of bigger number of tiles. Because of the fact that agents are working in a semi-distributed hierarchy, there are different responsibilities between them. One

master agent manages a group of slave agents, this makes migration takes place in different levels: decision taking level is different from decision execution level. As a result, there are two types of agents according to the role of each. They are as follows:

i *MigSup* It is migration supervisor which is responsible for taking migration decisions for migrate-able tasks from their source locations to their destinations. It initiates migration via sending commands to source tile of a migrating task and its destination one.

ii *MigCtrl* It is migration controller which is responsible for executing migration commands, sent by *MigSup*, by transferring task state from its source tile to its destination one taking care of communications with the neighbors and resumes the task at its destination. *MigCtrl* is able to execute migration commands on tile due to its ability to access the private main memory.

The whole system is divided into clusters where every cluster is composed of a number of tiles. Migration takes place inside a single cluster. There exists only one *MigSup* per cluster while one *MigCtrl* exists per tile. Not only are the APIs provided by `MProcFW` sufficient for migration but also necessary information about mapping and platform must be provided to *MigSup* and *MigCtrl*. This information exists in the form of tables linked with the agents, otherwise, they cannot know which migrating tasks to migrate or to which destination they shall be migrated to. This is in addition to the information about the neighbors of every migrating task that is necessary for accomplishing migration. The tables are as follows:

i Global View Table GVT contains information about all migrating tasks found in a cluster. GVT is composed of a number of records equal to the number of tiles in the cluster such that each record corresponds to a tile. Inside a record, all migrating tasks residing in the corresponding tile are listed along with their destinations. GVT is the table linked with *MigSup*, hence, only one exists per cluster. An example of GVT is in table 5.1, it is the GVT of the example shown in Fig 5.6. Every record contains beside tile index, the migrate-able tasks and their destination locations like in `Tile 2`, there are two migrate-able tasks $\{T2', T3\}$, however, only $T3$ can be migrated as it is in `Tile 2` while its replica in its destination tile (`Tile 3`) is not running. $T2$ is running on `Tile 1` so it cannot be migrated from `Tile 2`, hence in `Tile 1` record, migration possibility of $T2$ is $TRUE$.

ii Destination Look-up Table DLT contains more details about migrating tasks such as all neighbors' locations, all IDs of channels connecting a migrating task with its neighbor(s) and its destination tile. This table is linked with *MigCtrl*. It enables $MigCtrl_{src}$[2] to know where a migrating task is destined to. This is in addition to be capable of sending commands to right neighbor(s) to stop

[2]It is the *MigCtrl* that resides in the source tile of a migrating task.

Table 5.1: GVT of the example shown in Fig 5.6.

Tile index	Migrate-able task(s)	Destination location	Neighbor(s) location(s)		Migration possibility
			Position[α]	location	
`Tile 1`	$T2$	`Tile 2`	Predecessor	`Tile 1`	TRUE
			Successor	`Tile 2`	
`Tile 2`	$T2'$	`Tile 1`	Predecessor	`Tile 1`	FALSE
			Successor	`Tile 2`	
	$T3$	`Tile 3`	Predecessor	`Tile 1`	TRUE
			Successor	N/A	
`Tile 3`	$T3'$	`Tile 2`	Predecessor	`Tile 1`	FALSE
			Successor	N/A	

[α]A neighbor position is the corresponding relative position in the process network to the migrating task.

Table 5.2: DLT of `Tile 1` example shown in Fig 5.6.

Migrate-able task(s)	Destination location, task ID	Neighbor(s)				
		Relative position	Task ID	Tile ID	Channel ID	Port number
$T2$	`Tile 2`, $T2'$	Predecessor	$T1$	`Tile 1`	IB_{T1-T2}	0
		Successor	$T3$	`Tile 2`	CH_{T2-T3}	1

and/or resume communication during and after migration. An example of DLT is in table 5.2, it is the DLT of `Tile 1` in the example shown in Fig 5.6. The difference between DLT and GVT can be best shown in the difference between `Tile 1` record in the GVT (table 5.1) and in the DLT of the same tile. In DLT, much information is supplied about the channels connecting $T2$ and its neighboring tasks in application process network and their IDs.

In a typical migration case, *MigSup* takes a decision of migrating one of migrate-able tasks in some tile. It inquires its GVT to retrieve information about its destination tile. Then it sends commands to both $MigCtrl_{dest}$[3] and $MigCtrl_{src}$ MIG_IN and MIG_OUT, respectively. MIG_IN command makes $MigCtrl_{dest}$[4] prepares for receiving the task state, this is by setting up necessary communication channels, creating the task and waiting for task state that is sent by $MigCtrl_{src}$ so as to resume execution of the migrating task. $MigCtrl_{src}$, in turn, suspends neighbors of the migrating task to stop communications temporarily, stops the migrating task, collects its state and sends it to its destination. Then it resumes the neighbors after the reception and loading of tasks state on destination side. In order to achieve that `MProcFW` comes into play.

[3]It is the *MigCtrl* that exists in the destination tile of a migrating task.
[4]It is the migration controller that resides in the destination tile of a migrating task.

Communications take place either among *MigCtrls* or between *MigSup* and *MigCtrls*. That is why channels must exist between the agents. A star topology is adopted where channels exist only between all *MigCtrls* and *MigSup*. *MigSup* initiates migration by sending to corresponding *MigCtrl*, then it acts as a router to redirect commands and messages sent by one *MigCtrl* to another, this is due to the lack of direct channels between *MigCtrls*. *MigSup* starts routing after initiating migration and returns back to its initial state when it receives MIG_OUT_DONE[5] notification.

5.4 Migration principle

Migration algorithm is designed so that task migration functionality is split into two parts:

i Task migration decision taking part.

ii Task migration execution part.

The decision part is the one where migration decision is taken in *MigSup*, however, neither the algorithm of taking such decision nor its input(s) upon which the decision is taken is not in the scope of this work. The responsibility of the execution of the migration is the one of *MigCtrl*. *MigSup* initiates migration by triggering corresponding agents ($MigCtrl_{dest}$ and $MigCtrl_{src}$) to start executing migration process. In this section, the connection of the agents in the system and migration algorithm is expanded in details. We address how the design circumvents issues like the blockage that might occur during migration takes place due to the transient interruption in the execution of some tasks in the application process network.

5.5 Agents connection

The connection between *MigSup* and *MigCtrls* is depicted in Fig 5.7, between *MigSup* and *MigCtrl* two KPN channels exist in both directions, both are depicted as one bidirectional channel for brevity. Communication is inevitable between *MigCtrls* as will be expanded in the algorithm details yet it is intended not to use any channels between *MigCtrls*. This is chosen so as to enable the design to be scalable since the inter-*MigCtrls* communication takes place between via dedicated channels. Their number will get higher when a cluster contains more and more tiles. That is why *MigSup* initiates migration process and then works as a router to route messages from one *MigCtrl* to another preventing bloating the number of channels when more tiles exist in a cluster. The gain due to reduction of number of channels amortizes the extra delay in communication introduced by this technique. This gain of this choice can be shown as follows:

[5]MIG_OUT_DONE is a notification sent by $MigCtrl_{src}$ to *MigSup* to saying that migration has been completed. *MigSup* consequently sends to all *MigCtrls* to update their tables.

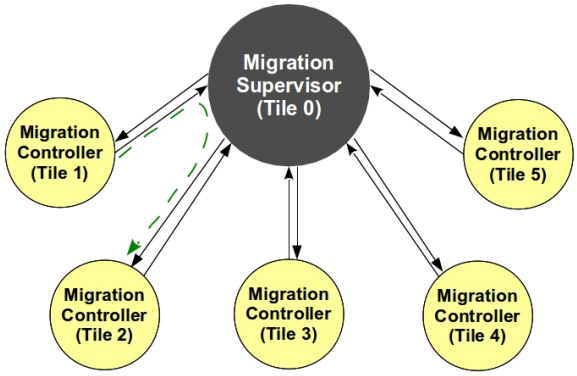

Figure 5.7: Communication channels between *MigSup* and *MigCtrls*, dashed arrow shows an example of inter-*MigCtrls* communication that passes through *MigSup*.

- If every pair of *MigCtrls* are connected by two unidirectional channels, number of channels equals $2\binom{n}{2} + 2n = n^2 + n$ channels, where n is the number of *MigCtrls*. Example: in Fig 5.7, $n = 5$ number of channels for inter-*MigCtrls* communications equals $2\binom{5}{2} = 20$ while number of channels connecting *MigCtrls* and *MigSup* equals 10 channels making total number of 30 channels.
- Using *MigSup* as a router, number of channels reduces to be linearly dependent on the number of *MigCtrls* and equals to $2n$ channels. Example: in Fig 5.7, just 10 channels are required for connection between *MigCtrls* and *MigSup*.

5.6 Blockage avoidance

Task migration process implies the pause of the neighboring tasks to the migrating task in order to stop temporarily the input flow of tokens to the migrating task. Such pausing interrupts the execution of the application tasks which might result in blockage if not planned in the right manner. If the `read` function that is called to receive tokens is synchronous[6] which is the case in KPN, the order of pausing the tasks can cause total halt in the system. In order to investigate that, we show in Fig 5.9 a typical process network or a part of it and we study it. T_m is the migrating task while T_i and T_o are its predecessor(s) and successor(s), respectively. T_i is one or more tasks that produce the input tokens of T_m. T_o is like T_i but succeed T_m in the process network.

If the order of pausing the tasks starts with pausing T_i task(s) then issues migration request to T_m then pauses T_0, T_m might hang waiting for tokens from T_i, and hence, `fire` procedure (shown in Fig 5.8) never reaches its end of execution. As a result, the pausing order is performed in counter direction of the token dependence,

[6]It means that task halts waiting to receive tokens.

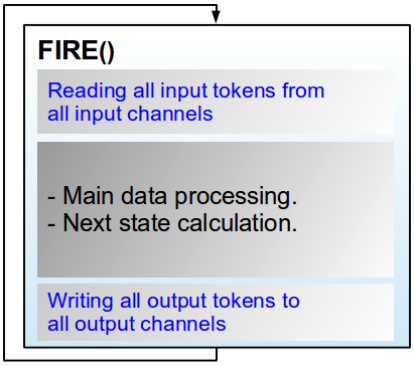

Figure 5.8: `fire` function consumes all input tokens from all channels, processes the data and finally sends out the output ones every iteration.

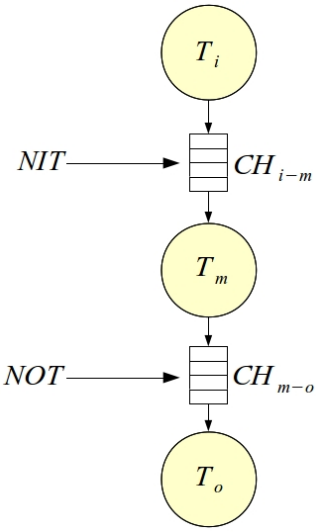

Figure 5.9: T_m is the migrating task. T_i is the set of tasks that produce input tokens to T_m. T_o is the set of tasks that consume output tokens from T_m. $CH_{src-dst}$ is a unidirectional channel that connects T_{src} and T_{dst}. NIT: Number of Input Tokens, NOT: Number of Output Tokens.

so T_o depends in consuming tokens on T_m and same applies to T_m but with respect to T_i. In order to avoid blockage, T_o are first paused then a migration request is issued to T_m then T_i is paused. After migration whatever the order of resumption is does not impact the consistency of the execution of the application.

5.7 Migration algorithm

In this section, the algorithm of the migration process is expanded. The sequence of commands and actions starting right after migration decision taking is explained showing the interaction between agents (*MigSup* and *MigCtrl*). Commands are classified into two types depending on their producers, they are as follows:

1. CONTROL commands: they are the commands that are sent by *MigSup* to *MigCtrl*, they are initiated by *MigSup* so they never require to be routed.

2. **TRANSFERRED** commands: they are the commands that are created by *MigCtrl* and sent to another *MigCtrl*, they are most probably created by $MigCtrl_{src}$ to either $MigCtrl_{dest}$ or other *MigCtrls* which control tiles containing neighboring tasks to T_m. As a result these commands always require to be routed as they are sent firstly to *MigSup* that routs them to their destinations.

CONTROL commands are listed as follows:

- *MIG-IN*, it is the command that informs $MigCtrl_{dest}$[7] that a migration is occurring and it is in the process of receiving the task state of a migrating task in this tile. This makes $MigCtrl_{dest}$ create the task by activating its code replica that resides in the memory.

- *MIG-OUT*, it is the command that informs $MigCtrl_{src}$[8] to start sending the migrating task with the index that is included in the arguments of the command from its source tile to its destination one.

- *UPDATE-DLT*, it is the command that follows every migration process that holds the updated information of the current locations for the migrating task, it is sent to every *MigCtrl*.

TRANSFERRED commands are listed as follows:

i PAUSE-UPDATE: the aim of this command is to pause a neighboring task and update the configurable channel connected between it and the migrating task, i.e. switch the channel to the other branch.

ii SYNCH-RESUME: this command is sent only to predecessors of the migrating task in the task graph to synchronize the contents in their CB with RB of the migrating task at its destination before its resumption. This is meant so as to preserve communication consistency.

iii RESUME: this command is sent to successors of migrating task in the task graph to resume them after migrating task state successfully sent to its destination.

iv LOAD-STATE: this command is sent to the destination of a migrating task so as to receive the state sent from the source.

When migration decision is taken, *MigSup* looks into GVT to determine both source and destination *MigCtrls*. Then firstly, it sends to $MigCtrl_{dest}$ *MIG_IN*, secondly, it sends to $MigCtrl_{src}$ *MIG-OUT*, thirdly, *MigSup* starts working as a

[7]The *MigCtrl* that resides in the destination tile of a migrating task.
[8]The *MigCtrl* that resides in the source tile of a migrating task.

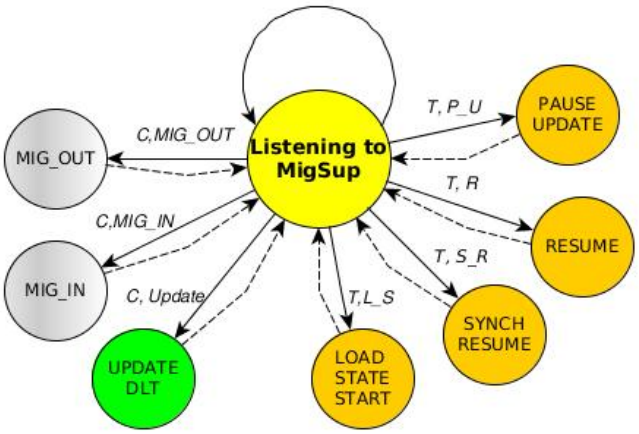

Figure 5.10: FSM of Migration controller. C: Control command, T: Transferred command

router until it receives a *MIG_OUT_DONE* message from the $MigCtrl_{src}$. Once a *MigCtrl* receives a command whether it is a CONTROL or TRANSFERRED, it changes its state of listening to a state that is corresponding to every command in order to execute the corresponding functionality, this is shown in Fig 5.10.

Fig 5.11 shows simplified sequence of migration algorithm, it is made in a brief way reducing implementation details to show how it works in theory. It starts right after taking the decision by the *MigSup*. *MigSup* retrieves the correct data from the GVT and determines its destination and locations of its neighbors T_i and T_o. Then it starts the same sequence of control commands as explained before. For the sake of brevity and clearness, we use $MigCtrl_{neighbor}$ to represent any *MigCtrls* that control one or more neighboring task to T_m in this example.

After *MigSup* issues its control commands and sends them to $MigCtrl_{src}$ and $MigCtrl_{dest}$. $MigCtrl_{src}$ starts the migration by pausing T_o by sending PAUSE-UPDATE command to $MigCtrl_{neighbor}$ via *MigSup* to pause T_o task(s) and update the channels connecting them with T_m. After pausing all T_o task(s), $MigCtrl_{src}$ send LOAD-STATE command accompanied by the state of the migrating task to $MigCtrl_{dest}$ so as to make the latter able to resume the migrating task on its tile. Once $MigCtrl_{dest}$ finishes the resumption of T_m, it sends a MIG_IN_DONE acknowledgement to $MigCtrl_{src}$, this is to enable $MigCtrl_{src}$ to start pausing T_i task(s) and updating all the channels that connect them with T_m. Afterwards, $MigCtrl_{src}$ sends to $MigCtrl_{neighbor}$ SYNCH_RESUME to synchronize CB and RB buffers, this is where, write-with-copy protocol (explained in 5.2.2.2) comes into play, this synchronization takes place only with T_i task(s) that resend all stored tokens in their CBs to T'_m in its destination tile. Once synchronization is finished, T_i are ready to resume their execution normally.

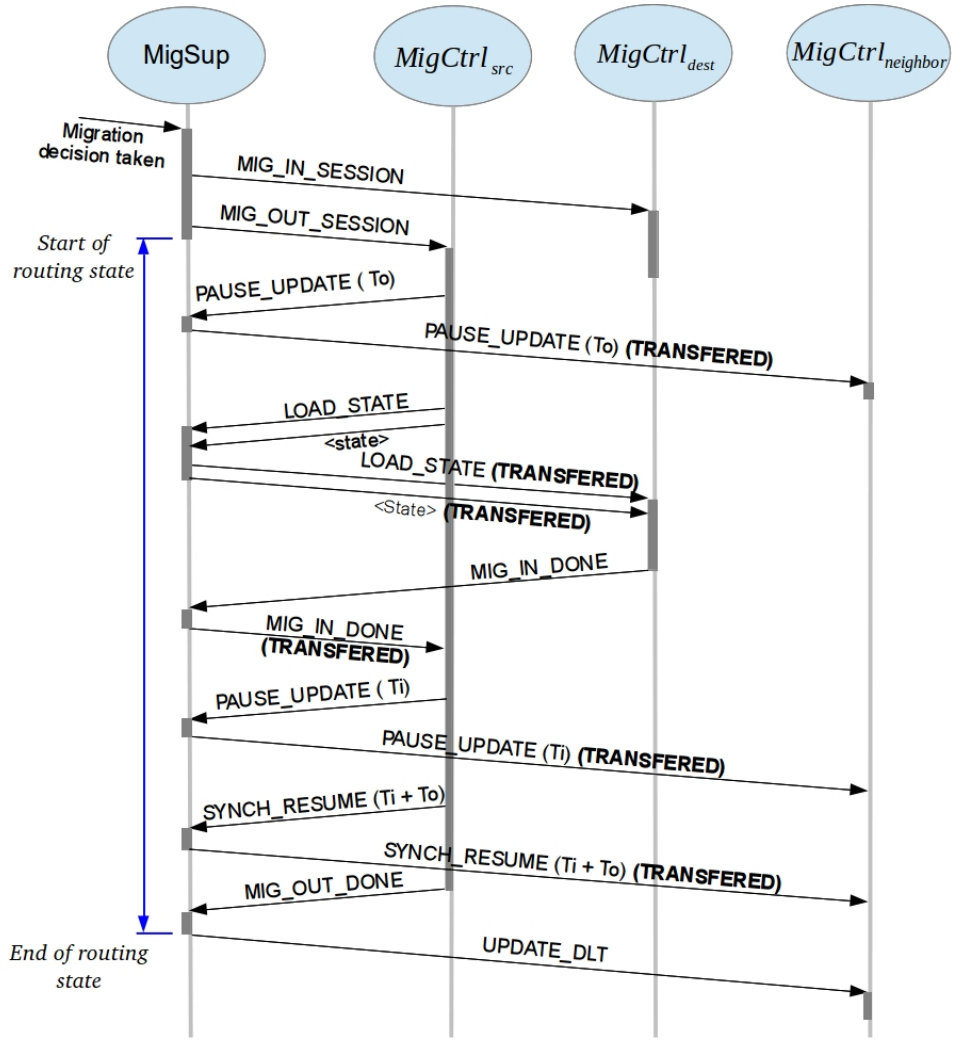

Figure 5.11: Migration sequence diagram

5.8 *MigSup* routing algorithm

One of *MigSup* states is the routing state where TRANSFERRED commands are routed. In order to have a functional routing, a unified header is devised. This unified header can accommodate different commands by including its ID and have their arguments along with its payload that is most probably is following command header. This header is shown in Fig 5.12.

When *MigCtrl* wants to send a command to another *MigCtrl*, the former puts the destination tile index in the command header. Once *MigSup* receives the command, it checks the command type CONTROL or TRANSFERRED and redirects the command to its destination tile when it finds command type TRANSFERED. Sometimes commands are followed by chunks of data like in sending task state, in order to redirect these chunks correctly *MigSup* detects that by checking payload

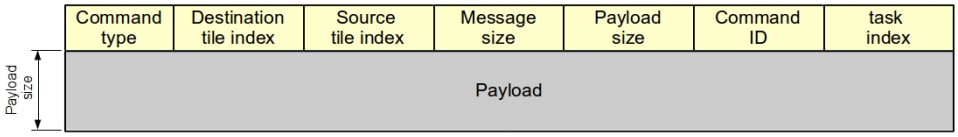

Figure 5.12: Unified header command with is payload.

size part in the header, when payload size is bigger than zero, this implies that payload shall follow the received command.

Chapter 6

Automatic generation and experimental results

Contents

In this chapter, we show how task migration solution (`MProcFW` + migration agents) is inserted in the software and automatically generated for the multi-tiled system EURETILE. Then results of the experiments that are performed on task migration are shown. Firstly, we show the platforms on which experiments are running, there are two platforms where one of them is an ARM-based simulation platform while the other one is an x86-based real hardware platform. Secondly, we expand the existing design flow and the tool-chain for automatic generation. Thirdly, we show how the task migration solution is inserted in the tool-chain in order to eventually synthesize software with task migration capability. Finally, we expand the results of the experiments that are run on both platforms. we evaluate the overhead both in terms of code size and execution time, we give estimation functions in order to predict the cost of migration to certain accuracy.

6.1 Platforms

6.1.1 Simulation platform

The simulation platform used for our experiments is developed in systemc TLM. It is composed of all components for a system such as peripherals (TTY, display ... etc.). A QEMU [44] emulator wrapper is developed and is included in every tile as a systemc component. This simulation platform developed in TIMA laboratory is detailed in [45]. A systemc TLM model for network device has been integrated in the platform. All components in a tile are connected by an abstract network-on-chip developed in systemc. Every tile has a processor component of type ARM cortexA9 architecture with its private memory.

This simulation platform is characterized by scalability so that tiles can be instantiated before every simulation, it can instantiate from 2 tiles to 128 tiles. Number and topology of tiles are specified and with the use of scripts, the platform is instantiated and associated. Each tile runs its own copy of the OS.

6.1.2 Hardware platform

The hardware platform is an Intel based multi-tiled fault-aware platform. It is called lattice QUantum chromodynamic ON Gpu QUonG, presented in [46]. It aims to deploy a GPU-accelerated HPC hardware platform mainly devoted to theoretical physics computations. The objective will be reached catalyzing the efforts of a community of physics researchers, delivering the applications computing requirements and executing platform benchmark, in collaboration with computer scientists and hardware and software experts in charge of design and develop the system itself. The QUonG project will deliver a complete reference hardware platform tailored for scientific computations leveraging on commodity devices (computing and control part) and on a dedicated custom interconnection system APEnet+ [47] for inter-node communication network.

The Hardware platform has 16 tiles, each has a Xeon SMP processor, private memory and a network device. All components inside a tile are connected by PCIe[1]. The platform rack, in reality, is shown in Fig 6.1. A server is installed and used in order to facilitate remote connection and software loading to different tiles. Since the platform is developed in Rome, Italy. As a result, remote connection is inevitable.

[1]Peripheral Component Interconnect Express, it is a computer high-speed serial bus. It is the successor of PCI and AGP.

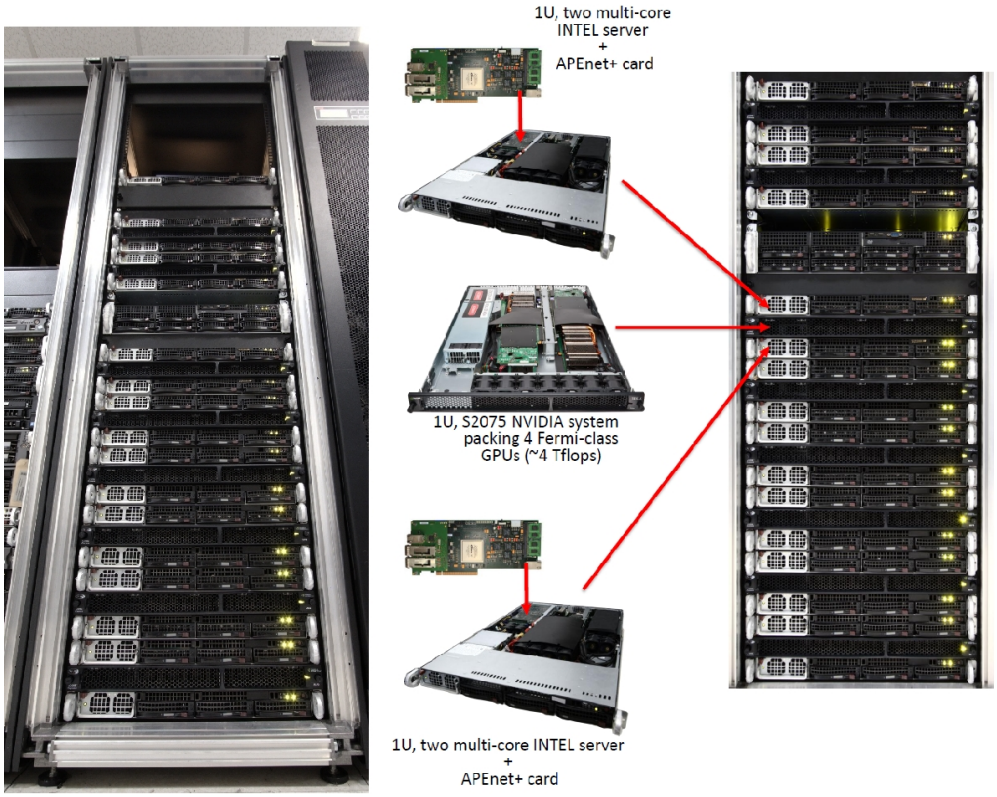

Figure 6.1: Hardware platform.

6.2 Operating System

We use a μkernel embedded OS coded in C99 called DNA-OS developed in TIMA and is presented in [14]. It has been chosen because of its small memory footprint, simplicity, small overhead and its layered structure. The base communication is run through virtual file system VFS. The OS runs single threaded tasks. It incorporates co-operative scheduling where a task has to yield in order for other tasks to take control, this is by explicit corresponding call of OS API.

It is capable of supporting SMP as well as uni-processors. `Libc` is widely supported. However, neither MMU nor dynamic loading is supported as It targets embedded systems domain. Its kernel follows the exokernel architecture [48]. It has been ported on x86 for high performance computing. As a result, DNA-OS is used as OS in the EURETILE software stack.

6.3 Design flow

The design flow developed in the context of the EURETILE project is based on DAL presented in section 3.6.2. It is designed to map multiple streaming parallel

applications on multi-tiled MPSoCs. It is a scenario-based design flow that supports design, optimization, and simultaneous execution of multiple applications targeting heterogeneous many-core systems. Applications are specified as KPNs. KPNs are suitable for a general description of a high-level design flow as they are determinate, provide asynchronous execution, and are capable to describe data-dependent behavior. In case a higher predictability is required, the application model can be restricted, e.g., to synchronous data flow SDF graphs [49]. To coordinate the execution of different applications, we use a FSM, where each scenario is represented by a state. Transitions between scenarios are triggered by behavioral events generated by either running applications or the run-time system.

Since this design flow is meant for embedded systems where number of scenarios is restricted, hence, scenarios are known in design time. The proposed design flow has to provide three key features to the system architect, they are as follows:

1. A high-level specification model that hides unnecessary implementation details but provides enough flexibility to specify dynamic interactions between applications.

2. An optimal mapping of the application onto the architecture in a transparent manner.

3. Run-time support to dynamically change the workload of the system.

Mappings are calculated off-line. Later, at run-time, the run-time manager monitors behavioral events, and applies the pre-calculated mappings to start, stop, resume, and pause applications according to the FSM. As the number of scenarios is restricted, an optimal mapping could be calculated for each scenario. However, assigning each scenario a different mapping might lead to bad performance due to reconfiguration overhead. Therefore, processes are assumed resident, i.e., an application has the same mapping in two connected scenarios. The result of this approach is a scalable mapping solution where each application has assigned a set of mappings that are individually valid for a subset of scenarios. The run-time manager is made up of hierarchically organized controllers that follow the architectural structure and handle the behavioral and architectural dynamism. In particular, behavioral dynamism leads to transitions between scenarios, and architectural dynamism is caused by temporary or permanent failures of the platform. The controllers monitor the behavioral events, change the current scenario, and start, stop, resume, or pause certain applications. Whenever they start an application, they select the mapping assigned to the new scenario. To handle failures of the platform, spare cores are allocated at design-time so that the run-time manager has the ability to move all processes assigned to a faulty physical core to a spare core. As no additional design-time analysis is necessary, the approach leads to a high responsiveness to faults.

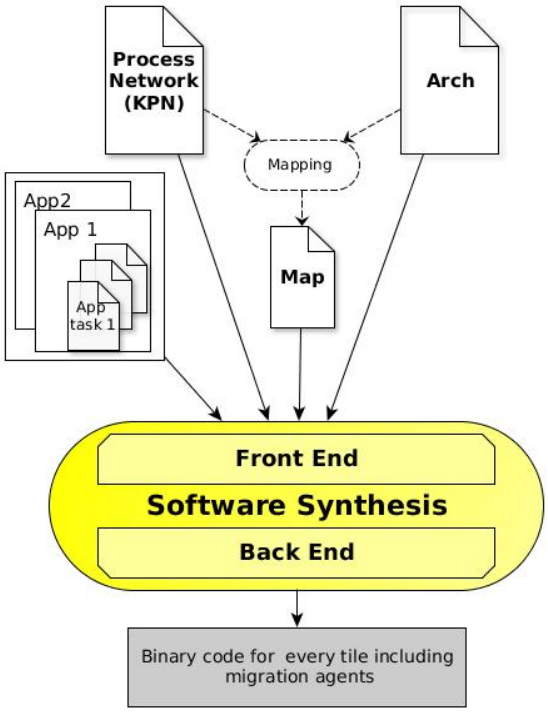

Figure 6.2: Whole design and synthesis flow. The feedback part in Y-chart flow is removed for the sake of brevity.

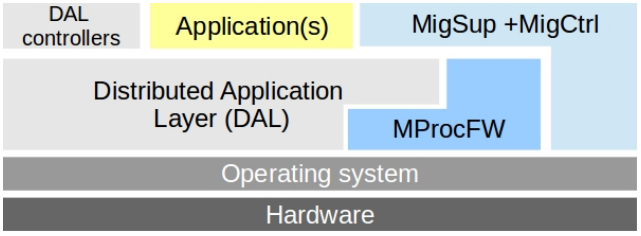

Figure 6.3: Layers of software `MProcFW`.

The design flow adopts Y-chart paradigm and is shown in Fig 6.2. It is a scenario-based one where all execution scenarios are combined into a global Finite State Machine FSM, the states of the latter describe the states of all applications. This design-flow supports design, optimization, and simultaneous execution of multiple distributed data-flow applications.

All execution scenarios are then checked via simulation so as to determine mapping according to quality of service. Although details of mapping phase are not given since it is out of the scope of this work, it is worth noting that spare tiles are chosen to be dedicated for migrating tasks in case of thermal spots. As a result, task code replicas exist at the predefined destinations and linked with the code. Those replicas are not executed until migration takes place.

`MProcFW` is integrated with DAL programming environment as depicted in Fig 6.3. Not a very big development effort has to be exerted in order to accomplish such integration. This effort can be summarized in the following:

1 Task model shown in Listing 5.1 in the previous chapter is adopted. This model (called `MProc_task_bootstrap`) forms a unified task handler that can run whatever application task. This handler argument is a unified data structure `TaskStructure` which contains pointers to functions `init, fire,` and `finish`.

2 Since there will be two kinds of channels: regular and configurable ones. High level procedures `read` and `write` that are found in Listing 5.1 shall differentiate between them and call the right corresponding API either from DAL or from `MProcFW`. That is because *write-with-copy protocol* is only applicable with configurable channels. As a result part of `MProcFW` layer is under DAL layer.

Although mapping phase is not in the scope of the work depicted in dashed lines in Fig 6.2, it is important to clarify the inputs and outputs of this phase. Basically, application(s) tasks graphs along with description of platform architecture are the inputs to mapping phase.

In order to clarify the flow, we show input and output files of a practical example. A 4 tile platform is used to house a small application, it is composed of three tasks: generator (`gen`), square (`sq`) and consumer (`cons`). `Gen` generates numbers in a predefined range. These numbers are sent to `sq` to be squared, `sq` has to keep track of the indexes of the inputs, hence, it is not a state-less task. Finally, `cons` displays the result of `sq`. The task graph is depicted in Fig 6.4 showing that `sq` is migrate-able and configurable channels connecting it with its neighbors.

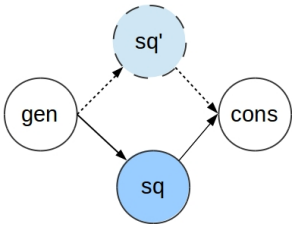

Figure 6.4: Process network
sq: migrate-able, sq': replica.

First input to mapping phase is the application(s) task-graphs files which are described in Extensible Markup Language XML as shown in Listing 6.1. The process network shown is that of the task graph depicted in Fig 6.4. It describes every process by itself along with its connecting channels. It is worth noting the following:

i Tasks that can be migrated are specified by assigning *"mig"* value to *type* attribute like in the case of `sq` task. Ordinary tasks that needn't be migrated are left with *"io"*.

```
<processnetwork>
  <process name="gen" type="io">
    <port type="output" name="1"/>
    <src type="c" location="gen.c"/>
    <src type="h" location="gen.h"/>
  </process>
  <process name="sq" type="mig">
    <port type="input" name="1"/>
    <port type="output" name="2"/>
    <src type="c" location="sq.c"/>
    <src type="h" location="sq.h"/>
  </process>
  <process name="cons" type="io">
    <port type="input" name="1"/>
    <src type="c" location="cons.c"/>
    <src type="h" location="cons.h"/>
  </process>
  <channel name="ch_gen_sq">
    <send process="generator" port="1"/>
    <rec process="square" port="1"/>
  </channel>
  <channel name="ch_sq_cons">
    <send process="square" port="2"/>
    <rec process="consumer" port="1"/>
  </channel>
</processnetwork>
```

Listing 6.1: Task graph description of the application whose process network is shown in Fig 6.4.

ii Configurable channels are not explicitly described. They have to be calculated by the SW synthesis tool latter knowing the migrating task and the spare tile(s). This is because Task migration insertion does not impact the design phase like the SW synthesis phase.

The second input to mapping process is the description of the 4 tile platform is shown in Listing 6.2. The tiles are connected by 3D torus NoC, tiles are found on the Z-axis making $1 \times 1 \times 4$ platform. Spare tile is chosen to be `tile_3`, this is by putting a non-zero value in `substitute` attributed, for example, *substitute="1"* in Listing 6.2. The whole 4 tiles are assembled in one cluster *cluster_0*.

The output of mapping phase is an XML file that describes where each tile shall run on which tile, it is shown in Listing 6.3. This does not include the replicas of migrating tasks, replicas placement are calculated latter by SW synthesis tool latter.

All these mentioned inputs along with the output of mapping phase and application(s) codes written in C language are all inputs to the SW synthesis tool as expanded in the following section.

```
<network x="1" y="1" z="4"/>
  <tile name="tile_0" id="0" substitute="0">
    <location x="0" y="0" z="0"/>
    <port name="port0" />
  </tile>
  <tile name="tile_1" id="1" substitute="0">
    <location x="0" y="0" z="1"/>
    <port name="port0" />
  </tile>
  <tile name="tile_2" id="2" substitute="0">
    <location x="0" y="0" z="2"/>
    <port name="port0" />
  </tile>
  <tile name="tile_3" id="3" substitute="1">
    <location x="0" y="0" z="3"/>
    <port name="port0" />
  </tile>
  <shared name="cluster_0">
    <port name="port0" />
    <port name="port1" />
    <port name="port2" />
    <port name="port3" />
  </shared>
</network>
```

Listing 6.2: XML description of $1 \times 1 \times 4$ Platform.

```
<mapping>
  <binding name="gen">
    <process name="gen" />
    <processor name="tile_1" />
  </binding>
  <binding name="cons">
    <process name="cons" />
    <processor name="tile_1" />
  </binding>
  <binding name="sq">
    <process name="sq" />
    <processor name="tile_2" />
  </binding>
</mapping>
```

Listing 6.3: The mapping.

6.4 SW synthesis tool

A two-stage tool chain is developed to perform the software synthesis phase. The software synthesis input consists of the application models, the code of the tasks, the mapping, and the architecture specification. The role of the synthesis tool is then to transform the input into binary code for platforms used. Two platforms are used: ARM based simulation platform and x86 based real hardware one. The tool is depicted in Fig 6.5. The tool is split into two parts, namely front-end and back-end. The front-end transforms parallel data-flow applications into multiple threads in C code. The back-end compiles and links the obtained code from the front-end on top of the OS and generates binary code for the target platform. In the following, the roles of both tool ends are expanded.

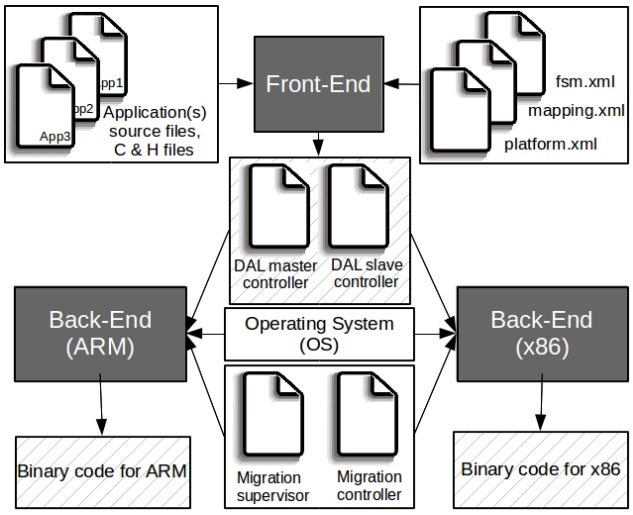

Figure 6.5: Software synthesis tool with its input and outputs for both platforms. The OS used is DNA-OS.

6.4.1 Front-end

After mapping phase is finished, we have the following input files for the front-end of the tool, as shown in Fig 6.5:

i Platform architecture file contains information about the system. It lists number of tiles, cluster(s) and spare tile(s). The file is called `platform.xml`.

ii The result of mapping phase. Mapping is described in a XML file called `mapping.xml`.

iii All the possible scenarios of the system and their FSM are described in a XML file called `fsm.xml`.

iv Application(s) codes with their process networks.

After parsing applications process networks, mapping, and FSM, the front-end part generates C codes for DAL controllers. It modifies application codes without changing any of the code that is responsible of functionality. The modification in applications codes is limited in mingling tasks names with their corresponding tile as well as names of their C files to be able to be instantiated several times. It, also, generates necessary code for FSM and mapping. This part of the tool, as well as, the back-end part is developed in Ruby. On migration side, both *MigSup* and *MigCtrl* codes are fixed, i.e. they need just to be placed in the correct tile.

6.4.2 Back-end

The main responsibility of the back-end is to provide two different tool chains targeting x86 or ARM architecture. However it still takes platform architecture,

task graphs (process networks) and mapping output. This is to generate GVT and DLTs for *MigSup* and *MigCtrls*, respectively. It generates source codes for each tile, a bootstrap file for each tile, and some specific files required to build the final binaries such as a compilation and linking script for each tile. The main role of the bootstrap file (`main.c`) is as follows:

1. allocates the threads
2. creates the channels
3. links the communication ports of every task to its corresponding channel.
4. starts the controllers on its corresponding tile.

Since this part of the tool chain supports two target platforms, it has specific parts for every target including the Hardware Abstraction Layer HAL and the interfaces to interact with the platform specific compilers and linkers.

MigSup and *MigCtrls* follow the same task model in Listing 5.1. Back-end part generates two functions in separate files holding the same names: `init_GVT()` and `init_DLT()`, these functions are invoked once in `INIT` procedure in *MigSup* and *MigCtrl*, respectively.

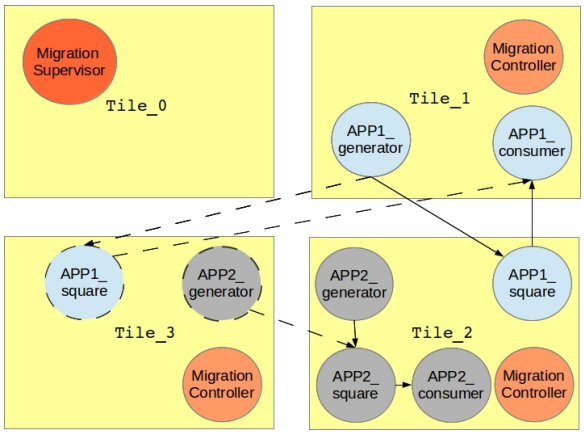

Figure 6.6: Mapping of the agents and applications on all four tiles.

6.5 First experiments

In this section, we show the first experiments that are considered to be a proof-of-concept of task migration. We illustrate the functionality of the task migration by real screen shots that show different steps of migration. We show such experiments on both platforms expanding the procedures performed in case of the HW one. Same mapping example shown in Fig 6.6 is used for both experiments undergone on both platforms. The three tasks are as follows: `generator (App1_generator)`,

`square (App1_square)`, and `consumer (App1_consumer)`. `App1_square` task is migrate-able, the code of its replica is changed to be multiplying by ten. This is to have a clear display of the effect of migration when it takes place.

Since the system is composed of tiles that are connected by 3D torus network-on-chip, tiles are designated by their Cartesian coordinates, this is in the following manner: `Tile n` shall be referred to as *tile*_*X*_*Y*_*Z* All four tiles are on Z-axis; therefore, tiles which are shown in fig 6.6 are designated as follows:

i `Tile 0` is referred to as *tile*_0_0_0

ii `Tile 1` is referred to as *tile*_0_0_1

iii `Tile 2` is referred to as *tile*_0_0_2

iv `Tile 3` is referred to as *tile*_0_0_3

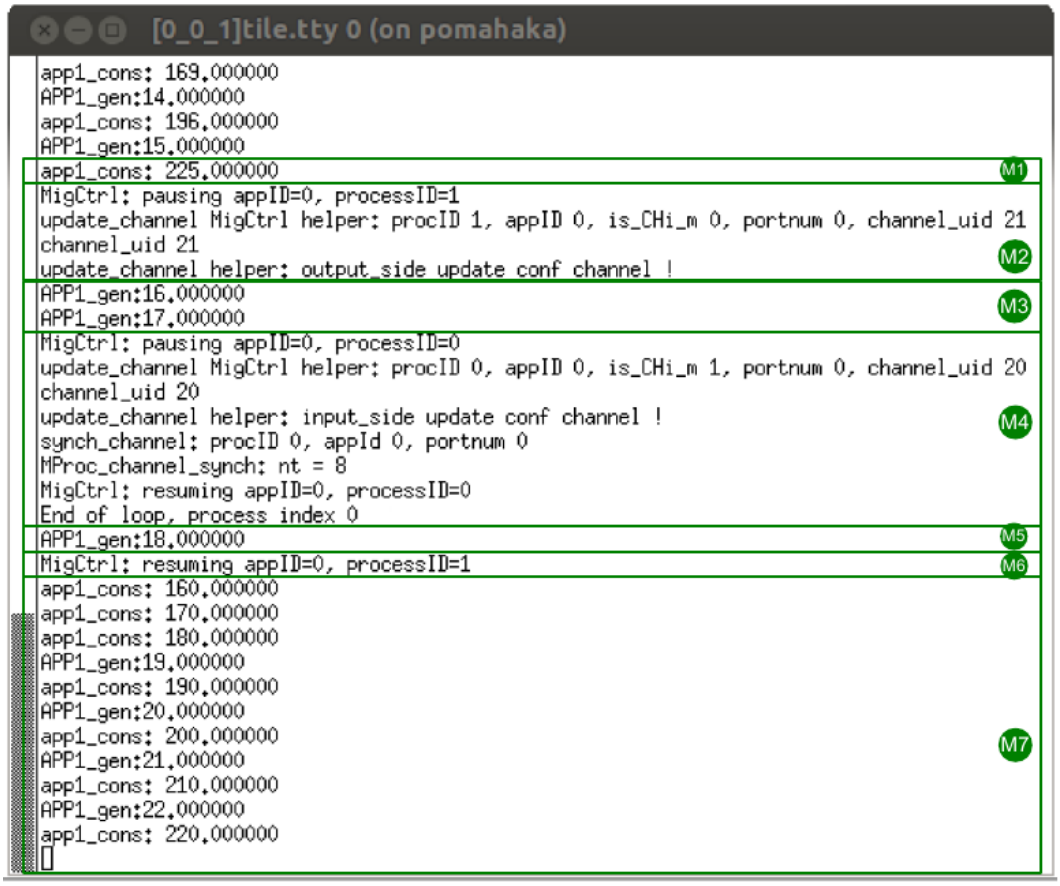

Figure 6.7: Screen shot of the output of *tile*_0_0_1.

6.5.1 Simulation

We study only the first migration that takes place during the execution of application 1. At an arbitrary point in time, `App1_square` migrates to *tile*_0_0_3 where there exists its code replica. The sole difference in the screen output between the one of simulation and the one of the real HW is the debug mode that is activated in simulation only. As a result, Log messages are displayed on the screen in addition to the regular output of tasks `App1_consumer` (always starts with `app1_cons`) and `App1_generator` (always starts with `App1_gen`), they show, in more details, the steps that take place in the corresponding tile. Fig 6.7 depicts the output of `App1_consumer` at *tile*_0_0_1 that records the period of time that starts almost with last moment in the execution by `App1_square` at its source (*tile*_0_0_2) till its resumption at its destination (*tile*_0_0_3).

There are seven migration marks that are added to show different steps in such period. Before explaining those marks, we recall how migration starts according to its algorithm. In order for `App1_square` to migrate, several different actions have to be taken by migration agents beforehand. *MigSup* sends MIG_IN and MIG_OUT to $MigCtrl_3$ and $MigCtrl_2$, respectively. Once $MigCtrl_2$ receives MIG_OUT, it pauses `App1_consumer`, issues migration request to `App1_square`, collect and send task state to $MigCtrl_3$ at *tile*_0_0_3, then pauses `App1_generator`. Afterwards, it sends SYNCH_RESUME command to $MigCtrl_1$ to synchronize CB of `App1_generator` and RB of `App1_square` at *tile*_0_0_3 and resume all paused neighbors to `App1_square`. Fig 6.7 shows the period that contains the reception of PAUSE_UPDATE and SYNCH_RESUME commands from $MigCtrl_2$ that concerns `App1_generator` and `App1_consumer`. The seven marks highlight commands actions as well as their effects. They are explained in the following:

M1 It shows the last value displayed by `App1_consumer` before the reception of PAUSE_UPDATE for the same task. Final value is $15^2 = 225$.

M2 The reception of PAUSE_UPDATE commands to pause `App1_consumer` (as shown in the figure, app. ID = 0, task ID = 1) and update the configurable channel connecting it with `App1_square`.

M3 `App1_generator` is still running as it is still generating numbers 16, 17.

M4 The reception of two commands PAUSE_UPDATE and SYNCH_RESUME. The first one is to pause `App1_generator` (as shown in the figure, app ID = 0, task ID = 0) and update the configurable channel connecting it with `App1_square`, no additional display that shows generations of `App1_generator` appears. The second command is sent to synchronize and resume both paused tasks. After synchronization takes place, resumption starts task by task. Resumption takes place here for `App1_generator`.

M5 As a result of the resumption, `App1_generator` continues generating reaching value 18.

M6 Resumption of `App1_consumer` takes place.

M7 `App1_consumer` continues displaying results newly calculated ($\times 10$) by `App1_square` in *tile*_0_0_3, Tokens are preserved as values carry on from 160.

Figure 6.8: Screen shot of the outputs of tiles *tile*_0_0_0, *tile*_0_0_2, and *tile*_0_0_3.

In fig 6.8, outputs of tiles *tile*_0_0_0, *tile*_0_0_2, and *tile*_0_0_3 are shown for the sake of comprehension. In *tile*_0_0_0, *MigSup* (referred to as *Broker*) sends MIG_IN and MIG_OUT to $MigCtrl_3$ and $MigCtrl_2$, respectively. Afterwards, it enters update mode in which it sends to *MigCtrls* for updating their DLT. In *tile*_0_0_3, no task displays its output as `App1_square` does not display its processing, hence, MIG_IN command reception is shown along with the creation of the task and its connecting channels. In *tile*_0_0_2, we see the beginning of the second migration that takes place in application 2 `App2_generator`, however, it is not shown since the first migration process shows enough details.

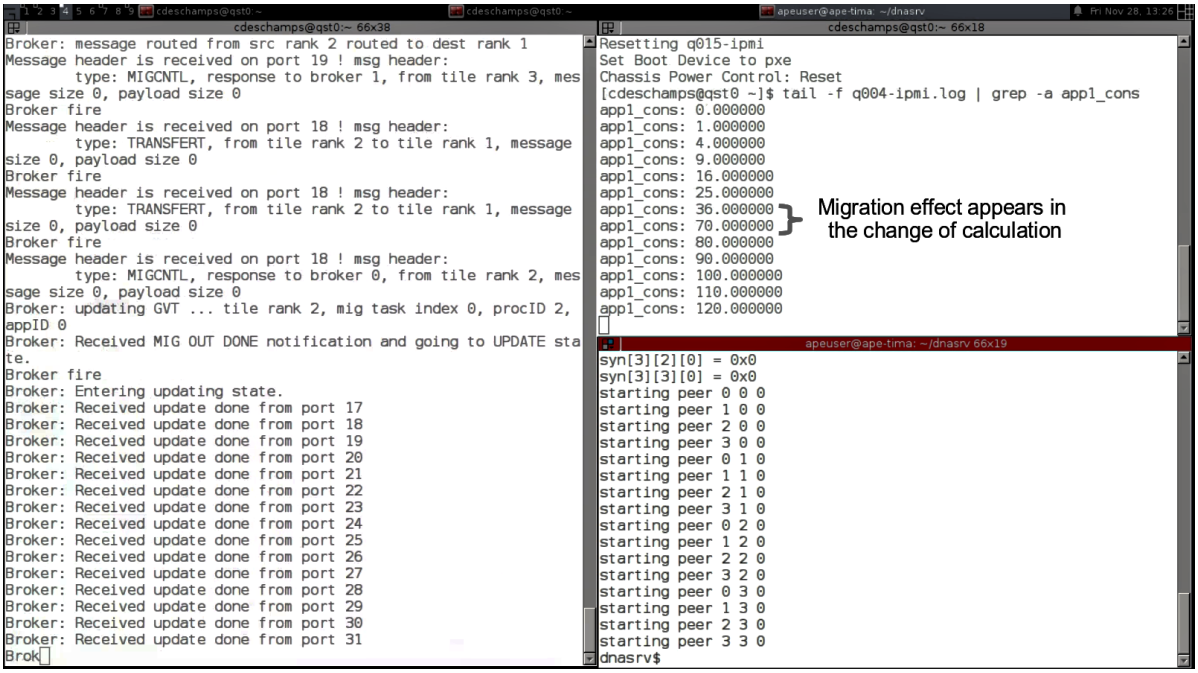

Figure 6.9: Screen shot of the outputs of QUonG platform: tile *tile*_0_0_0 (left half) and tile *tile*_0_0_1 (upper right part). Server output is shown in lower right part.

6.5.2 HW platform

In this section, we show the screen output of the real HW platform. Same applications with the same mapping shown in fig 6.6 are used for this experiment. However, the number of tiles is 16. As a result, the first four tiles are having the same mapping while the rest of them do not contain any application except that each one has a *MigCtrl* with a void DLT. The screens output of some HW tiles are shown in the right half of fig 6.9.

The HW platform is accessed remotely. A PXE[2] server is used in order to be able to run different SW on platform tiles. Firstly, a remote connection is established to access the server via `ssh` session. Secondly, a set of different scripts are used to reset, reboot tiles processors and connect terminals (using Linux `screen` command) to different tiles so as to be able to view their output. Thirdly, software is loaded from the local machine to remote tiles via another SW on the server, this software shows the synchronization of the tiles after they are reset such that what is shown in lower right part in fig 6.9 where the system contains 16 tiles which are all shown that they are starting.

[2]A Preboot Execution Environment server supplied a standardized client-server environment that boots software that is retrieved from network on PXE client with PXE capable network interface card.

Same explanation as previously shown in simulation in sec 6.5.1 can be given to the output of *tile*_0_0_1 shown in upper left part in fig 6.9. Migration takes place arbitrarily and calculation has changed from squaring to multiplying by ten as it is obvious in the transition between $36 = 6^2$ and $70 = 7 \times 10$. Output of right part of fig 6.9 shows that of the *MigSup* in *tile*_0_0_0, it shows the end of the update state where all DLTs of all *MigCtrls* on all tiles are updated.

Although we have our technique working on both platforms, we mention in the following sections the figures outputted from experiments run on the HW platform. This is because of the fact that they are more accurate and expressive than those coming out from simulation one which is used for debug and validation.

6.6 Performance overhead

Performance overhead due to migration is the time added on the execution time of an application due to migration. The measured execution time is the total time elapsed from the application starting time to its ending time including inter-tile communication. Since our solution is agent based, there shall be two components of this overhead. They are:

i Time added due to normal execution of the agents without migration.

ii Time added due to the process of migration itself.

That is why, to have a good idea about the overhead, we perform the measurement in three different cases for an application. They are as follows:

case 1 Migration agents do not exist. Hence migration cannot be performed. This case is meant for measuring the time an application needs to finish without any added overhead.

case 2 Migration agents exist but no migration is performed. This case is meant for measuring the overhead introduced by migration agents compared to case 1.

case 3 Migration process is done. This case is meant for measuring the overhead introduced by both migration agents and the migration process itself compared to cases 1 and 2.

The time is measured in x86 by calling function that uses High Precision Event Timer HPET to record the number of cycles from the beginning of execution. As a result, time is measured by calling the function twice at the start and end of the time required to be measured. Function call has negligible overhead.

Four tiles are used to accommodate two identical applications, each application contains three tasks. The tasks are: `generator` task, `processor` task and `consumer`

Table 6.1: Execution times of both applications and overheads.

Case	Starting time (μs)	Ending time (μs)	Execution time (μs)	Overhead (%)
Application 1				
1	65,027,461	67,247,538	2,220,077	N/A
2	28,401,948	30,623,281	2,221,332	**0.06**
3	21,754,757	24,379,930	2,625,173	**18.25**
Application 2				
1	33,163,108	35,387,392	2,224,284	N/A
2	28,687,143	30,911,856	2,224,713	**0.02**
3	40,811,884	43,358,217	2,546,333	**14.49**

task. All of the three tasks are not state-less tasks, i.e. they are developed to have states. The mapping of the application and migration agents is depicted in Fig 6.6. In Table 6.1 execution times are shown for every case and every application.

It is noticed that the agents add negligible overhead to the execution time of both applications as their increase is around 0.02% - 0.06%. Knowing that in typical situations migration is not an ordinary event that takes place regularly, there is almost no cost for deploying the agents along with the applications regarding performance.

The overhead due to migration increases the execution time by about 18% compared to that of case 1 of application 1 whereas just by 14% in application 2. The difference is due to the difference in the migrating task between both applications and its placement regarding its predecessors and successors in task graph. In application 1, APP1_square is the migrating task and it has two neighbors in different tiles. This increases the messages between *MigCtrls*, this takes longer time compared to application 2 where APP1_generator is the migrating task which has only one local neighbor.

6.7 Migration overhead vs. task state size

The aim of this set of experiments is to investigate how much migration overhead gets affected by changing task state size. This impacts the elapsed time taken in the transfer of task state size. It is also desired to investigate if the migration algorithm takes longer times when dealing with bigger sizes.

The experiments are executed on 4-tile platform with the same application and mapping shown in Fig 6.6. Task state size of `App1_square` is increased every time and migration time is measured. The data payload in each packet a network inter-

Table 6.2: Task state size vs. overhead. Third column contains percentage increase normalized to overhead of $4kB$.

Size (kB)	Overhead (μs)	Percentage increase (%)
4	530,923	-
8	531,017	0.0176
12	531,021	0.0184
16	531,066	0.0268
20	531,099	0.033
28	531,191	0.0504
34	531,271	0.0654
40	531,360	0.0823

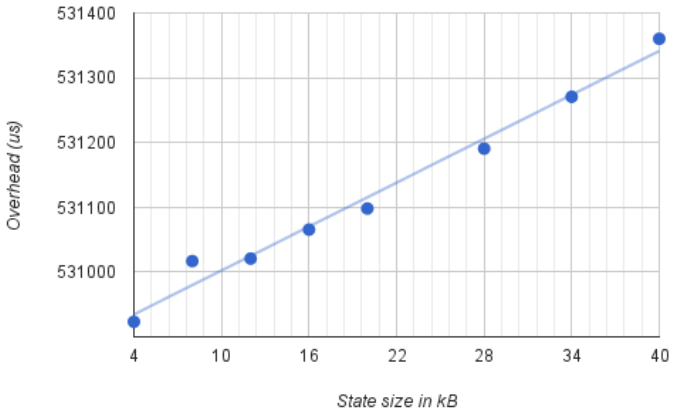

Figure 6.10: Performance overhead (in μs) vs. task state size (in kB).

face can send from its tile is limited to 4 kB, i.e. all tasks states that contain data with sizes until 4 kB can be sent in one packet. Task states with sizes more than 4 kB can be split into a number of 4 kB packets. As a result, when overhead time gets longer due to extra packets, it increases in ladder-like shape versus the state size where every step width of this ladder equals 4 kB. Table 6.2 shows states sizes with corresponding migration overheads.

There is a slight increase to the overall overhead due to the incremental state size increase. In Fig 6.10, a graph is depicted to show the increase in overhead due to the increases in state size. The scattered points show linear relation between task state size and overhead due to migration. A straight line equation is deduced by minimum sum of squared distances technique. Equation 6.1 can give good estimates of migration overhead for bigger state sizes.

$$\begin{gathered} y = 11.3x + 5.3 \times 10^5 \quad \mu s \\ \text{where} \quad x = 4 \times \left\lceil \frac{S}{4} \right\rceil \quad kB, \\ S \quad \text{is size in} \quad kB, \\ y \quad \text{is overhead in} \quad \mu s. \end{gathered} \tag{6.1}$$

$$\frac{dy}{dx} = 11.3 \quad \mu s/kB \quad \text{or} \quad 45.2 \quad \mu s/4kB \tag{6.2}$$

This means that every new packet of 4 kB added to the communication, there are approximately 45.2 μs added to the overhead. This can be observed in the numbers supplied in the previous table, for example, the difference in overhead between the two cases where state sizes are 16 kB and 12 kB respectively, is 44.7 μs and the same applies to the two cases 20 kB and 28 kB respectively 92.52 μs. This linear relation, however, is specific to several parameters like migrating task mapping.

6.8 Variation of migration overhead

The aim of this set of experiments is to investigate if the elapsed time due to migration varies even if it is repeated under almost the same conditions. This is to test how deterministic the algorithm of migration is. With such information, we can decide whether it is feasible to use predicted migration time with applications that have to be executed with certain amount of quality of service QoS.

A number of migration experiments are repeated and every time the same task is migrated in an arbitrary time. Every run, the machine has to reboot and that is to avoid cache influence. Task state size is fixed in all the experiments and does not exceed 4 kB, i.e. payload of one message is enough to transfer the task state. The results are given in Table 6.3. The range of variation of the overheads measured is 2.25 ms as illustrated in Fig 6.11.

Figures show that the variation of the overhead due to migration is limited. This is obvious as the whole range of variation between maximum and minimum of the set of overhead results under study is approximately 0.6% of the average time and the standard deviation is quite small (approximately 1 ms). This contributes to calculating better estimates of migration overhead depending on application mapping, migrating task state size and OS used. This makes the proposed migration solution able of introducing nearly deterministic migration costs for applications.

6.9 Overhead versus number of channels

Since task migration is performed in a distributed architecture via agents scattered among a number of tiles in a cluster, migration procedure is undergone via exchange of commands between tiles. That is why, elapsed time due to migration may depend

Table 6.3: Variation of migration overhead.

Experiment	Starting time (μs)	Ending time (μs)	Execution time (μs)	Overhead to Case 1 (μs)
1	27,801,647	30,425,029	2,623,382	402,057
2	45,614,455	48,237,843	2,623,387	402,062
3	78,243,189	80,866,640	2,623,450	402,125
4	36,361,808	38,987,310	2,625,503	404,177
5	121,649,077	124,273,322	2,624,246	402,920
6	24,258,600	26,884,037	2,625,437	404,111
7	56,529,629	59,154,948	2,625,319	403,994
8	69,221,222	71,844,467	2,623,245	401,920
9	31,294,232	33,919,547	2,625,315	403,990
10	88,779,483	91,402,913	2,623,430	402,105
Average time (μs)				402,946
Maximum time (μs)				404,177
Minimum time (μs)				401,920
Standard deviation (μs)				1,004
Difference (Max - Min) (μs)				2,258
Percentage of (Max-Min) to average time				0.56%

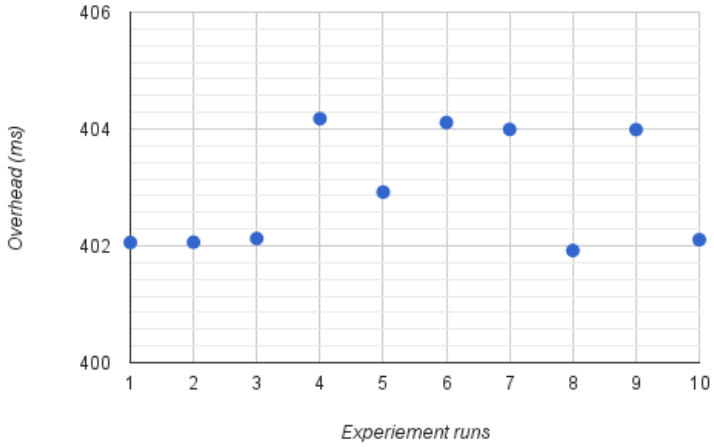

Figure 6.11: Migration overheads (in ms) variation.

on the number of neighboring tasks to the migrating one and how they are mapped. Knowing that, the aim of these experiments is to investigate how much migration performance overhead gets impacted when the number of neighbors for the same migrating task increases. The obvious consequence of an increase in the number of neighbors to a migrating task is the increase in the number of channels, hence, an increase in the number of commands that deals with such channels. As a result, longer time is supposed to be taken for migration.

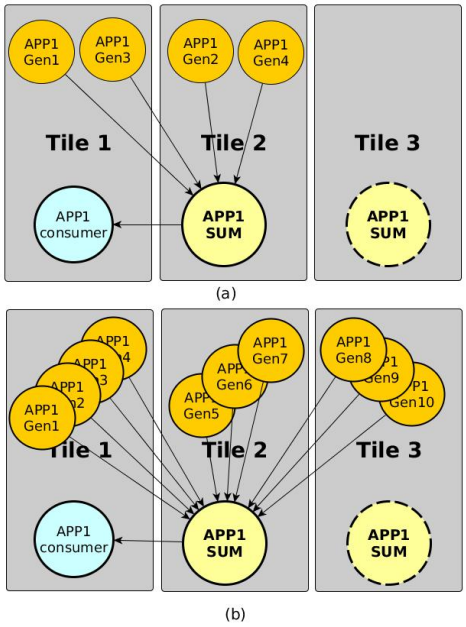

Figure 6.12: Mappings of the modified task graph on the three tiles (tile_1 - tile_3). (a) mapping is for 4 generators. (b) mapping is for 10 generators.

Application 1 task graph has been modified to produce two examples where the single generator is replaced by the following:

i Four generators as shown in Fig 6.12.(a). They are distributed on tile_1 and tile_2.

ii Ten generators as shown in Fig 6.12.(b). They are distributed on tile_1, tile_2 and tile_3.

The consumer and the migrating task, however, are kept as they were. Migrating task accumulates the input data coming from the generators. *MigSup* still resides in a separate tile (tile_0) as in Fig 6.6. The two examples are executed on the x86 based real hardware platform. Execution times are measured in case 2 and case 3. This is because it has already been shown that without migration, the overhead added due to the agents is negligible. Execution tiles are shown in table 6.4.

The migration time obviously increases when the number of the neighbors of a migrating task increases. This is due to the increase in number of necessary commands sent by $MigCtrl_{src}$ to all *MigCtrls* found in neighbors' tiles to execute the migration.

We can further investigate the relation between the number of channels and performance overhead due to migration by consolidating all the measurements from all the experiments shown in the past sections. We list all experiments with the

Table 6.4: Performance overhead with bigger number of generators.

Case	Starting time (μs)	Ending time (μs)	Execution time (μs)	Overhead to C2 (%)
Four generators				
2	17,482,080	19,565,805	2,083,724	N/A
3	24,940,435	27,628,979	2,688,544	**29**
Ten generators				
2	30,300,077	32,434,774	2,134,697	N/A
3	33,822,369	37,231,950	3,409,580	**59.7**

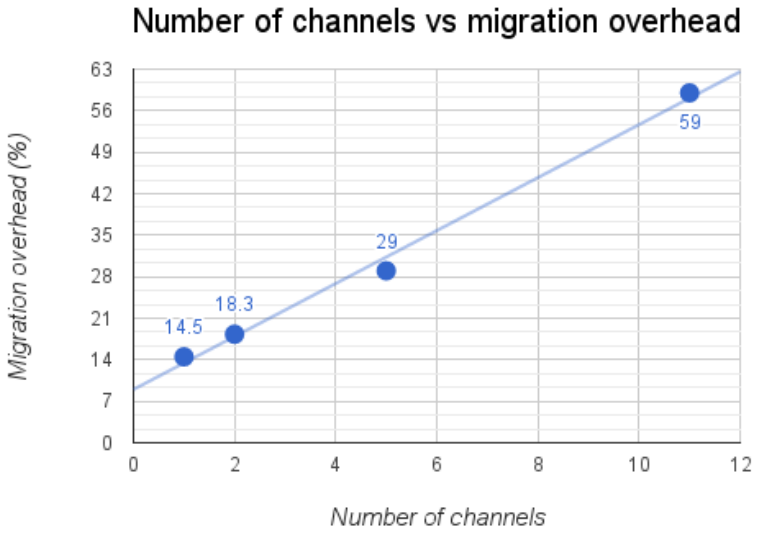

Figure 6.13: A linear relation can be deduced from the points scattered relating number of channels and migration overhead.

number of channels and their corresponding migration overhead in table 6.5. The values of migration overhead we have are for numbers of channels 1, 2, 4, and 10. The graph that is shown in Fig 6.13, shows that the relationship between the number of channels and the migration overhead is linear. A straight line can be deduced from the given points, the equation is shown in 6.3. This linear relationship can give us good estimates of migration overhead knowing the number of channels between the migrating task and its neighboring tasks. Starting, then, from equation 6.3 we can calculate how much overhead added by an increase in the number of channels by one, a simple differentiation outcome is shown in equation 6.4. Equation 6.4, migration performance overhead increases by almost 5% when there is one increase in number of channels. It is worth noting that the linear relation between number of channels and migration performance overhead varies depending on some parameters such as communication means, device and driver in the system. However, different platform parameters like communication do not change the linearity of the relationship but

Table 6.5: Number of channels vs. migration performance overhead.

Experiment	Number of channels	Migration overhead (%)
(App2) in table 6.1	1	14.5
(App1) in table 6.1	2	18.3
In table 6.4	4	29
In table 6.4	10	59

the coefficients of equation 6.3.

$$\begin{gathered} OH = 4.466N + 8.987 \\ OH : \text{overhead } \% \\ N : \text{number of channels} \end{gathered} \tag{6.3}$$

$$\frac{dOH}{dN} = 4.466 \tag{6.4}$$

6.10 Memory overhead

The aim of this set of experiments is to investigate how much memory increases due to the insertion of task migration agents and the existence of migrating tasks replicas. This takes place via comparing memory size between case 1 and case 3.

In table 6.6, memory overheads are shown. All the numbers are for `text` section and all overheads are measured compared to just this section as this represents the memory needed to be stored in flash memory. On x86 real HW platform, the overhead ranges from 7.2% to 9.2% which is reasonable. On the simulation side, agents and replicas add approximately the same amount of memory like on hardware but since on real HW more architecture specific codes are needed to make it operational, the part added by them is not significant compared to that on simulation. An example of these specific codes are the PCIe driver, this is in addition to the ported version of DNA-OS. That is why in table 6.6, the whole `.text` codes size of x86 are bigger than those of ARM. However, in table 6.7 the codes of both generated functions `init_GVT` and `init_DLT` on x86 are smaller than that on ARM, this is due to the difference in compiler options. `GCC` compiler is used in both cases with the correct architecture compiler option but the main reasons why there is such difference is that an optimization option `O2` is used with the compilation in case of x86 and the fact that RISC[3] code is usually bigger than CISC[4] because RISC is *load-store* architecture. As a result, the whole codes of x86 are bigger in size than that of ARM due to the added drivers and the difference between platforms, however, when comparing similar codes, it is found that those compiled on x86 are

[3]Reduced Instruction Set Computers. ARM is RISC architecture
[4]Complex Instruction Set Computers. x86 is CISC.

Table 6.6: Flash memory overhead (in bytes).

Section	**text section**			
Platform	x86			
Tile	***tile 0***	***tile 1***	***tile 2***	***tile 3***
W/O agents	258,118	250,502	251,382	249,702
With agents	276,630	272,054	273,542	272,710
Overhead (%)	**7.2**	**8.6**	**8.8**	**9.2**
Platform	ARM			
Tile	***tile 0***	***tile 1***	***tile 2***	***tile 3***
W/O agents	202,236	188,416	190,440	186,584
With agents	237,844	228,520	231,648	229,160
Overhead (%)	**17.6**	**21.3**	**21.6**	**22.8**

Table 6.7: Generated `init_GVT()` and `init_DLT()` code sizes on all tiles (in bytes).

x86			**ARM**		
`init_GVT`	`init_DLT`		`init_GVT`	`init_DLT`	
Tile 0	***Tile 1***	***Tiles 2,3***	***Tile 0***	***Tile 1***	***Tile 2,3***
564	99	615	1140	196	1272

smaller due to the optimization used during compilation.

In table 6.7, we investigate the generated code sizes of both functions `init_GVT()` and `init_DLT()` used by both agents: *MigSup* and *MigCtrl*, respectively. Since migrating tasks and their replicas reside only in *tile 2,3*, code size of `init_DLT()` in *tile 1* is significantly smaller than that of those found in *tile 2,3*. Since there are two migrating tasks in tile 2 as well as in tile 3, `init_DLT()` has same code size in both; however, the data put inside DLTs are different. `init_GVT()` contains 4 records corresponding to the 4-tiles, it has information about all migrating tasks found in all tiles in the cluster. That is why it is close to the `init_DLT()` of *tile 2,3*.

We now investigate the size occupied in RAM by GVT and DLT. They both are stored in the *heap* and occupy same memory on both architectures. GVT takes 240 bytes, both DLTs on tile 2 and 3 take the same memory size, each takes 152 bytes while that of tile 1 takes just 24 bytes.

We devise an equation 6.5 to calculate DLT size of memory occupied in RAM. Since DLT contains data structures of migrating tasks and their neighbors, size can be calculated easily. Equation terms are defined as follows:
S_{DLT}: total DLT size (bytes). $S_{c_{DLT}}$: DLT header ($S_{c_{DLT}} = 24$ bytes). N_m: Number of migrating tasks. $S_{m_{DLT}}$: constant size added by each migrating task data

structure ($S_{m_{DLT}} = 28$ bytes). N_{n_i}: Number of neighbors of a migrating task with index i (bytes). $S_{n_{DLT}}$: constant size added by each neighbor structure pointed to in the structure of migrating task ($S_{n_{DLT}} = 24$ bytes). By applying equation 6.5 on our case we have with $N_m = 2$ where $N_{n_0} = 2$ and $N_{n_1} = 1$, we get $S_{DLT} = 24 + 2 \times 28 + (2 \times 24 + 1 \times 24) = 152$ bytes.

$$S_{DLT} = S_{c_{DLT}} + N_m S_{m_{DLT}} + \sum_{i=0}^{N_m - 1} N_{n_i} S_{n_{DLT}} \tag{6.5}$$

Same applies to GVT size of memory occupied in RAM. In equation 6.6, GVT size can be calculated. Terms are as follows:
$S_{c_{GVT}}$: GVT header ($S_{c_{GVT}} = 8$ bytes), $S_{t_{GVT}}$: constant tile record header ($S_{t_{GVT}} = 20$ bytes), N: number of tiles in the cluster. N_{m_i}: number of migrating tasks in tile i. $S_{m_{GVT}}$: migrating task structure size ($S_{m_{GVT}} = 32$ bytes), $N_{n_{ij}}$: number of neighbors of a migrating task with index j on a tile with index i, $S_{n_{GVT}}$: neighbor structure size ($S_{n_{GVT}} = 4$ bytes). In our case: $N = 4$, $N_{m_0} = N_{m_1} = 0$, $N_{m_2} = N_{m_3} = 2$, $N_{n_{20}} = N_{n_{30}} = 2$, $N_{n_{21}} = N_{n_{31}} = 1$. By substituting: $S_{GVT} = 240$ bytes

$$S_{GVT} = S_{c_{GVT}} + N S_{t_{GVT}} + \sum_{i=0}^{N-1} \left(N_{m_i} S_{m_{GVT}} + \sum_{j=0}^{N_{m_i} - 1} N_{n_{ij}} S_{n_{GVT}} \right) \tag{6.6}$$

These equations clarify the relation between the size in RAM occupied by either DLT or GVT and different parameters like number of tile, migrating tasks or neighbors for every migrating task.

6.11 Rationale

We summarize here the rationale behind our choices in this chapter. Regarding the type of application used, we intentionally use a small application without much need for processing as an unfavorable case. This is to show how significant task migration performance overhead is to an already short execution time. This is also to show that task migration is still feasible to be performed even when the application is a small one, otherwise, it would be better to restart a migrating task at its destination if its migration overhead dominates its execution time.

We would like also to address comparison(s) with related work. The results given here cannot be compared to results given in the literature. This is because, to the best of our knowledge, there is no work in the literature which gives direct performance overhead numbers due to migration and investigates how this overhead increases with the increase in the number of migrating task neighbors. This is in addition to the fact that migration time depends on several different parameters, they are as follows:

- Migration protocol, how migration is performed.

- Platform architecture, processor architecture and communication means (network-on-chip, bus ...)
- Application mapping, where every application task shall be running on the system.
- Whenever an OS is used or bare metal.

Chapter 7

Conclusion and future work

Contents

In this chapter, we conclude the presented work to show our contributions and how this work is useful in the domain of multi-tiled architectures whether for embedded or high performance computing systems. Then, we discuss the future work that could extend our proposed solution. These extensions would increase our solution's features and abilities. The expanded work here is not yet implemented or validated and this is why it is considered to be future work.

We discuss here two proposals for two problems mentioning the rationale behind these proposals. In the second section, we discuss how with limited modification our solution can be tolerant to intermittent link faults. In the third section, we explain how address collision problem can be alleviated without impacting the nature of our solution.

7.1 Conclusion

We have covered several task migration issues in different architectures. We have proposed a light-weight agent based task migration solution targeting multi-tiled architectures. The solution is explained from both methodological and algorithmic point of view. We took into our consideration implementation issues and showed implementation details of the majority of the solution.

The given solution has several advantages. It is transparent to application programmers. Also since it is based on middleware layer and its communication interface is compliant with POSIX, it is portable to different core architectures. This technique is operational on real hardware platform as well as simulations. It shows low performance overhead (18% of the execution time of a typical small application where migrating task has one predecessor and one successor). Its performance overhead shows negligible variation when migration is repeated several times, hence, its cost with respect to the application can be considered to be deterministic.

Since now systems are packing more and more cores into them thanks to advancements in scaling technology and the introduction of even new ones that goes into Nano world, we did not ignore the scalability of a task migration solution and how important it is. We also provide an implemented framework for automating the insertion of the agents with different applications via an automatic generation tool-chain. This is to place the agents (*MigSup* and *MigCtrl*) in correct tiles according to application(s) mapping and more importantly generate their tables (GVT and DLT respectively). As a result better scalability is attained.

Future work that is expanded here can be easily extended to our work to make the solution more and more resilient to issues and tolerant to faults. This is achievable without much effort and at the same time without losing the advantages of the solution we propose.

7.2 Fault tolerance increase: link integrity issue

Until this point in this work, the task migration solution we have proposed cannot function in the presence of broken data channels between the migrating task T_m and one or more of its either predecessor (T_i) or successor (T_o). This is because of the possibility that both CB and RB on both ends of a configurable channel may not be synchronized, hence, tokens loss is inevitable. Another reason for not having a functional migration with broken data channels is the possibility that `FIRE` procedure is blocked waiting for incoming tokens from a faulty channel.

The given task migration solution uses migration point technique in order to stop and migrate T_m. Tasks are loop based and a migration point is checked every iteration which is the execution of `FIRE` procedure. A Migration point is not like an interrupt, hence, the iteration has to be completed every time in order to reach the migration point. The `FIRE` procedure is composed of three main parts which are: reading input tokens, processing input tokens, and writing the outcome of the processing part in the form of output tokens, if any[1]. As a result, data channels that connect application tasks together have to be functional as reading from broken

[1]Sometimes tasks do not have to send any tokens as they do not have successors; however, no task can exist without at least one predecessor.

ones will prevent the iteration from being completed, hence, the migration point would be unreachable. The same applies to the output data channels that are broken. Once T_m tries to write output tokens to T_o, a failure may be returned back by the communication driver which detects the break. Consequently, migration agents (especially $MigCtrl_{src}$) would not be able to proceed with the migration algorithm.

In normal execution case a migration request is issued and after a response time[2] it gets handled and migration starts. The worst case is when a migration request arrives right after the migration point check took place, in this case, migration response time (T_{r_M}) becomes equal to the time elapsed for finishing the whole iteration (or execution time to `FIRE` procedure) (T_{FIRE}). The time taken by `FIRE` to finish execution can be deduced to be the result of the summation of times taken by the three main parts of `FIRE` procedure neglecting the time taken for checking the migration point. As a result, T_{r_M} can be calculated from equation 7.1. A fault in an input data Channel Ch_x, for instance, makes the time elapsed for reading from that channel too long to finish normally the execution ($R_{Ch_x} \rightarrow \infty$ units of time).

$$T_{FIRE} = \sum_{i=1}^{N} R_{Ch_i} + T_{proc} + \sum_{i=1}^{M} W_{Ch_i}$$

N, M: numbers of input and output channels, respectively.

R_{Ch_i}: Time elapsed waiting for receiving tokens from Ch_i

T_{proc}: Time elapsed to process input tokens and produce output result

W_{Ch_i}: Time elapsed for sending output tokens to Ch_i (7.1)

7.2.1 Proposed solution

The aim of this section is to explain our proposed solution. The main aim of such solution is to circumvent the two aforementioned issues that might result from broken channels. It is just required that task migration still can function in the presence of a physical defect and by that it becomes tolerant to broken channels faults.

In the provided `MProcFW` layer, the high level `read` function can detect migration request if it is waiting for incoming tokens but no further action regarding migration can be taken. Migration request can also be detected by `write` function. As a result, from the implementation perspective, migration detection in communication primitives is possible and existent without proceeding in migration process so as to avoid losing partially received tokens. At the writing part written output tokens that can never reach T_o will be lost in case of migration, too.

The main idea of the solution can be explained in the following points:

[2]Response time is the time elapsed starting from the arrival of a migration request till actual migration starts.

1. In order to avoid losing input tokens, all received tokens would be stored in RBs during the whole T_{FIRE} and only those which were consumed should be released from RBs after successful writing. This insures that tokens (even consumed ones) are still stored in CBs at T_i side(s) which means they can be redirected to destination tile when migration takes place and safe resumption is still possible.

2. In order to avoid losing output tokens, there should be a notification from the communication driver informing the writing function that a certain output channel is down so all output tokens that are failed to be sent are tagged with a special tag that informs $MigCtrl_{src}$ that these tokens need to be resent and stored in CB tagged. $MigCtrl_{src}$ has to additionally check all CBs for similar tags and all unsent tokens should accompany the state to $MigCtrl_{dest}$. As a result, a limited modification in `LOAD_STATE` command is required so that it can send both, each is preceded by an informing tag so that $MigCtrl_{dest}$ can differentiate and know the correct CB of the correct configurable channel and resend the tokens from there to the correct T_o. A limited modification is done also to the structure of CBs so as to store tags also as depicted in fig 7.1.

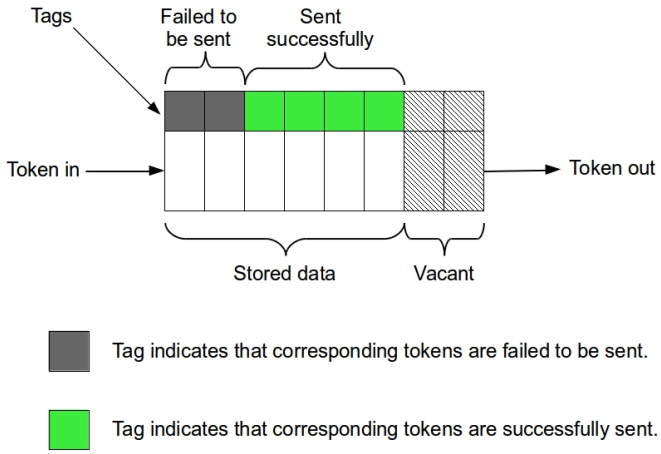

Figure 7.1: Modified CB with tags. A tag exists for every token stored.

The second point involves limited modifications in the code of *MigCtrls*, as well as, in the code of `write` API provided by `MProcFW`. Unlike the first point which can be achieved with a small modification in the task model in addition to another limited one in `read` function. We propose to separate both reading and writing parts out of `FIRE` procedure leaving only the processing part inside. This leads to the addition of `READALL` and `WRITEALL` procedures in the task model and application developers have to provide them along with the other procedures. Inside `READALL` a developer has to read all the input tokens required for the processing part and the same applies to `WRITEALL` in which all output tokens should be written, reading

and writing are done as usual by calling the same APIs from `MProcFW`.

In listing 7.1, the modified version of the task model is shown, the original used task model was shown earlier in listing 5.1 in section 5.2.1. The added `ack_rel_RB` function is responsible for releasing consumed tokens from RBs and it is called after `WRITEALL` procedure by checking the received ACKs. `ppINCh` and `ppOUTCh` are pointers to arrays of input and output channels, respectively. In `READALL` all tokens are read from external channels and stored in RBs while in `FIRE` all tokens are read from RBs to internal variables for processing. The same applies to `WRITEALL` but with CBs instead of RBs.

```
procedure INIT(TaskStructure *t)  // initialization
  initialize();
end procedure
procedure READALL(Channel **ppCh ,TaskStructure *t)
  read(ppINCh[0]->RB0, l0);     // read l0 bytes from Ch0
  .
  .
  read(ppINCh[n]->RBn, ln);     // read ln bytes from Chn
end procedure
procedure FIRE(TaskStructure *t) // execution
  fifo_get(ppINCh[0]->RB0, &inVAR0);
  .
  .
  fifo_get(ppINCh[n]->RBn, &inVARn);
  outVAR0 = process_0(inVAR1,inVAR2, ... inVARn);
  .
  .
  outVARm = process_m(inVAR1,inVAR2, ... inVARn);
  fifo_add(ppOUTCh[0]->CB0, &outVAR0);  // write q0 bytes to CB0
  .
  .
  fifo_add(ppOUTCh[m]->CBm, &outVARm);  // write qm bytes to CBm
end procedure
procedure WRITEALL(Channel **ppCh ,TaskStructure *t)
  write(ppOUTCh[0]->CB0, q0);    // write q0 bytes to Ch0
  .
  .
  write(ppOUTCh[m]->CBm, qm);    // write qm bytes to Chm
end procedure
ack_rel_RB(ppINCh);
procedure FINISH(TaskStructure *t) // cleanup
  cleanup();
end procedure
```

Listing 7.1: Modified task model.

7.2.2 Example

We give here an example for the sake of further explanation. A five task application exists where T_3 is migrate-able. We show how the original task model would look as if they are filled directly with developer's code to show the communication part. This is to show how without the proposed modifications in the task model blockage is possible. We, then, inspect three scenarios; a normal execution one and the others are faulty ones. One of the input channels is broken in one scenario while one of the output channels is broken in the other.

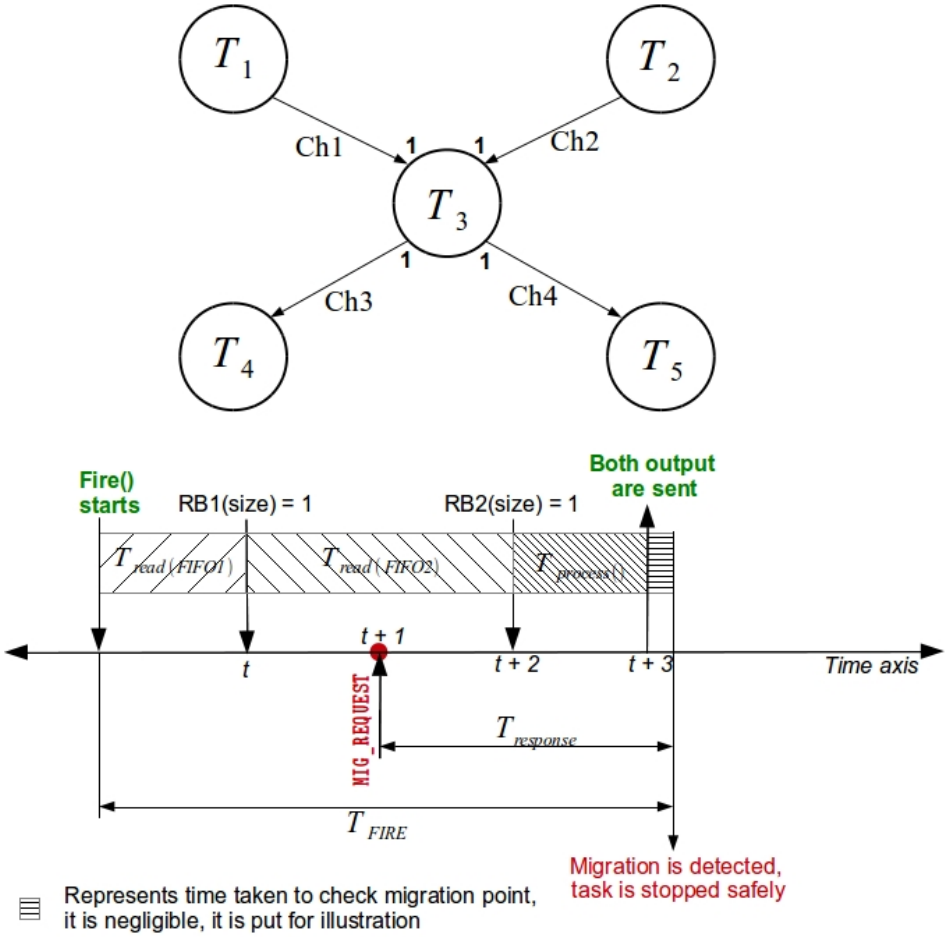

Figure 7.2: Upper right part depicts the process network of the application, the behavior and interaction of migration request are illustrated on time axis. One token is consumed by T_3 via every input channel $Ch1$ and $Ch2$ and one token is sent by T_3 via channel $Ch3$ to T_4.

Original task model in this example case is listed in listing 7.2. We inspect first the normal execution scenario, both process network and normal execution scenario timing are shown in fig 7.2. At time t units, T_1 writes to T_3 while at $t+2$ units T_2

writes to T_3, we neglect the time taken by the tokens in the means of communication. Migration request is issued in between the two readings at $t+1$ units. Even with the original task model, migration can be handled and continued normally after a certain response time ($T_{r_M} = T_{response}$).

```
procedure INIT(TaskStructure *t)  // initialization
  initialize();
end procedure
procedure FIRE(TaskStructure *t) // execution
  read(Ch1->RB1, n);              // read n bytes from RB1.
  read(Ch2->RB2, m);              // read m bytes from RB2.
  l = process1(n,m);              // processing i/p data.
  q = process2(n,m);              // processing i/p data.
  write(Ch3->CB3, l);           // write l bytes to CB3 & send.
  write(Ch4->CB4, q);           // write q bytes to CB4 & send.
end procedure
procedure FINISH(TaskStructure *t) // cleanup
  cleanup();
end procedure
```

Listing 7.2: Original task model with details on reading and writing sections of T_3.

Suppose that the second scenario has the same timing of the first except that $Ch2$ is broken, hence, no reception can be fulfilled. With the original task model, migration point cannot be reached. This highlights the difference between the original and modified versions of the task model. Modified task model looks like what is shown in listing 7.3. The second function call of `read` in `READALL` procedure would detect the migration request that arrived at $t+1$ units. Received tokens remain stored in RBs until they get released in the function call `ack_rel_RB()` which is never reached since T_3 gets stopped at `READALL`.

```
procedure INIT(TaskStructure *t)  // initialization
  initialize();
end procedure
procedure READALL(Channel **ppCh ,TaskStructure *t)
  read(ppINCh[0]->RB0, l0);     // read l0 bytes from Ch1
  read(ppINCh[1]->RB1, l1);     // read l1 bytes from Ch2
end procedure
procedure FIRE(TaskStructure *t) // execution
  inVAR0 = fifo_get(ppINCh[0]->RB0,l0);
  inVAR1 = fifo_get(ppINCh[1]->RB1,l1);
  outVAR0 = process_0(inVAR0,inVAR1);
  outVAR1 = process_1(inVAR0,inVAR1);
  fifo_add(ppOUTCh[0]->CB0, &outVAR0);
  fifo_add(ppOUTCh[1]->CB1, &outVAR1);
end procedure
procedure WRITEALL(Channel **ppCh ,TaskStructure *t)
  write(ppOUTCh[0]->CB0, q0);    // q0 = sizeof(outVAR0)
```

```
  write(ppOUTCh[1]->CB1, q1);    // q1 = sizeof(outVAR1)
end procedure
ack_rel_RB(ppINCh);
procedure FINISH(TaskStructure *t) // cleanup
  cleanup();
end procedure
```

Listing 7.3: Modified task model. ppINCh[0] represents Ch1. ppINCh[1] represents Ch2. ppOUTCh[0] represents Ch3. ppOUTCh[1] represents Ch4. Same notation applies to RBs and CBs.

Suppose that the third scenario has a broken output channel which is $Ch4$. The migration request arrives between the two writings in `WRITEALL`. As a result, the second `write` detects the failure in sending output tokens to T_5 hence it tags them and stores them in `ppOUTCh[1]->CB1` and then stops T_3. When $MigCtrl_{src}$ proceeds with the migration process, it checks both CBs for tagged tokens. Once it finds tagged tokens, it sends them after the state to $MigCtrl_{dest}$. $MigCtrl_{dest}$, then, resends the received tagged tokens and leaves an untagged copy in the corresponding CB there. Finally, migration is done and safe resumption is possible, afterwards.

7.2.3 Conclusion

We have shown a solution that makes our task migration more tolerant to data channels faults. It is assumed, however, that control channels which connect *MigSup* and *MigCtrls* are reliable because the tile in which *MigSup* is run is a reliable piece of HW. We investigate here the gained advantages from this solution as an extension to our original one, they are as follows:

1. The required modifications in order to have an extended task migration solution would not lead to increase the time elapsed for migration, hence, performance overhead almost remains as it is.
2. With the separation the main three parts of `FIRE` procedure, the migration response time T_{r_M} depends only on T_{proc} which is not usually significantly long. This leads to the effect of increasing migration points although only one is used.
3. No memory overhead is introduced to obtain this feature.
4. Changing the task model may lead to lower re-usability as legacy codes would need to be modified; however, transparency is still supported.

7.3 Address collision alleviation

We propose a solution for address collision issue. This solution can be added to our task migration technique. In this section, we expand the solution with the aid of an example for the sake of clarification.

We have expanded earlier the problem of address collision in section 4.2.1. It stems from the possibility that the state of the migrating task may contain pointers that are mostly dynamically allocated in RAM. Such state cannot be migrated as it would create `segmentation fault` error at the destination once the task is resumed. This is because pointers still have their old addresses which do not correspond to dynamically allocated chunks of memory in the heap at the destination.

Our task migration solution is an agent based, i.e. agents exist in application layer and are supported from a middleware layer beneath. As a result, any solution to address collision issue to be as extension to our solution should be a *user-level* solution. This extension should not violate the transparency of the whole task migration solution or any other of its characteristics.

This solution to address collision problem is like what is presented in [36] with some modification that fit our case and requirements. The solution presented in [36] is not transparent as application developers have to use specific library. A developer has to replace any ordinary pointer declaration with a specific structure, for instance. This is, in addition to the use of specific functions that are supplied by the aforementioned library instead of those supplied by `libc`.

The main objective of this solution is to enable *MigCtrls* to transfer states that contain variables and dynamically allocated chunks of memory. This means that $MigCtrl_{src}$ would be able to collect such state and send it in a format that can be readable and re-constructible by $MigCtrl_{dest}$ on the destination tile. After reconstruction, the task would be able to resume its execution normally. We put this proposed solution briefly in the following points then we expand them, afterwards:

1. *MigCtrls* are equipped with a way by which they can know where the pointers locations in the state are. *MigCtrls* require this information away from the application developer. This information can be supplied to the *MigCtrl* statically.

2. For every dynamic allocation process is performed, both the address and the amount of allocated memory should be registered in a record. The table that contains all records that correspond to all dynamic memory allocations would be accessible by the *MigCtrl*. As a result, this table has to be filled dynamically.

3. Introducing alternative functions (APIs) responsible for dynamic allocation that replace those used by developers like (`malloc`, `calloc, free` and `realloc`). Such alteration in developer's code has to be performed automatically with an aid from a tool so as not to violate the transparency and re-usability of application codes. As a result, tools for performing code transformation before compilation and linking in order to replace functions used for dynamic

allocation by the developer by new introduced ones that can fill the table mentioned above.

7.3.1 Solution description

The proposed solution is based on the intervention in application codes after its development away from their developers, in addition to adding some work to *MigCtrls* to deal with dynamic allocation in almost all its case. In the following, we expand how applications codes can be modified transparently without altering their behavior in addition to the kind of modifications.

It is assumed that all pointers are supposed to be pointing at dynamically allocated chunks of memory. *MigCtrl* is supposed to scan the migrating task state looking for pointers. This requires static information about the state items as a *MigCtrl* knows where a state begins in the memory buy it does not know what the contents are. Consequently, there should be a sort of map that informs $MigCtrl_{src}$ (responsible for collecting migrating state) what items a state has.

Such map can be extracted from application code without altering the code. We call it *state map*. We propose it to be a string that is composed of tuples, every tuple is a triplet that represents an item in the state structure by listing its data type size, number if it is declared as an array, and if it is a pointer. A tuple is as follows:`(data type size, number, pointer)`. First two elements in a tuple are self-explanatory unlike the third one that needs a bit more clarification, the aim from the third element is to describe the pointer. Its value equals to zero if the tuple describes a variable not a pointer whereas it equals to 1 or 2 if it is a pointer or a pointer to pointer. These tuples are altogether listed in a series and supplied to *MigCtrl* in a string format. During migration, once $MigCtrl_{src}$ is collecting the state before sending `LOAD_STATE` command to $MigCtrl_{dest}$, it would use the information supplied by the state map string to know where variables and pointers exist.

To this end, a *MigCtrl* can run through a state knowing the items in memory. However, a *MigCtrl* is still incapable of collecting data in a dynamically allocated memory in the heap. This is due to the fact that it does not have any information except the beginning address starting from which it can collect data from the heap. As a result, extra information about every dynamic allocation that takes place during the execution has to be available to *MigCtrl* to know how much data it needs to copy and send to $MigCtrl_{dest}$. This kind of information is known only during run-time. As a result some code modification is inevitable. We propose a way to fulfill that without having to violate the transparency of the task migration solution we propose. In order to register run-time information about dynamic allocations, we propose a format in which such information would exist, a means to store this information, and a way to save this information after every dynamic allocation. We assume that dynamic allocations are done by calling standard `malloc, calloc,` or

`realloc` and that freeing space at the end of execution is done by calling standard `free`, all are from `libc`.

We propose to save run-time information about these dynamic allocations in a table accessible to *MigCtrl*. Addresses, size of data type, and number are all saved in this table such that a record represents a dynamic allocation process, we call this table *Pointer Table*. This enable the *MigCtrl* that once it finds a pointer (which contains an address in the heap) in the state it can then search for it in the aforementioned table and by finding its record, it can then collect the required data from the heap. When $MigCtrl_{src}$ sends this data to $MigCtrl_{dest}$, it precedes the collected data with a flag that represents a pointer without sending the value of the pointer (address) found in the memory at source tile. This tag also contains the information found in the corresponding record of the Pointer Table. With this technique several kinds of pointers can exist in a state including a linked list.

To this end, we need to know how information related to dynamic allocations can be stored. We propose a group of APIs that perform the same functionality as `malloc, calloc, realloc,` and `free`, in addition to filling the information entered by the application developer in the Pointer Table. Then, we propose that these newly introduced APIs would replace the standard ones used by developers by the aid of a tool that does code transformation. For example `void *malloc(size_t size);`
is replaced by:
`void *MALLOC(uint32_t n, uint32_t s, PointerTable_t *pPtrTbl);`
`uint32_t`: is unsigned int `PointerTable_t`: is the data type of the Pointer Table (listed in listing 7.4), it is a structure that contains records and each one is of a `PointerTableRec_t` structure that is listed in listing 7.4: New arguments are as follows: `n`: number of items if it is a table, `s`: data type size in bytes.

```
typedef struct _PointerTable_t {
    PointerTable_record_t  *table;
    uint32_t               size;
    uint32_t               full;
}PointerTable_t;
typedef struct _PointerTable_record_t{
    uint32_t size;       /* size of datatype in bytes*/
    uint32_t number;     /*number of items in a table*/
    void     * p_add;
}PointerTable_record_t;
```

Listing 7.4: The structure of the table of pointers.

In order to automate this solution, we propose a transformation flow that is able of transforming code with a parsing and code editing tool. The flow is depicted in fig 7.3. We first start with application `.c` and `.h` files, they are inputted to a

preprocessor that functions the same of preprocessing in the compiler, its output is composed of the same files but with all names are resolved if any `#define` is used. Then we would be having the state structure that is inputted to state map extraction stage, the output of this stage is the string that is added to the corresponding task files so that *MigCtrl* can access it.

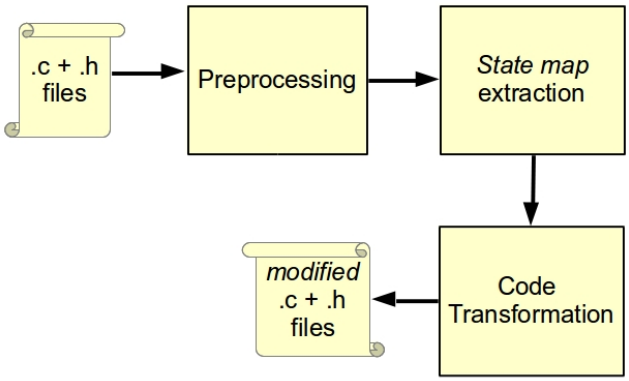

Figure 7.3: Flow of code modification.

7.3.2 Example

We put here an example that contains a variety of pointers and pointer to pointers. It is in listing 7.5. A tuple that describes every item exists as a comment in the same line of code. The state map string of such state is as follows: `char state_map="(4,1,0)(1,1,1)(8,1,2)(0,1,1)(0,1,2) ((8,1,0)(4,1,0)(4,1,0),50,0)((4,1,0)(1,1,1)(L,1,1),1,1) ((1,40,0)(1,1,1),1,2)(8,2,1)";`

Symbol (L) is the value of the first element (size of data type) in a tuple that represents a linked list. `*pstrct2` is a linked list. Once $MigCtrl_{src}$ finds (L) in a tuple while parsing the state map, it gets informed that this is a linked list and then it has to start a specific routine to collect all linked items by checking `next` pointer. More complex types of linked lists can be followed and collected with more complicated algorithms; however, this is the scope of implementation choices.

```
typedef struct _sruct1
{
  unsigned char l[40];     /*(1, 40, 0)*/
  char     *str;           /*(1, 1, 1)*/
}struct1;
typedef struct _struct2
{
  int              a;      /*(4, 1, 0)*/
  char             *p;     /*(1, 1, 1)*/
  struct _struct2 *next;   /*(L, 1, 1)*/
```

```
}struct2;
typedef struct _sruct3
{
  double a;          /*(8, 1, 0)*/
  float  b;          /*(4, 1, 0)*/
  int    c;          /*(4, 1, 0)*/
}struct3;
typedef struct _state_t /*state structure*/
{
  int    a;          /*(4, 1, 0)*/
  char   *pstr;      /*(1, 1, 1)*/
  double **pp;       /*(8, 1, 2)*/
  void   *p;         /*(0, 1, 1)*/
  void   **genpp;    /*(0, 1, 2)*/
  strct3 A[50];      /*((8, 1, 0)(4, 1, 0)(4, 1, 0), 50, 0)*/
  strct2 *pstrct2;   /*((4, 1, 0)(1, 1, 1)(L, 1, 1), 1, 1)*/
  strct1 **ppstrct1; /*((1, 40, 0)(1, 1, 1), 1, 2)*/
  double *pdble[2];  /*(8, 2, 1)*/
}state_t;
```

Listing 7.5: The state structure.

7.3.3 Conclusion

We have proposed a solution for address collision issue that can be considered as a doable extension to our original task migration solution or even similar agent based solutions. We think that this solution will not be of negligible performance overhead, though. The time of collecting data from the heap depends on some parameters like the type of the data and its size. Collecting a linked list, for instance, is not something trivial regarding the required time. Searching the Pointer Table should be taken care of to avoid as much as possible the variance of searching time or to minimize the searching time to the lowest possible extent. However, task migration is not a common event which makes its performance overhead acceptable if the penalty of not doing it is data loss.

Appendix A

MProcFW details

In this section, more details are given about MProcFW

A.1 Tasks category

This category lists all functions that create, control and destroy processes in table A.1.

API name	Role
MProc_proc_create	fills in the channel structure and open the necessary file descriptor for inter-tile communications.
MProc_proc_bootstrap	is responsible for non-blocking writing using write and copy protocol.
MProc_proc_start	is responsible for non-blocking reading using write and copy protocol.
MProc_proc_pause	is invoked when every application writes to another. Internally, it polls over *MProc_channel_write_nb*. Suffix 's' refers to synchronous write which is compatible to blocking writing in KPN applications.
MProc_proc_migrate	is invoked when every application writes to another. Internally, it polls over *MProc_channel_read_nb*. Suffix 's' refers to synchronous read which is compatible to blocking reading in KPN applications.
MProc_proc_stop	is invoked during migration process. Its role is to update only configurable channels, i.e. to change the branch of the configurable channel so as to connect the migrating task in its new location to its neighbors to keep the communication consistent after resumption
MProc_proc_delete	is invoked during migration process so as to avoid losing unprocessed tokens by the migration task in its original location. Thanks to *write with copy* protocol used, the sender always keeps a copy of all tokens that were sent and have not been consumed yet by the receiving task. This facilitates synchronization between the neighbor and the migrating task after reaching its destination.

Table A.1: APIs provided by MProcFW

A.2 Channels category

This category lists all functions that create, control and close channels in table A.2.

API name	Role
MProc_channel_create	fills in the channel structure and open the necessary file descriptor for inter-tile communications.
MProc_channel_write_nb	is responsible for non-blocking writing using write and copy protocol.
MProc_channel_read_nb	is responsible for non-blocking reading using write and copy protocol.
MProc_channel_write_s	is invoked when every application writes to another. Internally, it polls over *MProc_channel_write_nb*. Suffix 's' refers to synchronous write which is compatible to blocking writing in KPN applications.
MProc_channel_read_s	is invoked when every application writes to another. Internally, it polls over *MProc_channel_read_nb*. Suffix 's' refers to synchronous read which is compatible to blocking reading in KPN applications.
MProc_channel_update	is invoked during migration process. Its role is to update only configurable channels, i.e. to change the branch of the configurable channel so as to connect the migrating task in its new location to its neighbors to keep the communication consistent after resumption
MProc_channel_synch	is invoked during migration process so as to avoid losing unprocessed tokens by the migration task in its original location. Thanks to *write with copy* protocol used, the sender always keeps a copy of all tokens that were sent and have not been consumed yet by the receiving task. This facilitates synchronization between the neighbor and the migrating task after reaching its destination.
MProc_delete_channel	is invoked to close any file descriptor opened if necessary.

Table A.2: APIs provided by MProcFW

Appendix B

List of publications

Conference, Journal research articles and technical reports:
Authored:

1. Ashraf El-Antably, Frédéric Rousseau. "**Task migration in multi-tiled mpsoc: challenges, state-of-the-art and preliminary solutions**". Technical report, Journées Nationales du Réseau Doctroal en Micro-nanoélectronique JNRDM Marseille, France, 2012.

2. Ashraf El-Antably, Nicolas Fournel, and Frédéric Rousseau. "**Lightweight task migration in embedded multi-tiled architectures using task code replication**". Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on, pages 93-99, Oct 2014.

3. Ashraf El-Antably, Nicolas Fournel, and Frédéric Rousseau. "**Integrating task migration capability in software tool-chain for data-flow applications mapped on multi-tiled architectures**". In Digital System Design (DSD), 2015 Euromicro Conference on, pages 33-40, Aug 2015.

4. Ashraf El-Antably, Olivier Gruber, Nicolas Fournel, and Frédéric Rousseau. "**Transparent and portable agent based task migration for data-flow applications on multi-tiled architectures**". In Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, CODES '15, pages 183-192, Piscataway, NJ, USA, 2015. IEEE Press.

Co-authored:

1. Lars Schor, Iuliana Bacivarov, Luis Gabriel Murillo, Pier Stanislao Paolucci, Frédéric Rousseau, **Ashraf El-Antably**, Robert Buecs, Nicolas Fournel, Rainer Leupers, Devendra Rai, Lothar Thiele, Laura Tosoratto, Piero Vicini, Jan Weinstock, "**EURETILE Design Flow: Dynamic and Fault Tolerant Mapping of Multiple Applications Onto Many-Tile Systems**". Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on, ISPA 2014, pages 182-189, Italy, 2014, IEEE Press.

2. Pier Stanislao Paolucci, Andrea Biagioni, Luis Gabriel Murillo, Frédéric Rousseau, Lars Schor, Laura Tosoratto, Iuliana Bacivarov, Robert L. Buecs, Clément Deschamps, **Ashraf El-Antably**, Roberto Ammendola, Nicolas Fournel, Ottorino Frezza, Rainer Leupers, Francesca Lo Cicero, Alessandro Lonardo, Michele Martinelli, Elena Pastorelli, Devendra Rai, Davide Rossetti, Francesco Simula, Lothar Thiele, Piero Vicini, Jan H. Weinstock. "**Dynamic Many-process Applications on Many-tile Embedded Systems and HPC Clusters: the EURETILE programming environment and execution platforms**". Journal of Systems Architecture, 2015 Nov 24, available online.

Bibliography

[1] Gordon E Moore et al., "Cramming more components onto integrated circuits", *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. (Cited on pages 2 and 23.)

[2] "The curve of moore's law". (Cited on page 24.)

[3] Michael Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. (Cited on page 26.)

[4] A. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 2^{nd} edition, 1992. (Cited on page 28.)

[5] W. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin, "Thermal-aware allocation and scheduling for systems-on-a-chip design", in *Proceedings of the Design, Automation, and Test in Europe (DATE'05)*, 2005. (Cited on page 29.)

[6] Shekhar Borkar, "Design perspectives on 22nm cmos and beyond", in *46th ACM/IEEE Design Automation Conference,*, 2009, pp. 93–94. (Cited on page 30.)

[7] John L. Hennessy and David A. Patterson, *Computer Architecture : A Quantitative Approach*, 4^{th} edition, 2007. (Cited on page 32.)

[8] D.S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration", *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000. (Cited on pages 36 and 37.)

[9] 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.* IEEE, 2007, pp. 59–64. (Cited on pages 8, 37, 50, 54 and 67.)

[10] A. Andrea, A. Andrea, C. Salvatore, and P. Michele, "Assessing task migration impact on embedded soft real-time streaming multimedia applications", *EURASIP Journal on Embedded Systems*, vol. 2008, 2008. (Cited on pages 8, 37, 50, 54 and 67.)

[11] Eugene M Izhikevich, "Which model to use for cortical spiking neurons?", *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004. (Cited on page 42.)

[12] Pier S Paolucci, Ahmed Jerraya, Rainer Leupers, Lothar Thiele, Piero Vicini, et al., "Shapes:: a tiled scalable software hardware architecture platform for embedded systems", in *Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06. Proceedings of the 4th International Conference*. IEEE, 2006, pp. 167–172. (Cited on page 43.)

[13] ARM Limited, "Mpcore linux 2.6 smp kernel and tools". (Cited on page 50.)

[14] X. Guérin and F. Pétrot, "A system framework for the design of embedded software targeting heterogeneous multi-core socs", in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 153–160. (Cited on pages 14, 50 and 91.)

[15] A. Barak, O. La'adan, and A. Shiloh, "Scalable cluster computing with mosix for linux", *Proc. 5-th Annual Linux Expo*, vol. 100, 1999. (Cited on page 50.)

[16] 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*. European Design and Automation Association, 2006, pp. 15–20. (Cited on page 50.)

[17] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the multi-processor soc design space with systemc", *The Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, 2005. (Cited on page 50.)

[18] A. Greiner, "Tsar: a scalable, shared memory, many-cores architecture with global cache coherence", in *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, 2009. (Cited on page 50.)

[19] F. Mulas, D. Atienza, A. Acquaviva, C. Salvatore, and L. Benini, "Thermal balancing policy for multiprocessor stream computing platfroms", *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 12, pp. 1870–1882, 2009. (Cited on pages 50 and 51.)

[20] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, and M. Robert, "Mpi-based adaptive task migration support on the hs-scale system", in *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*. IEEE, 2008, pp. 105–110. (Cited on page 50.)

[21] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii, "Adaptive task migration policies for thermal control in mpsocs", in *VLSI 2010 Annual Symposium*. Springer, 2011, pp. 83–115. (Cited on page 50.)

[22] E.W. Brião, D. Barcelos, F. Wronski, and F.R. 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*. IEEE, 2007, pp. 296–299. (Cited on page 51.)

[23] D. Barcelos, E. Wenzel Brião, and F. Rech 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*. 2007, SBCCI '07, pp. 282–287, ACM. (Cited on page 51.)

[24] Gabriel Marchesan Almeida, Sameer Varyani, Rémi Busseuil, Gilles Sassatelli, Pascal Benoit, Lionel Torres, Everton Alceu Carara, and Fernando Gehm Moraes, "Evaluating the impact of task migration in multiprocessor systems-on-chip", in *Proceedings of the 23rd symposium on Integrated circuits and system design*. ACM, 2010, pp. 73–78. (Cited on pages 4, 5, 51 and 55.)

[25] FangFa Fu, Liang Wang, Yu Lu, and Jinxiang Wang, "Low overhead task migration mechanism in noc-based mpsoc", in *ASIC (ASICON), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 1–4. (Cited on pages 4, 5, 51 and 52.)

[26] Wei Quan and Andy D Pimentel, "A system-level simulation framework for evaluating task migration in mpsocs", in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*. IEEE, 2014, pp. 1–9. (Cited on pages 4, 5, 51 and 52.)

[27] Stefan Wallentowitz, Volker Wenzel, Stefan Rösch, Thomas Wild, Andreas Herkersdorf, and Jörg Henkel, "Dependable task and communication migration in tiled manycore system-on-chip", in *Forum on Specification & Design Languages*, 2014. (Cited on pages 4, 6, 51 and 52.)

[28] Pranav Tendulkar and Sander Stuijk, "A case study into predictable and composable mpsoc reconfiguration", in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 293–300. (Cited on pages 5 and 51.)

[29] Emanuele Cannella, Onur Derin, Paolo Meloni, Giuseppe Tuveri, and Todor Stefanov, "Adaptivity support for mpsocs based on process migration in polyhedral process networks", *VLSI Design*, vol. 2012, pp. 2, 2012. (Cited on pages 4, 6, 51 and 52.)

[30] E.F. Deprettere, T. Stefanov, S.S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary parameterized dataflow graph

counterparts", in *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, Sept 2006, pp. 186–190. (Cited on page 52.)

[31] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov, "Pn: A tool for improved derivation of process networks", *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 19–19, Jan. 2007. (Cited on page 53.)

[32] L. Gantel, S. Layouni, M. Benkhelifa, F. Verdier, and S. Chauvet, "Multiprocessor task migration implementation in a reconfigurable platform", in *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*. IEEE, 2009, pp. 362–367. (Cited on pages 8, 54 and 67.)

[33] 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*. IEEE, 2005, pp. 234–239. (Cited on pages 8, 54 and 67.)

[34] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors", in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462. (Cited on page 55.)

[35] S. Sinha, M. Koedam, R. Van Wijk, A. Nelson, A.B. Nejad, M. Geilen, and K. Goossens, "Composable and predictable dynamic loading for time-critical partitioned systems", in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 285–292. (Cited on page 55.)

[36] H. Jiang and V. Chaudhary, "Process/thread migration and checkpointing in heterogeneous distributed systems", in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. 2004, pp. 10–pp, IEEE. (Cited on pages 57, 61, 62, 63 and 123.)

[37] B. Dimitrov and V. Rego, "Arachne: A portable threads system supporting migrant threads on heterogeneous network farms", *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 5, pp. 459–469, 1998. (Cited on pages 57 and 63.)

[38] E. Mascarenhas and V. Rego, "Arachne: architecture of a portable thread system supporting mobile processes.", Tech. Rep. CSD-TR 95-017, Purdue University, 1995. (Cited on page 57.)

[39] G. Antoniu, L. Bouge, and R. Namyst, "An efficient and transparent thread migration scheme in the pm2 runtime system", *Parallel and Distributed Processing*, pp. 496–510, 1999. (Cited on pages 58, 60 and 63.)

[40] S. Milton, "Thread migration in distributed memory multicomputers.", Tech. Rep. TR-CS-98-01, The Australian National University, 1998. (Cited on page 58.)

[41] G. Antoniu and C. Perez, "Using preemptive thread migration to load-balance data-parallel applications", *Euro-Par'99 Parallel Processing*, pp. 117–124, 1999. (Cited on pages 58 and 63.)

[42] H. Jiang and V. Chaudhary, "Compile/run-time support for thread migration", in *Proceedings of* 16^{th} *International Parallel and Distributed Processing Symposium*, 2002, pp. 58–66. (Cited on pages 62 and 63.)

[43] H. Jiang and V. Chaudhary, "Thread migration/checkpointing for type-unsafe c programs", *International Conference on High Performance Computing*, vol. 2913, no. 5, pp. 469–479, 2003. (Cited on page 63.)

[44] Fabrice Bellard, "Qemu, a fast and portable dynamic translator.", in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46. (Cited on page 90.)

[45] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot, "Using binary translation in event driven simulation for fast and flexible mpsoc simulation", in *Proceedings of the* 7^{th} *IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2009, pp. 71–80. (Cited on pages 14 and 90.)

[46] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P.S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Quong: A gpu-based hpc system dedicated to lqcd computing", in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, July 2011, pp. 113–122. (Cited on pages 14 and 90.)

[47] R Ammendola, A Biagioni, O Frezza, A Lonardo, F Lo Cicero, P S Paolucci, D Rossetti, F Simula, L Tosoratto, and P Vicini, "Apenet+ 34 gbps data transmission system and custom transmission logic", *Journal of Instrumentation*, vol. 8, no. 12, pp. C12022, 2013. (Cited on page 90.)

[48] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management", in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 1995, SOSP '95, pp. 251–266, ACM. (Cited on page 91.)

[49] Edward Lee, David G Messerschmitt, et al., "Synchronous data flow", *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. (Cited on page 92.)