COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

CENTRALE NANTES

MÉMOIRE DE DOCTORAT DE

L'ÉCOLE CENTRALE DE NANTES

ÉCOLE DOCTORALE N° 641
Mathématiques et Sciences et Technologies du numérique, de l'Information et de la Communication
Spécialité : *Automatique, productique et robotique*

Par
Antoine BERNABEU

Support d'Exécution pour les Systèmes Intermittents

Projet de recherche doctoral présenté et soutenu à Centrale Nantes, le 15 décembre 2023
Unité de recherche : UMR 6004, Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

Daniel CHILLET	Professeur des Universités, Université de Rennes
Emmanuel GROLLEAU	Professeur des Universités, ISAE-ENSMA

Composition du Jury :

Présidente :	Hanna KLAUDEL	Professeure des Universités, Université Paris-Saclay
Examinateurs :	Daniel CHILLET	Professeur des Universités, Université de Rennes
	Emmanuel GROLLEAU	Professeur des Universités, ISAE-ENSMA
Dir. de recherches doctorales :	Olivier H. ROUX	Professeur des Universités, École Centrale de Nantes
Co-enc. de recherches doctorales :	Sébastien FAUCOU	Maître de Conférences, Nantes Université
Co-enc. de recherches doctorales :	Mikaël BRIDAY	Maître de Conférences, École Centrale de Nantes

Remerciements

Les travaux présentés dans ce mémoire ont été effectués au sein du laboratoire des sciences du numérique de Nantes (LS2N). Je remercie le personnel du laboratoire pour m'avoir accueilli.

Je remercie particulièrement mes encadrants de thèse, Olivier Roux, Mikaël Briday, Sébastien Faucou ainsi que Jean-Luc Béchennec pour leurs implications dans mes travaux mais également pour leurs disponibilités. Nos discussions m'ont beaucoup appris sur les systèmes informatiques à plusieurs niveaux, que ce soit matériel ou logiciel.

Je voudrais adresser mes remerciements à Daniel Chillet et Emmanuel Grolleau pour avoir accepté de rapporter mes travaux. Je remercie également Hanna Klaudel pour avoir accepté d'évaluer cette thèse.

Je tiens également à remercier Vincent Lostanlen pour son implication dans une aventure mêlant plusieurs disciplines et qui je l'espère sera poursuivie et aboutira à de merveilleux travaux. Merci à Laurent Godet de m'avoir fait découvrir la discipline de l'écoacoustique.

Je remercie tous les membres de l'équipe Temps-Réel pour leurs retours sur mes travaux mais également pour les nombreuses discussions autour d'un café.

Un merci également aux membres de l'équipe Signal Image et Son du LS2N pour leurs entrain, et aussi pour leur efficacité à créer des évènements festifs.

Un grand merci aux doctorants du LS2N que j'ai côtoyés pour toutes ces pauses et ces moments de convivialité, trop nombreux pour être cité un par un, je vous fais un grand merci groupé.

Enfin, merci à mes parents et à mon frère pour leurs soutiens.

SOMMAIRE

TABLE DES FIGURES

INTRODUCTION

Depuis les années 2010, le nombre d'ordinateurs connectés a considérablement augmenté et intègre des objets de plus en plus petit, en passant par les téléphones portables jusqu'à des systèmes de taille millimétrique. De plus en plus d'objets auxquels sont ajoutés des capacités de calcul et de communication rentrent dans ce champ d'interconnexion. On appel cela l'Internet des Objets (IoT). Ce réseau d'appareils multiplie les applications possibles et les services disponibles. Les objets du quotidien comme les habits et l'électroménager sont ainsi instrumentés. Tous ces objets sont principalement contraints par leur encombrement ce qui engendre d'autres contraintes telles que leur consommation énergétique ou la qualité de service de l'application. Pour les objets placés dans des zones difficiles d'accès ou dans un environnement hostile, ils sont généralement alimentés par des piles ou des batteries.

Les piles et les batteries impliquent le remplacement manuel périodique de ces dernières pour maintenir le fonctionnement des objets alimentés à long terme. Cependant, lorsque le réseau de capteurs s'étend à des milliers de nœuds, la maintenance devient une tâche fastidieuse et coûteuse en main d'œuvre.

Réduire la consommation de piles et de batteries est un enjeu majeur. La consommation de pile et de batteries liée à l'IoT sera suffisante en 2025 (on prédit une consommation aux alentours de 46 TW) pour alimenter en électricité des pays [LA19]. Par exemple, 46 TW correspond à la consommation en électricité du Portugal en 2015.

Ainsi, la collecte d'énergie issue de l'environnement du système est couramment employée pour prolonger sa durée de vie, augmentant ainsi les périodes de fonctionnement avant le remplacement des batteries. Cependant, la récolte d'énergie reste par nature non prédictible et cela rend l'alimentation du système instable.

Cela a donné naissance à des paradigmes de traitement informatique pilotés par l'énergie, appelés *energy-driven computing* [Sli+20a]. Ainsi avec les technologies de récolte d'énergie, il est possible d'alimenter une partie des objets de l'IoT, ou du moins d'étendre la durée de vie des unités de stockage d'énergie.

Dans ce contexte, il est possible de se passer de piles et de batteries pour alimenter les systèmes IoT notamment grâce au développement des systèmes à très faible consom-

mation d'énergie et également à celui d'unités de stockage d'énergie alternatives tel que les supercondensateurs. Sans unité de stockage d'énergie, la consommation d'énergie du système alimenté doit à tous moments être inférieure ou égale à la récolte d'énergie. Cette contrainte entrave le développement d'applications qui nécessitent beaucoup de temps car à chaque perte d'alimentation, l'état volatil de l'application est perdu.

Pour contourner cette contrainte, le paradigme de calcul intermittent émerge en acceptant de prendre en compte les dynamiques de la récolte d'énergie et donc de perdre l'alimentation de manière non prédictible. Pour cela, un condensateur (et dans la plupart des cas, un supercondensateur) est utilisé pour stocker une petite quantité d'énergie afin de permettre au système d'effectuer des calculs. En conservant la progression des calculs à travers les cycles d'alimentation, les systèmes intermittents permettent le développement de toutes sortes d'application [Win+20]. La contrepartie de ce paradigme est que la qualité de service est sacrifiée car par nature, le système n'a pas d'alimentation continue. Prendre en compte les pertes d'alimentation n'est pas naturel lors de la conception d'application embarquée. Par définition, le programme peut s'arrêter à tout moment, laissant aux concepteurs un grand nombre de possibilités à gérer. Les outils classiques de conception ne sont pas adaptés à ces problématiques.

Ainsi, plusieurs approches peuvent être envisagées comme l'utilisation de langage de programmation dédiés, ou de composants matériels spécifiques. Les travaux de cette thèse se concentrent sur la modélisation de ce type de système et sur la conception d'un support d'exécution intégrant la connaissance de la consommation d'énergie afin d'apporter une solution à la gestion d'applications sous le paradigme intermittent.

Cas d'étude

Notre travail s'inscrit dans une démarche de collaboration avec d'autres équipes de recherche, notamment liées au traitement du signal et à l'étude de la biodiversité. Ainsi, le développement d'un capteur éco-acoustique servira de démonstrateur pour les méthodes développées dans ce manuscrit. Les capteurs sonores sont le bon niveau de complexité, l'application embarquée collecte des données afin d'avoir une vision statistique de l'environnement sonore, une exécution continue n'est pas nécessaire. De plus, les environnements étudiés sont potentiellement difficile d'accès, et la présence de l'homme peut perturber l'écosystème, ce qui est un argument pour limiter l'utilisation de batteries..

Contributions

La présente thèse aborde la problématique de la modélisation d'un système intermittent, en partant de la modélisation de sa consommation d'énergie et de l'application qu'il embarque, pour aller jusqu'à l'intégration de ces différents modèles dans un prototype fonctionnel d'exécution intermittente.

Notre modélisation d'un système intermittent est basée sur des méthodes formelles, et plus particulièrement sur les modèles de réseaux de Petri temporels à coût introduits par Boucheneb [Bou+17]. Notre modélisation étend ces travaux en apportant des éléments pour répondre au problème d'ordonnancement sous contrainte. Ainsi, nous intégrons la notion de progrès d'une application dans ce modèle afin d'optimiser l'avancement d'une application intermittente sous contrainte d'énergie.

Le modèle de consommation d'énergie utilisé dans cette modélisation repose sur les propriétés physiques des composants du système.

L'intégration de ces deux modèles dans un support d'exécution fournit une solution viable pour relever le défi de l'ordonnancement des tâches dans les systèmes à contraintes énergétiques. Nous proposons d'appliquer cette approche spécifiquement aux systèmes intermittents afin d'optimiser les opérations de sauvegarde nécessaires à la bonne exécution des applications. Des approches algorithmiques exactes et heuristiques sont proposées pour résoudre le problème d'ordonnancement sous contraintes.

Nous avons mis en œuvre la stratégie d'ordonnancement ainsi que le support d'exécution intermittent dans un système d'exploitation temps-réel.

Enfin, nous avons étudié l'interaction de l'approche intermittente avec des algorithmes de traitement audio et nous avons réalisé une preuve de concept d'un capteur éco-acoustique autonome.

Organisation du manuscrit

Le manuscrit est constitué de 5 chapitres. Le chapitre 1 présente les architectures matérielle et logicielle des systèmes étudiés ainsi que le modèle de consommation d'énergie utilisé par la suite.

Le chapitre 2 introduit le problème d'ordonnancement sous contrainte d'énergie. Une modélisation des systèmes intermittents à l'aide de réseaux de Petri temporisés à coût est utilisée afin d'apporter une solution au problème d'ordonnancement sous contrainte d'énergie. La sémantique du modèle est présentée ainsi que les algorithmes aboutissant à

une solution au problème proposé.

Dans le chapitre 3, nous proposons un support d'exécution pour les systèmes intermittents avec un modèle de calcul. La contribution proposée utilise la modélisation et les algorithmes présentés au chapitre précédent. Une implémentation sur une cible réelle est présentée et nous comparons les résultats avec des supports d'exécution intermittent de l'état de l'art.

Le chapitre 4 introduit le contexte de l'étude de cas proposée, celle d'un capteur intermittent éco-acoustique. Les notions de base sur l'étude de la biodiversité, le traitement des signaux audio et les algorithmes utilisés sont présentés.

Enfin, le chapitre 5 présente le prototype de capteur éco-acoustique que nous avons créé, incluant le support d'exécution intermittent et les algorithmes de traitement audio présentés.

ÉTAT DE L'ART

Modèle d'exécution intermittent

Dans cette section, nous présentons les différentes approches utilisées dans l'état de l'art pour prendre en charge le paradigme intermittent.

Deux principales approches ont été étudiées pour permettre une exécution intermittente sur des systèmes à faibles consommation d'énergie.

La première consiste à insérer des points de sauvegarde (ou checkpoints) dans l'application pour sauvegarder l'état volatil du système. La deuxième approche se base sur un modèle d'exécution transactionnel où l'application est décomposée en tâches atomiques.

Approches reposants sur les *checkpoints*

Mementos [RSF11], un des premiers travaux dans ce domaine, est un support d'exécution pour les systèmes de l'ordre de grandeur des systèmes RFID. Mementos est une pile logicielle alliant une bibliothèque permettant de sauvegarder et de restaurer l'état du système et des optimisations lors de la compilation afin de placer des mesures d'énergie dans le code. Plusieurs méthodes d'instrumentation du code par le compilateur sont proposées afin de placer les points de sauvegarde à chaque itération de boucle, à chaque retour de fonction ou de manière périodique en utilisant un timer. Cependant, Mementos peut rencontrer le problème de cohérence mémoire lorsque l'on utilise des mémoire volatiles et non-volatiles à l'exécution. Ainsi, si l'état de la mémoire non-volatile est modifié après un point de sauvegarde mais avant une perte d'alimentation, il n'y a pas de mécanisme pour remettre la mémoire non-volatile à l'état exact lors du point de sauvegarde.

QuickRecall [Jay+15] se focalise sur les systèmes possédant uniquement de la mémoire non-volatile, en sauvegardant et restaurant les registres du CPU et ceux des périphériques du système. Cette technique rencontre le même problème que Mementos vis-à-vis de la cohérence de la mémoire non-volatile.

Hibernus [Bal+15] utilisent un comparateur afin de sauvegarder l'état du système uniquement lorsque la tension d'alimentation atteint un seuil bas. Par rapport à Mementos, cela permet de retirer toutes les mesures d'énergie placées lors de la compilation. Une

extension d'Hibernus [Bal+16] étend cette fonctionnalité en ajoutant une étape d'étalonnage afin d'apprendre le temps nécessaire et le coût énergétique de la sauvegarde de l'état du système. Cette étape permet de placer le seuil bas de tension au plus près du seuil d'extinction du système et donc de maximiser les périodes de calcul.

Chinchilla [ML18] place des points de sauvegarde statiques de manière conservative, c'est-à-dire afin de garantir la terminaison de chaque bloc de code entre deux sauvegardes, mais choisit lors de l'exécution d'inhiber ou non certains points de sauvegarde.

Ratchet [WH16] découpe l'application à la compilation en section atomique et ainsi, il ne nécessite aucune intervention du développeur ni aucun matériel spécifique. Ainsi une analyse du code permet de détecter les portions idempotentes et des checkpoints sont insérés entre elles.

TICS [Kor+20] permet aux développeurs de choisir entre un checkpointing périodique ou via un comparateur qui monitore la tension d'alimentation. Quand celle-ci passe sous un seuil, un checkpoint est réalisé. La particularité de TICS est d'intégrer des outils afin de gérer les exécutions sensibles au temps. Ainsi, il est possible d'annoter des fonctions afin de les exécuter seulement si une condition de temporalité est respectée. Par exemple, si une donnée est trop ancienne, la fonction peut ne pas s'exécuter.

Plus récemment, ImmortalThread [YCY] met en place un mécanisme de mini-checkpoint, le *program counter* est sauvegardé en mémoire non volatile après chaque opération. Ce modèle de programmation permet de développer des applications avec du multithreading.

Approches reposant sur les tâches

DINO [LR15] est un modèle d'exécution et de programmation reposant sur le principe *task-based*. L'application est découpée en tâches atomiques et une sauvegarde du système (registres du CPU, piles etc ...) est faites à la fin de chaque tâche. Un mécanisme de gestion des données est également intégré à DINO afin de garantir la cohérence des données en mémoire non-volatile. Aux frontières entre les tâches, les données non-volatiles utilisées dans la tâche sont inclues dans la sauvegarde afin de les rétablir si une perte d'alimentation survient avant la fin de la tâche.

Chain [CL16] nécessite également de définir des tâches pouvant être relançables et toujours cohérentes par rapport à la mémoire. Pour cela, les tâches ont accès uniquement à la mémoire volatile et un système de *commit* des données vers la mémoire non-volatile est introduit pour transmettre des données entre les tâches. Ainsi, à la fin de chaque tâches, seulement les données validées vers la mémoire non-volatile sont enregistrées. Il

n'est plus nécessaire de sauvegarder la majorité de la mémoire volatile.

Mayfly [HSS17] étend ce modèle d'exécution en ajoutant des contraintes sur le temps d'exécution des tâches. Le développeur de l'application utilisant Mayfly peut décrire des contraintes temporelles telles qu'un temps d'expiration pour des données récupérées ou un temps minimal entre deux échantillons. Cette extension nécessite du matériel supplémentaire afin de garder une traçabilité du temps lorsque la plateforme est éteinte. Pour cela, plusieurs techniques peuvent être utilisées comme l'utilisation d'une RTC ou en regardant la décroissance de la tension aux bornes de condensateurs [Hes+16]. Cependant, ces techniques ne sont efficaces que si la RTC reste alimentée ou que le temps d'extinction de la plateforme est de l'ordre de grandeur du temps nécessaire à la décharge du condensateur utilisé. De plus, les circuits utilisés pour le maintien de la base de temps doivent consommer un ordre de grandeur de moins d'énergie que la plateforme pour assurer un avancement dans l'application.

Alpaca [MCL17] suit le même principe que Chain, en définissant des tâches atomiques et idempotentes. L'idée principale d'Alpaca est de spécifier les données intéragissant entre les tâches, et ces données sont privatisées localement. À l'aide d'une analyse lors de la compilation, ces données sont détectées et si elles ont été modifiées à l'exécution, elles sont mis à jour dans la mémoire principale à la fin de chaque tâche.

Plus récemment, pour étendre le paradigme intermittent à d'autres applications, la nécessité d'avoir un modèle d'exécution réactif a été soulevée. InK [Yı+18] est un support d'exécution permettant de définir des tâches périodiques et des tâches réactives. InK exécute les tâches de manière gloutonne dès que l'énergie est disponible. CatNap [ML20] étend ce modèle réactif en gardant une partie de l'énergie disponible pour des tâches définies comme critiques afin d'assurer leur exécution. De plus, une partie de l'application peut être dégradée en période de faible récupération d'énergie afin de garantir une progression. REHASH [Bak+21] est une extension de InK reprenant le concept d'adaptation et de dégradation de la qualité de service de l'application. Des heuristiques basées sur les temps actifs et inactifs de la plateforme servent à conditionner la dégradation ou l'amélioration de la qualité de service de l'application. Cela nécessite d'implémenter plusieurs versions des algorithmes utilisés.

Enfin, Sytare [Ber+19] est une couche logicielle qui gère les périphériques asynchrones pour les systèmes intermittents. Cette couche logicielle entre le noyau et l'application traque le changement d'état des périphériques et permet de remettre les périphériques dans l'état où ils étaient lors d'une perte d'alimentation.

Modélisation de systèmes intermittents

Les travaux initiaux sur la modélisation de systèmes intermittents se sont focalisés sur les aspects fonctionnels avec pour objectif de formaliser la preuve de l'exactitude de l'exécution [Ber+20b ; SLJ20]. Ces travaux ont pour but de prouver que les mécanismes de sauvegarde et de restauration permettent de maintenir un état cohérent du système (mémoires, périphériques) de sorte à pouvoir établir une équivalence entre l'exécution intermittente et l'exécution continue du système. Dans tous les cas, les ressources physiques telles que les ressources de calcul, le temps et l'énergie sont soit ignorées, soit modélisées de manière très abstraite. En dehors du cadre de la modélisation formelle, plusieurs travaux proposent des solutions pour simuler une exécution intermittente avec un focus sur l'énergie. Certains d'entre-eux sont orientés sur la prédiction statique de la pire consommation d'énergie d'une exécution [Wäg+18], alors que d'autres comme *Fused* [Sli+20b], sont plus orientés sur la simulation. Les modèles de simulation sont très détaillés mais sont également très complexes, les rendant inutilisables pour l'exploration de l'espace des solutions lors de la conception.

Pour développer des modèles plus haut-niveau, des plateformes de mesure de la consommation énergétiques des systèmes intermittents ont vu le jour [DAL18 ; Ber+20a]. Ces plateformes utilisent principalement la mesure de l'intensité électrique afin de calculer la consommation des composants du système. Elles ont pour hypothèse que la tension d'alimentation est constante ce qui nécessite un régulateur de tension entre le supercondensateur et le MCU.

Plateforme intermittente

Différentes plateformes intermittentes ont été développées en parallèle des travaux sur les supports d'exécution. WISP [Sam+08] est l'une des premières plateformes à voir le jour. De la taille d'une pièce de monnaie, elle utilise de la récolte d'énergie par ondes radioélectriques via une antenne. WISP utilise le système RFID (Radio Frequency IDentification) pour transmettre des données, ce qui limite les applications car la portée des transmissions est relativement faible, de l'ordre de la dizaine de centimètre. Flicker [HS17] est une plateforme intermittente modulaire incorporant de nombreuses fonctionnalités. Il est possible de combiner plusieurs sources d'énergie pour la récolte et ils utilisent une approche de stockage d'énergie fédérée, c'est-à-dire que chaque composant consommant de

l'énergie utilise son condensateur personnel. Cette approche modulaire rend la plateforme extensible mais le coût de fabrication est relativement élevé, environ 200$. À la différence des travaux précédents, Capybara [CRL18] se distingue par une approche conjointe matérielle/logicielle. Ils utilisent également une approche fédérée pour le stockage d'énergie, avec la particularité de pouvoir déconnecter certaines capacités pour rendre le système plus réactif. Camaroptera [Nar+19] est une plateforme accueillant une caméra et un composant LoRa pour l'envoi de données. La récolte d'énergie ne se fait que via des panneaux solaires et un seul supercondensateur est utilisé pour tout le système. Plus récemment, Protean [Bak+22] reprend l'approche modulaire de Flicker en l'améliorant avec la possibilité d'utiliser n'importe quelle microcontrôleur (MCU) grâce à l'ajout d'une interface standard M.2. Cela permet d'utiliser des architectures plus complexes et d'augmenter les capacités de calcul du système.

Plateforme de surveillance acoustique

Les plateformes de surveillance acoustique se sont largement développées récemment [Sug+19]. Contrairement aux comptages effectués par des opérateurs humains sur le terrain, elles permettent de surveiller une plus large zone, à n'importe quelle heure, que la zone soit difficile d'accès ou non, et cela avec un impact moindre sur les environnements étudiés. Les plateformes les plus utilisées par les bioacousticiens sont les plateformes Song-Meter, Audiomoth et le Swift Recorder [TR21]. Leurs prix varient de 60 $ à 1239 $. Malgré une baisse progressive des prix de ces systèmes ainsi que leur miniaturisation, les besoins des chercheurs ne sont pas toujours satisfaits et cela mène au développement de capteurs spécifiques [Kwo17].

Plus récemment un capteur acoustique pour la surveillance de sons urbains alimenté par un panneau photovoltaïque a été développé [Yun+22]. Ce capteur intègre l'autonomie calculatoire et énérgétique, mais repose sur des batteries enfreignant ainsi l'autonomie opérationnelle présentée en section 4.1.1.

Première partie

Modélisation et conception d'un support d'exécution pour le calcul intermittent

Chapitre 1

Description du système

Ce chapitre présente la catégorie de système étudiée tout au long de ce manuscrit. Il est divisé en deux parties. La première partie décrit les différents composants d'un système sans batterie qu'ils soient logiciels ou matériels. Une attention particulière est portée à la description du support d'exécution dans le cadre du calcul intermittent.

Dans la deuxième partie, nous nous intéressons plus particulièrement à la modélisation de la consommation d'énergie pour ces systèmes sans batterie. À partir d'une étude des composants électroniques qui constituent un système sans-batterie et d'expérimentations, un modèle linéaire de consommation d'énergie est proposé pour décrire la consommation du système.

1.1 Systèmes intermittents

Nous avons introduit précédemment les systèmes intermittents comme étant des systèmes sans batterie sous le paradigme de calcul intermittent.

On peut parler de système intermittent, tant pour la partie matérielle que pour la partie logicielle du système. Nous proposons de donner une définition pratique des différents composants qui constituent un système intermittent.

1.1.1 Composants matériels

La récolte d'énergie

Les systèmes sans batterie sont alimentés via la récolte d'énergie issue de l'environnement du système. Plusieurs catégories d'énergie peuvent être récoltées et transformées en énergie électrique utilisable pour le fonctionnement du système.

L'intensité lumineuse : La collecte d'énergie à partir de la lumière consiste à convertir la lumière du soleil en énergie électrique utilisable. Elle est également appelée énergie photovoltaïque ou solaire. L'énergie de la lumière est généralement extraite à

l'aide de cellules photovoltaïques qui s'appuient sur l'effet photoélectrique pour éjecter des électrons des cellules vers le circuit électronique qu'elles alimentent. La puissance de sortie produite par une cellule photovoltaïque dépend de l'intensité de la lumière, de la taille de la cellule et de son efficacité conformément au principe photovoltaïque [Mil+16]. Il convient de noter que la lumière du soleil n'est pas la seule source d'énergie, car la lumière de l'éclairage électrique peut également être récoltée, par exemple la lumière d'une diode électroluminescente (LED). La lumière a une densité de puissance comprise entre $15\,\mathrm{mW\,cm^{-2}}$ et $100\,\mathrm{mW\,cm^{-2}}$ [Hab+17 ; SK11 ; Pra+18].

Ondes radioélectrique : Les ondes radios, qui sont des ondes électromagnétiques avec une fréquence inférieure à 300 MHz peuvent également être utilisées pour de la récolte d'énergie. L'utilisation de l'énergie issue d'ondes radioélectriques n'est pas nouvelle [Bro69] mais les contraintes des technologies utilisées en terme d'espace n'étaient pas compatibles avec une utilisation pour des capteurs embarqués. Pour récolter l'énergie issue d'ondes radioélectriques, une antenne est nécessaire afin de canaliser le signal et permet de produire un signal électrique utilisable de même fréquence. La densité de puissance est de l'ordre de $0.2\,\mathrm{nW\,cm^{-2}}$ à $1\,\mathrm{\mu W\,cm^{-2}}$ [Kim+14 ; Yil09].

Vibrations et énergie mécanique : Les vibrations peuvent être une source d'énergie utilisable pour des systèmes embarqués. Pour se faire, la conversion de l'énergie issue de vibrations en énergie électrique se fait en deux étapes. La première, consiste à utiliser les vibrations afin de mettre en mouvement deux objets l'un par rapport à l'autre à l'aide d'un système de masse-ressort. C'est ce mouvement relatif qui est ensuite converti en énergie électrique. Cette dernière conversion peut se faire à l'aide de différents éléments, soit un matériau piézoélectrique, une bobine magnétique ou un condensateur variable [WJ17]. Pour la conversion à l'aide d'un matériau piézoélectrique, la nature de ce matériau produit une charge électrique lorsqu'il subit une déformation. Pour la conversion à l'aide d'une bobine magnétique, une force électromotrice est générée par le mouvement relatif entre une bobine et un aimant ; cette force électromotrice est régie par la loi de Lenz-Faraday. Enfin pour la conversion à l'aide d'un condensateur variable, le mouvement des deux objets va modifier la structure d'un condensateur (ses armatures conductrices) afin de générer une charge électrique. La densité de puissance varie de $3.8\,\mathrm{\mu W\,cm^{-2}}$ à $500\,\mathrm{\mu W\,cm^{-2}}$ [Cal+05] en fonction de la conversion mécanique/électrique utilisée.

Température : Les variations spatiales et temporelles de la température, via l'effet Seebeck, peuvent faire apparaître une différence de potentiel à la jonction de deux matériaux. Cette source d'énergie peut être récoltée et la densité de puissance varie en fonction

des matériaux utilisés et de la différence de température. Pour un écart de 5 °C, la densité de puissance peut varier de $0.2\,\mu W\,cm^{-2}$ à $53\,\mu W\,cm^{-2}$ [Bot+04].

Tampon d'énergie

Les systèmes sans batterie possèdent tout de même un élément permettant le stockage d'une quantité d'énergie utilisable dans un futur proche. La plupart du temps, cela se traduit par un élément capacitif. L'intérêt principal de cet élément de stockage est de découpler l'énergie issue de la récolte de l'énergie utilisée par le système. Sans ce stockage, il faudrait à tout moment équilibrer la récolte d'énergie et la consommation du système, ce paradigme est appelé *power-neutral* [Sli+20a].

Le développement des supercondensateurs, et notamment des supercondensateurs à double couches permet d'obtenir des condensateurs avec de très grandes capacités relativement à la taille du composant. Ces condensateurs à double couche ont une plus grande densité énergétique que les condensateurs traditionnels grâce à la maximisation de la surface des armatures et de la très faible distance qui les sépare [Raz+18].

On peut noter également le développement de techniques permettant de créer ces supercondensateurs en utilisant des matériaux biodégradables pouvant être éliminés en toute sécurité pour à la fois l'environnement et les agents [Dya+13].

Mémoire non volatile (NVM)

Sur un système intermittent, comme le système redémarre souvent en raison d'interruptions de courant, une mémoire non volatile (NVM) est nécessaire pour suivre l'état du système et assurer la continuité de l'application. La sauvegarde de l'état du système nécessite une mémoire non volatile qui soit à la fois rapide, *i.e.* au niveau de l'ordre de grandeur du temps de cycle du processeur, tout en ayant une faible consommation d'énergie. Le type de mémoire non volatile le plus courant est la mémoire Flash. La mémoire flash offre une solution pour une mémoire à faible coût et à grande capacité, mais elle souffre de lenteurs et d'un manque d'efficacité énergétique lors des accès en écriture (en comparaison avec la mémoire vive statique (SRAM)). De plus, elle supporte un nombre réduit d'écritures, de l'ordre de 10^5 [Bou+18], ce qui la rend inadaptée à une utilisation dans le contexte des systèmes intermittents, car la sauvegarde de l'état du système peut se produire plusieurs fois par seconde.

D'autres types de mémoires non volatiles sont apparus ces dernières années [Mee+14]. Cela inclut la ferroelectric random access memory (FRAM) [Ish12] aussi appelée FeRAM,

qui a atteint un niveau de maturité suffisant pour être disponible dans certains microcontrôleurs [Fox+04]. La FRAM semble favorable dans le contexte intermittent par rapport à la mémoire Flash en étant plus rapide : temps d'accès en lecture/écriture de 75 ns contre 200–500 µs et en offrant une endurance beaucoup plus élevée que la mémoire Flash : elle permet $10^{15} \sim 10^{16}$ cycles de lecture-écriture, d'où une durée de vie exprimée en décennies [Bou+18]. Dans une certaine mesure, la FRAM peut être utilisée comme mémoire de travail de la même manière que la SRAM.

Unités de calcul et périphériques

L'unité de calcul d'un système intermittent fait généralement référence à une unité centrale de calcul (CPU) qui gère l'exécution des instructions, effectue les calculs et gère le flux de données au sein du système. Un périphérique désigne tout composant matériel permettant au système d'effectuer des opérations spécifiques. Les périphériques offrent des fonctionnalités supplémentaires au système et lui permettent d'interagir avec son environnement. Nous ajoutons également aux périphériques, toute partie matérielle qui accélère un calcul spécifique, comme les coprocesseurs ou par exemple les processeur de signal numérique (DSP) qui accélèrent le traitement de donnée audio ou vidéo. On parle de microcontrôleur (MCU) lorsqu'au sein du même circuit intégré (IC) sont réunis les éléments suivants :

— au moins un CPU,
— de la mémoire volatile et/ou non volatile,
— des entrées/sorties,
— des périphériques.

Les MCU sont principalement conçus pour effectuer des tâches spécifiques au sein d'un système complexe. Par exemple, une voiture possède des dizaines de MCU pour gérer différentes opérations telles que le contrôle moteur ou l'ABS. Concernant les MCU, on différencie les périphériques internes (physiquement dans le circuit intégré) des périphériques externes qui eux ne sont pas dans le circuit intégré, qui peuvent être ajoutés ou enlevés et qui communiquent avec le MCU via ses entrées/sorties.

1.1.2 Composants logiciels

Pour la partie logicielle, nous nous proposons de décrire les composantes liées d'une part au support d'exécution et d'autre part à l'application.

Le support d'exécution intermittent

Un support d'exécution intermittent désigne un environnement logiciel spécialisé conçu pour faciliter l'exécution de programmes ou d'applications sur un système intermittent. Comme pour les supports d'exécution sur les systèmes alimentés en continu, les éléments classiques comme par exemple les éléments pour la gestion de l'exécution, la gestion de l'accès aux ressources partagées et la gestion des erreurs sont présents. Pour un support minimal de l'intermittence sur une architecture comportant de la mémoire volatile et non volatile, le support d'exécution doit au moins fournir les éléments pour la sauvegarde de l'état avant une interruption de l'alimentation et sa restauration lorsque l'alimentation reprend. Plus récemment, les problèmes d'incohérence de la mémoire non volatile [RL14] ont été soulevés. Afin de décharger le développeur de la tâche complexe de cette gestion de cohérence mémoire, un support d'exécution intermittent doit intégrer les éléments afin de s'en assurer. Pour aller plus loin, l'optimisation de la consommation d'énergie, mais aussi la prédiction de la récolte peuvent être intégrés dans le support d'exécution afin d'améliorer les performances en termes de qualité de service.

L'application

Nous appelons application la partie logicielle spécifique fournit un service en particulier pour répondre aux besoins de l'utilisateur. Une application utilise les ressources et les services fournis par le support d'exécution. Prenons l'exemple d'une application de détection et de classification du chant des oiseaux. Un certain nombre de tâches sont nécessaires pour réaliser cette application, ce qui requiert diverses ressources. Tout d'abord, nous devons acquérir le signal sonore ambiant, puis traiter ce signal pour en extraire les caractéristiques associées aux oiseaux, et enfin déterminer à quelle espèce appartient ce chant et envoyer le résultat à une station distante. Toutes ces tâches font partie de l'application et le support d'exécution est là pour fournir les services nécessaires à l'exécution correcte des tâches.

1.2 Modélisation de la consommation d'énergie

La consommation d'énergie est une mesure essentielle pour évaluer l'efficacité et la durabilité des systèmes informatiques. L'énergie représente un intrant indispensable pour les appareils électroniques et il y a une volonté constante de minimiser cette consommation

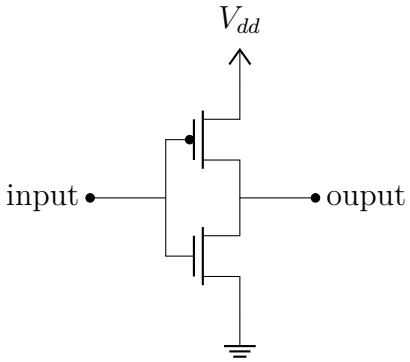
pour diverses raisons, notamment le dimensionnement, la longévité ainsi que des aspects financiers. La consommation d'énergie est d'autant plus importante pour les systèmes embarqués où la taille et le poids de l'ensemble du système sont des facteurs décisifs. Pour les systèmes sans batteries dans le contexte du calcul intermittent, minimiser la consommation d'énergie n'est pas nécessairement l'objectif car lorsque le tampon d'énergie (*i.e.* le supercondensateur) est plein, toute énergie récoltée sera perdue car non utilisée et non stockée. Ainsi, pour ces systèmes, on préfèrera parler d'efficacité de l'utilisation de l'énergie afin d'utiliser au maximum l'énergie récoltée, sans toutefois en gaspiller. Comme nous l'avons vu dans la section 1.1, la catégorie de système étudiée dans ce travail comprend deux grandes familles de composants : les composants fournissant de l'énergie au système, qui sont les parties de récolte et de stockage, et les composants utilisant cette énergie, qui sont les parties d'acquisition, de calcul et de transmission de données. Nous allons à présent détailler la consommation d'énergie d'un système intermittent.

1.2.1 Consommation d'énergie d'un microcontrôleur

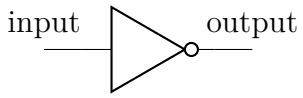
On se place dans le cas où notre système sans batterie est composé d'un MCU ainsi que de différents périphériques externes. Pour les applications que nous envisageons, ces MCU sont construits en utilisant la technologie Complementary Metal Oxide Semiconductor (CMOS). Nous supposons que tous les circuits électroniques utilisés dans les composants d'acquisition, de calcul et de transmission des données sont basés sur cette technologie. Cette hypothèse n'est pas très restrictive, car la technologie CMOS est la technologie dominante actuellement utilisée pour fabriquer des circuit intégré (IC). Plus de 95% des ICs sont fabriqués en utilisant la technologie CMOS [Bak19].

Fonctionnement d'un circuit CMOS : Un microcontrôleur est composé de portes logiques. Ces portes logiques sont connectées entre elles pour créer des fonctions logiques telles que de la mémoire ou des unités arithmétiques. Une porte logique en technologie CMOS consiste en un assemblage de transistors dont les grilles sont connectées aux signaux d'entrée. La figure 1.1 présente un inverseur comme exemple de porte logique. L'inverseur est composé de deux transistors CMOS et inverse la valeur logique de l'entrée.

Consommation statique La consommation d'énergie statique d'un circuit CMOS, également appelée consommation d'énergie en régime permanent, est due au courant qui circule entre l'alimentation et la masse. La figure 1.2 montre la représentation du circuit électrique pour la modélisation de la consommation d'énergie statique d'un inverseur basé sur la technologie CMOS. Une petite partie du courant passe par les diodes de



(a) Cicruit électrique d'un inverseur (fonction logique NON) avec des transistors CMOS



(b) Symbole de l'opérateur logique inverseur

FIGURE 1.1 – Représentation d'un inverseur en technologie CMOS et son symbole.

polarisation inverse des transistors (le courant noté i_{stat} sur la figure 1.2). Cette consommation statique est exprimée par l'équation (1.1). Pour le type de système considéré, la consommation d'énergie statique est très faible par rapport à la consommation d'énergie dynamique. Cependant, la consommation statique d'un MCU peut être estimée en mesurant la consommation d'énergie lorsqu'il est dans un état de faible consommation d'énergie avec la plupart de ses composants arrêtés (y compris les horloges) mais le CPU toujours alimenté.

$$P_{stat} = I_{stat} \times V_{dd} \tag{1.1}$$

Consommation dynamique La consommation d'énergie dynamique d'un circuit CMOS est étroitement liée à la capacité de charge due à la sortie de la porte logique (soit une autre porte logique, soit la sortie du circuit). Cette capacité est notée C_L. Nous allons réutiliser l'exemple de l'inverseur de la figure 1.3a. Pour l'inverseur, la capacité de charge emmagasine de l'énergie jusqu'a atteindre une tension égale à V_{dd} lorsque la sortie du circuit est à un niveau logique haut et se décharge lorsque la sortie est à un niveau logique bas. Cette consommation d'énergie peut être exprimée par l'équation (1.2) où f_{clk} est la fréquence de l'activité de commutation des transistors.

$$P_{dyn} = f_{clk} \times C_L \times V_{dd}^2 \tag{1.2}$$

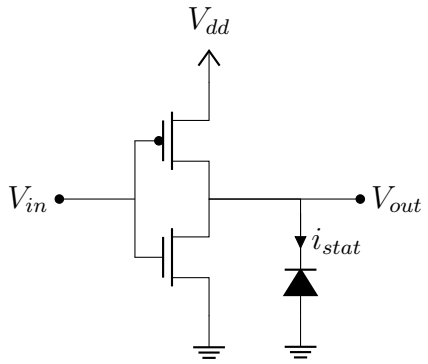
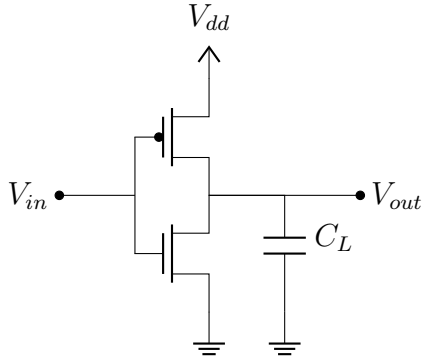
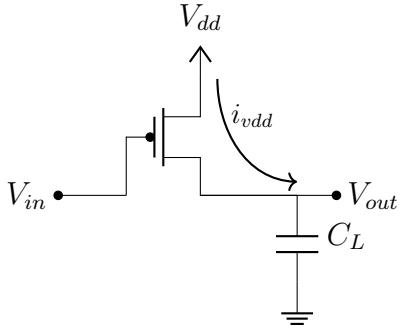


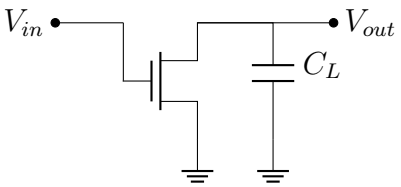
FIGURE 1.2 – Consommation statique d'un inverseur CMOS



(a) Circuit électrique d'un inverseur CMOS avec un condensateur de charge



(b) Circuit électrique équivalent pendant une transition niveau bas - niveau haut



(c) Circuit électrique équivalent pendant une transition niveau haut - niveau bas

FIGURE 1.3 – Consommation dynamique d'un inverseur CMOS

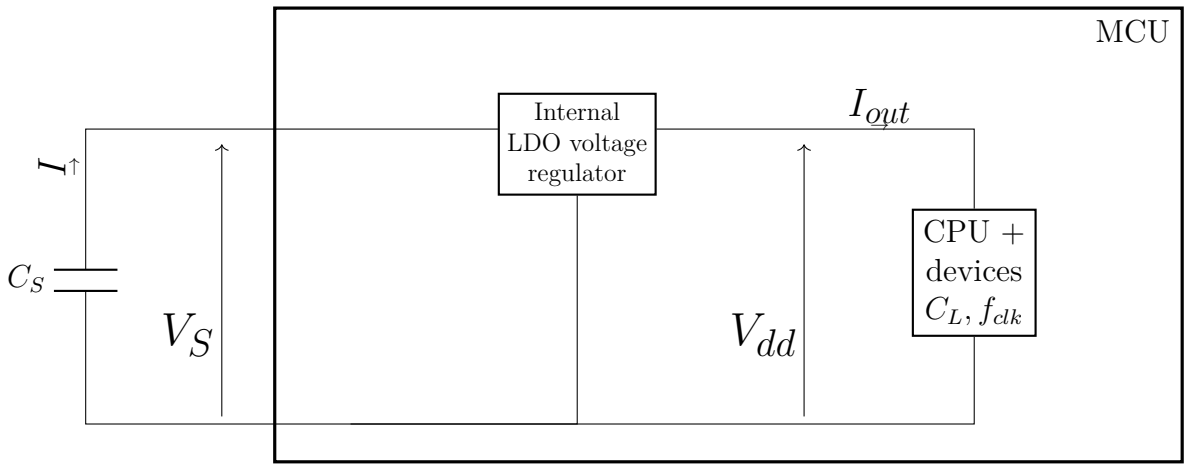


FIGURE 1.4 – Schéma d'un MCU alimenté directement par une supercapacité

1.2.2 Un modèle linéaire

Considérons un système tel que celui décrit dans la figure 1.4. Ici, le composant de stockage d'énergie désigné par C_s dans la figure est un condensateur (et potentiellement un supercondensateur). Le MCU est alimenté avec une tension V_{dd} par l'intermédiaire d'un régulateur à faible chute de tension (LDO) qui prend une tension V_s comme entrée. Généralement, ce LDO se trouve à l'intérieur du IC et ne peut donc pas être retiré. Comme nous ne nous intéressons qu'à la consommation d'énergie, les composants de récolte d'énergie du système n'entrent pas dans le champ d'application de notre modèle.

L'énergie fournie par la récolte d'énergie est stockée dans le supercondensateur. Cette énergie E est exprimée par l'équation (1.3) avec C_S la capacité de cette supercapacité et V_S la tension à ces bornes.

$$E = \frac{1}{2} \times C_S \times V_S^2 \tag{1.3}$$

Comme indiqué dans la Section 1.2.1, un composant de la partie utilisant l'énergie stockée dans le supercondensateur a une consommation d'énergie qui peut être décomposée en deux parties : une partie statique et une partie dynamique. Nous supposons que la consommation d'énergie statique de ces composants, qui sont fabriqués en utilisant la technologie CMOS, est négligeable par rapport à la consommation d'énergie dynamique. Cette hypothèse semble raisonnable car il y a jusqu'à 6 ordres de grandeur entre la consommation d'énergie dynamique et la consommation d'énergie statique en fonction de la technologie utilisée et de la température [KK15]. Avec cette hypothèse, la puissance

dissipée par un tel composant est définie par l'équation (1.4) avec f_{clk} la fréquence de commutation des capacités de charge, C_L les capacités de charge et V_{dd} la tension fournie.

$$P_{cmos} \simeq P_{dyn} = f_{clk} \times C_L \times V_{dd}^2 \tag{1.4}$$

Ces composants sont alimentés par un LDO, maintenant ainsi une tension constante aux transistors CMOS. Par conséquent, V_{dd} est constant dans le temps.

Cependant, le fonctionnement du LDO induit une dissipation de puissance P_{loss}, défini dans l'équation (1.5) avec V_s la tension d'entrée, V_{dd} la tension de sortie, I_{out} le courant de sortie et I_q le courant de repos qui est le courant consommé par le LDO pour son circuit de contrôle interne lorsqu'il est actif. Le courant de repos est défini comme la différence entre le courant d'entrée et le courant de sortie.

$$P_{loss} = (V_S - V_{dd}) \times I_{out} + V_S \times I_q \tag{1.5}$$

En négligeant le courant de repos I_q consommé par le LDO, nous pouvons approximer que $I = I_{out}$, donc la consommation d'énergie comme :

$$\begin{aligned} P_{total} &= P_{cmos} + P_{loss} \\ V_s \times I &= f_{clk} \times C_L \times V_{dd}^2 + (V_S - V_{dd}) \times I_{out} \\ I &= f_{clk} \times C_L \times V_{dd} \end{aligned} \tag{1.6}$$

comme le condensateur fournit le courant :

$$\frac{dV_S}{dt} = \frac{f_{clk} \times C_L \times V_{dd}}{C_s} \tag{1.7}$$

À partir de l'équation 1.7, la dynamique de variation de tension est caractérisée par des termes constants (V_{dd} et C_S) et des termes qui évoluent également dans le temps (f_{clk} et C_L).

On définit donc un mode de fonctionnement du système (que l'on appellera mode) comme une paire (f_{clk}, C_L) où f_{clk} est une fréquence fonctionnement et C_L est la somme des capacités des composants du circuit actif. Tant que le système reste dans le même mode, la décroissance de la tension aux bornes du condensateur alimentant le système est

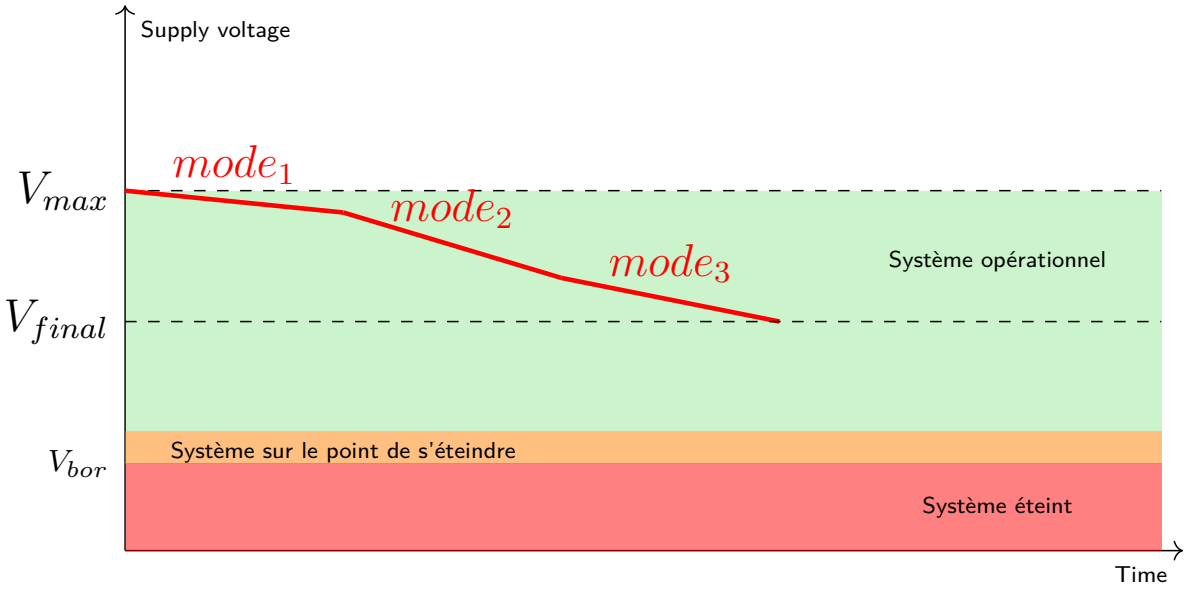


FIGURE 1.5 – Graphique représentant la fonction affine par morceau modélisant la tension d'alimentation du système intermittent. 3 modes sont exécutés en séquence. La tension disponible de départ est V_{max}, la tension disponible après l'exécution des 3 modes est V_{final}. V_{bor} représente le seuil de tension à partir duquel le système n'est plus alimenté.

directement proportionnelle au temps.

On peut donc suivre et anticiper la consommation d'énergie du système avec ce modèle linéaire par rapport au temps. En connaissant au préalable les différents modes, leur temps d'exécution et leur ordonnancement, la tension V_S est modélisée par une fonction affine par morceau sans aucun saut. La figure 1.5 illustre cela avec un exemple comportant trois modes successif et une tension de départ égale à V_{max}.

Modèle énergétique avec régulateur de tension

Il est courant que les systèmes embarqués soient alimentés par un régulateur de tension DC-DC en amont (*i.e.* un régulateur de tension externe au MCU). Son objectif est alors d'alimenter l'ensemble du système avec une tension constante prédéfinie. Tant que le régulateur de tension externe est capable de fournir le courant consommé par le système, la tension sera constante. Dans ce cas, le système peut être décrit comme illustré par la figure 1.6.

L'ajout d'un régulateur de tension externe peut être intéressant dans certains cas. À partir de l'équation (1.5) nous savons que la dissipation de puissance du LDO dépend de la différence entre V_S et V_{dd}. Cette dernière est une constante alors que V_S peut être

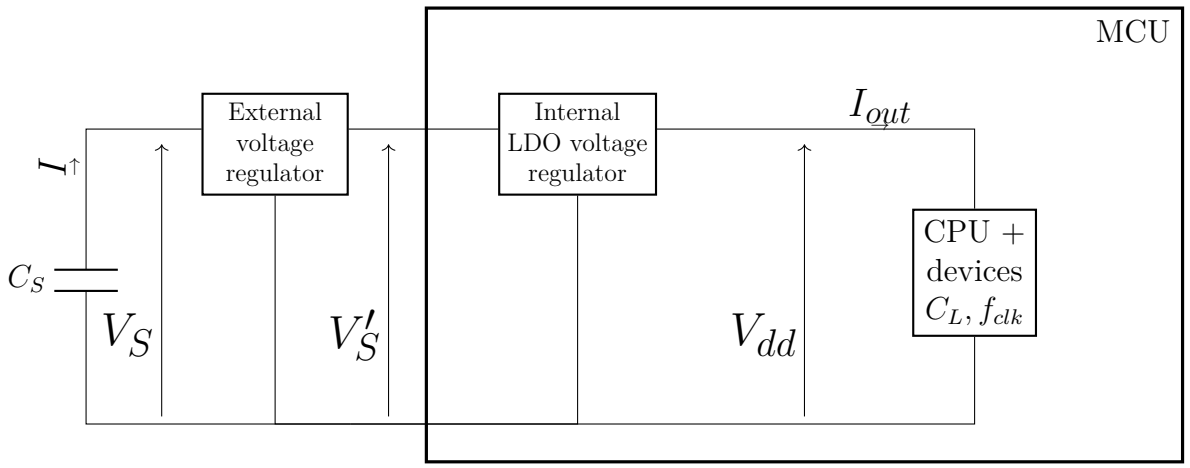


FIGURE 1.6 – Schéma d'un MCU alimenté par une supercapacité à travers un régulateur de tension

ajustée. Avec un régulateur de tension externe, l'entrée du LDO peut être fixée à une valeur très proche de V_{dd} afin de minimiser la dissipation de puissance du LDO. Cependant, cela modifiera le comportement du modèle de consommation d'énergie présenté précédemment. En se reportant à la figure 1.6, l'équation de la puissance dissipée par le LDO devient :

$$P_{loss} = (V'_S - V_{dd}) \times I_{out} + V_S \times I_q \tag{1.8}$$

en supposant que l'on souhaite minimiser la puissance dissipée par le LDO :

$$V_{dd} < V'_S \leq V_S$$

Les régulateurs de tension ont des performances qui varient en fonction du courant de sortie (*i.e.* le courant consommé par le système, I_{out} sur la figure 1.6). Le rendement d'un régulateur de tension noté η est le rapport entre la puissance de sortie et la puissance d'entrée.

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V'_S \times I_{out}}{V_S \times I}$$

La perte de puissance due au régulateur de tension externe peut être exprimée comme suit :

$$P_{lossExtReg} = V_S \times I - V'_S \times I_{out} = P_{in} \times (1 - \eta)$$

Il faut alors trouver un compromis entre la réduction de la dissipation de puissance du LDO interne au système et l'efficacité du régulateur de tension externe. Pour obtenir une efficacité maximale en utilisant un régulateur de tension externe, la tension de sortie V'_S doit être réglée sur la tension minimale requise par le système et la perte de puissance associée à l'ajout du régulateur doit être inférieure au gain sur la perte de puissance du LDO.

Avec l'ajout d'un régulateur de tension externe, la chute de tension aux bornes du supercondensateur est :

$$\begin{aligned} P_{total} &= P_{cmos} + P_{loss} + P_{lossExtReg} \\ V_S \times I &= f_{clk} \times C_L \times V_{dd}^2 + (V'_S - V_{dd}) \times I_{out} + (V_S \times I - V'_S \times I_{out}) \\ V_S \times \frac{dV_S}{dt} &= \frac{V'_S \times f_{clk} \times C_L \times V_{dd}}{C_S} \end{aligned} \tag{1.9}$$

La résolution de l'équation différentielle (1.9) donne une diminution de la tension aux bornes du supercondensateur comme une fonction racine carré par rapport au temps, en assumant un unique mode de fonctionnement. Pour se rapporter à un modèle linéaire comme en section 1.2.2, on peut utiliser comme variable non pas la tension aux bornes du tampon d'énergie alimentant le système mais la tension au carré.

1.2.3 Expérimentation

Pour valider le modèle de consommation d'énergie présenté en section 1.2.2, nous avons fait une campagne de mesure de la tension du supercondensateur alimentant le système pour différents modes de fonctionnement. Pour s'assurer que la mesure de la tension ne perturbe pas la mesure, nous avons utilisé une carte *ad hoc*. Ce circuit a pour mission de charger le supercondensateur à son maximum (on amène sa tension à 3.3 V) et de déconnecter toutes sources d'alimentation pour recharger le supercondensateur lors de la mesure (liaison système/hôte, debuggueur). Le schéma de ce circuit est présenté en annexe 5.3.

Pour notre système intermittent, nous avons utilisé le MSP430FR5994 [1]. Ce MCU de Texas Instrument peut être alimenté avec une tension comprise entre 1.8 V et 3.3 V. La carte d'évaluation MSP430FR5994-Launchpad intègre une supercapacité de 220 mF qui peut être utilisée pour alimenter le MCU. Cependant, cette supercapacité possède

1. `https://www.ti.com/tool/MSP-EXP430FR5994`

une résistance série électrique de 75 Ω, ce qui produit une chute de tension significative lorsque le système requiert un courant important. À titre d'exemple, la chute de tension peut atteindre jusqu'à 7.5 V lorsque le système requiert 100 mA ce qui peut être le cas lors de transmission radio. Pour limiter cette chute de tension, nous avons à la place utilisé une supercapacité avec une résistance électrique plus faible, de l'ordre de 1 Ω.

La figure 1.7 montre les différentes pentes de tension pour 3 modes.

1.3 Conclusion du premier chapitre

Dans ce chapitre, nous avons décrit les composants matériels et logiciels de la catégorie de système étudié, les systèmes sans batterie intermittents. À partir d'une analyse de la consommation électrique de ces systèmes, nous avons présenté une modèle de consommation d'énergie linéaire dans le cas d'une absence de régulateur de tension externe entre le supercondensateur et le système alimenté. Toutefois, si un régulateur de tension externe est présent, on peut toujours se ramener à un modèle linéaire en utilisant la tension au carré comme grandeur d'entrée. Ce modèle linéaire a été validé par une campagne expérimentale en utilisant des composants sur étagère pour le système intermittent et une carte *ad hoc* pour l'automatisation des mesures.

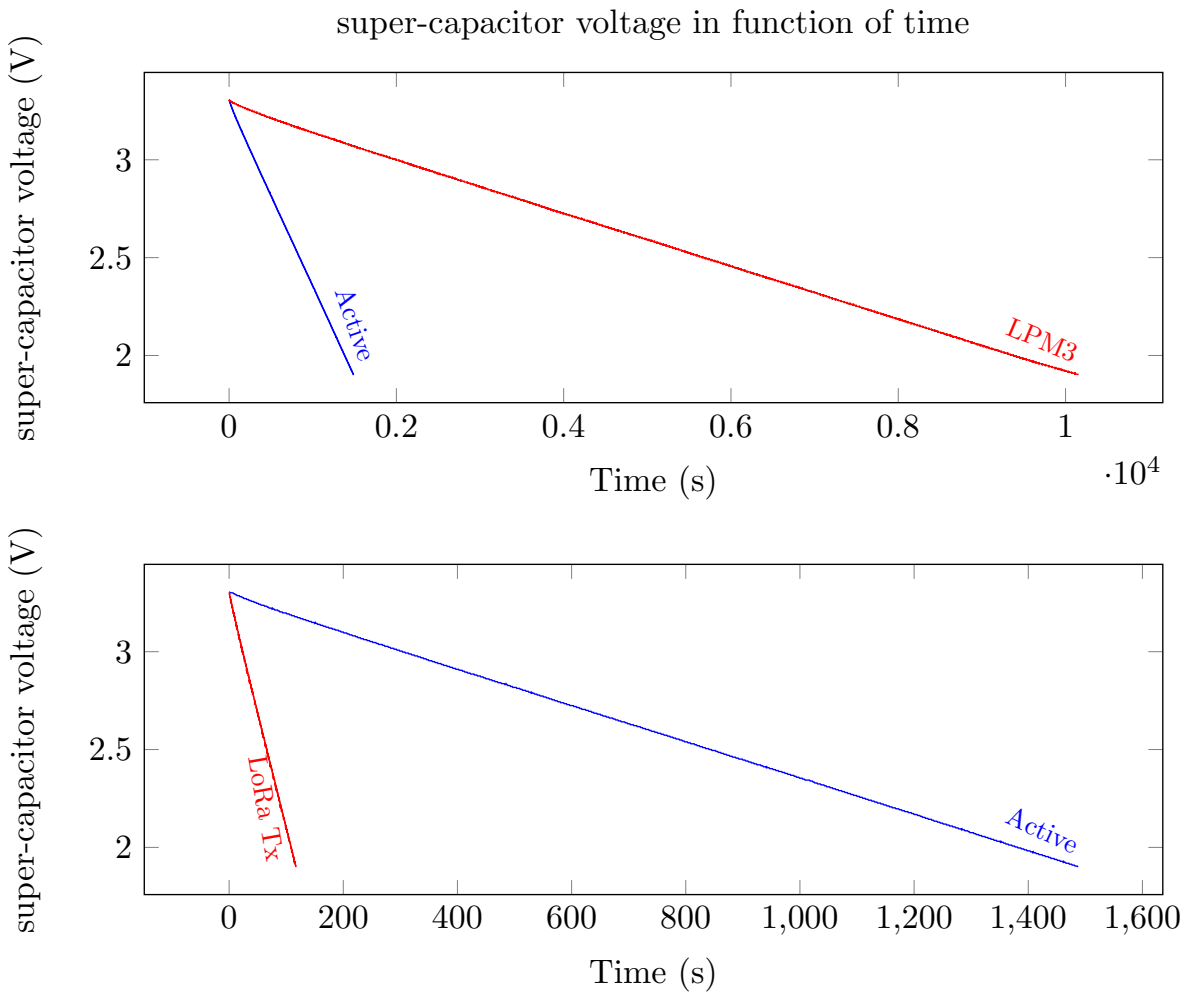


FIGURE 1.7 – Évolution de la tension aux bornes du supercondensateur pour différents modes de fonctionnement. Le mode *LPM3* (Low-Power Mode 3) [Msp] est un mode où de nombreux composants ne sont pas alimentés (le CPU, les périphériques internes, certaines horloges). Le mode *Active* est un mode de fonctionnement où le CPU est alimenté. Le mode *LoRa TX* est un mode où le périphérique radio LoRa est alimenté et en train d'envoyer des données à une puissance de 14 dBm. Pour ces mesures, on utilise un supercondensateur de 0.5 F et une mesure est effectuée toutes les 1 s.

Chapitre 2

Modélisation d'un système intermittent

Dans ce chapitre, nous nous intéressons à la modélisation et à l'analyse de systèmes intermittents. Pour décrire les différents aspects du système, un formalisme permettant de décrire le parallélisme, l'écoulement du temps et la consommation d'énergie est nécessaire. Le formalisme choisi est celui des Réseaux de Petri temporels à coût. À partir de ce formalisme, nous définissons un modèle pour la génération d'ordonnancement garantissant deux critères :

— Les opérations logicielles ou matérielles de l'application ne sont pas lancées si elles ne peuvent pas terminer avec l'énergie courante.
— L'ordonnancement accorde la priorité aux opérations logicielles ou matérielles qui contribuent le plus à l'avancement de l'application.

En effet, nous avons vu précédemment qu'à cause de l'exécution intermittente de l'application, si une opération en cours s'arrête prématurément suite à une perte d'alimentation, il faudra utiliser de l'énergie et du temps afin de refaire l'opération et dans le pire des cas, le système peut être dans un état instable. On souhaite donc garantir la terminaison de toute opération commencée. Le deuxième point permet de choisir entre plusieurs opérations lorsqu'il y a conflit pour l'accès aux ressources partagées. La notion d'avancement de l'application est purement subjective et dépend de l'application. Si on reprend l'exemple d'une application de détection du chant des oiseaux introduit en section 1.1.2, suivant les finalités de l'application, la priorité peut être mise soit sur l'acquisition, soit sur l'envoi de données.

Pour commencer, nous proposons un modèle de calcul pour les systèmes intermittent qui permet de formaliser les deux critères proposés ci-avant, il est décrit en section 2.1. Afin de générer nos ordonnancements, nous définissons les limites du problème que nous souhaitons adresser en section 2.2. En section 2.3 et 2.4, nous définissons le formalisme utilisé pour notre modélisation d'un système intermittent. Basé sur le formalisme présenté,

nous proposons un algorithme symbolique pour l'exploration de l'espace d'état contraint par le coût en Section 2.5, et nous poursuivons en présentant des algorithmes pour synthétiser des ordonnancements optimaux avec une exploration exhaustive de l'espace d'état en section 2.6 et partielle en section 2.7.

2.1 Modèle de calcul

Nous considérons un modèle de calcul intermittent, parallèle, basé sur des tâches. Dans ce modèle, une charge de travail du système est composée de différentes tâches. Ces tâches peuvent être des opérations logicielles, matérielles, ou assemblage d'opérations logicielles et matérielles [1] La présence de tâches matérielles implique un modèle d'exécution parallèle : une tâche matérielle peut être exécutée en même temps qu'une autre tâche qui n'utilise pas les mêmes ressources.

Notre modèle de calcul étant un modèle intermittent, une tâche a une sémantique transactionnelle : une fois commencée, soit la tâche finit avant la prochaine perte d'alimentation, soit l'avancement est perdu et la tâche devra recommencer depuis le début. Même si cette contrainte pourrait être levée pour les tâches purement logicielles (en sauvegardant leur état volatile dans le checkpoint), il est plus difficile pour les tâches avec des opérations matérielles utilisant des périphériques dont le contexte est inaccessible (comme les accélérateurs) ou des périphériques qui ne peuvent être interrompus (un émetteur radio émettant des données).

Les tâches peuvent être reliées par des liens de précédences. Par exemple, un ensemble de données peut être échantillonné avec un capteur, puis traité par des fonctions logicielles, et enfin le résultat de ces fonctions logicielles est envoyé par transmission radio. Des tampons intermédiaire peuvent être ajoutés entre les différentes tâches afin de former une exécution pipelinée. Ainsi, plusieurs résultats à la suite peuvent être enregistrés en mémoire en attendant d'avoir assez d'énergie pour faire la prochaine transmission radio. L'intérêt du modèle d'exécution en pipeline est d'offrir une plus grande flexibilité à l'ordonnanceur, y compris en présence de contraintes de précédence, pour choisir les prochaines tâches à exécuter en fonction des ressources disponibles (mémoire, énergie) et des objectifs fonctionnels (progression). De plus, c'est un modèle général, qui peut modéliser une exécution sans pipeline (en utilisant une taille des tampons de 0 entre les différentes

1. Par exemple, la capture de données de capteurs analogiques à intervalles réguliers peut être réalisée sans intervention du logiciel en utilisant un timer, le dispositif de conversion analogique-numérique (ADC) et le dispositif d'accès direct à la mémoire (DMA).

tâches). Il est important de noter que même en présence de contraintes de précédence liant toutes les tâches du système, le modèle d'exécution en pipeline permet d'exploiter partiellement le parallélisme offert par la plateforme [2].

Dans le modèle, le temps d'exécution des tâches est fixé entre le meilleur temps d'exécution estimé (BCET) et le pire temps d'exécution estimé (WCET).

2.2 Énoncé du problème

Soit un capteur intermittent utilisant l'architecture décrite en section 1.1 et mettant en œuvre le modèle de calcul décrit en section 2.1. Nous nous intéressons au problème de l'ordonnancement des tâches du système afin de maximiser son efficacité énergétique. Pour cela, nous avons deux objectifs :

— d'une part, lorsque des choix doivent être faits entre plusieurs tâches, l'énergie doit être consommée pour exécuter celles qui font le plus progresser le système vers ses objectifs fonctionnels ;

— d'autre part, l'énergie ne doit jamais être utilisée pour exécuter une tâche dont la terminaison n'est pas garantie.

La formulation de ce problème fait apparaître plusieurs éléments. Tout d'abord, une consommation d'énergie doit être attachée à une tâche, et plus largement, nous devons être capables de calculer la consommation d'énergie instantanée du système à un instant donné en fonction des tâches en cours. Le modèle linéaire de consommation d'énergie présenté en section 1.2.2 sera utilisé. Chaque tâche est alors considérée comme un mode et est associée à une pente de tension qui caractérise sa consommation d'énergie.

La consommation du système est obtenue en combinant les pentes des tâches en cours. L'introduction de la consommation d'énergie, permet de donner un critère pour le découpage de l'application en tâches. Les autres modèles précédemment proposés pour les systèmes intermittents sont basés sur des aspects *best-effort*, c'est-à-dire en exécutant le plus d'instruction possible avec l'énergie disponible sans prévision de terminaison de la tâche en cours [LR15 ; MCL17]. Cela laisse le développeur de l'application sans piste pour le découpage de l'application en tâche de façon adaptée à une exécution intermittente.

Deuxièmement, il faut quantifier la manière dont une tâche fait progresser le système vers ses objectifs fonctionnels. Bien entendu, il ne s'agit pas d'une quantité physique

2. Le modèle d'exécution en pipeline reste général, car nous pouvons toujours considérer une tâche logicielle comme un tout, avec un pipeline à une seule étape

objective. Il s'agit plutôt d'une valeur que le concepteur du système doit pouvoir associer à chaque tâche. Nous désignons *récompense* la quantification de ces objectifs fonctionnels. Il convient de noter que la *récompense* associée aux tâches n'est pas nécessairement une valeur fixe, mais qu'elle peut changer en fonction de l'état du système. Par exemple, l'accumulation de données dans un tampon va faire augmenter la *récompense* de la tâche consommant les données de ce tampon. Ainsi, plus le tampon se rempli, plus il devient urgent de le vider afin d'éviter de perdre des données à cause d'une dépassement de la capacité du tampon.

L'idée d'attribuer une valeur aux tâches afin de les ordonnancer n'est pas nouvelle et a déjà été explorée dans différents domaines d'application [HCL93 ; Jai+14 ; PB00]. À notre connaissance, c'est la première fois qu'elle est utilisée en conjonction avec un modèle informatique intermittent.

Comme nous voulons obtenir des ordonnancements fiables (c'est-à-dire garantissant que toute tâche dont l'exécution est commencée se terminera avant la prochaine perte d'alimentation) en toutes circonstances, dans notre modélisation du système, nous n'utilisons que le WCET des tâches. Avec le modèle linéaire de consommation d'énergie que l'on considère, cela garantit que l'on se place dans le cas de la plus grande consommation d'énergie pour les tâches si il n'y a pas de préemption [Wäg+18].

Étant donné un modèle qui tient compte de toutes les dimensions énumérées ci-dessus, le problème que nous souhaitons résoudre est d'obtenir une exécution conduisant à un état où la *récompense* est maximale parmi tous les états atteignables sous une contrainte d'énergie donnée.

La contrainte d'énergie représente la quantité d'énergie disponible utilisable. Si plus d'un état atteint la *récompense* maximale, nous nous intéressons à l'état qui l'atteint en dépensant le moins d'énergie possible. Nous expliquons dans les sections suivantes comment formaliser et résoudre ce problème en utilisant une extension des Réseaux de Petri temporels avec coût.

2.3 Formalisme

2.3.1 Notations

On notera $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ et $\mathbb{R}$ respectivement les ensembles des entiers naturels, des entiers relatifs, des rationnels et des réels. On considèrera 0 comme un élément de $\mathbb{N}$ et $\mathbb{N}^*$

l'ensemble $\mathbb{N}$ privé de 0. $\mathbb{R}_{\geq 0}$ est l'ensemble des réels positifs ou nuls. Pour $n \in \mathbb{N}$, nous notons $[\![0, n]\!]$ l'ensemble $\{i \in \mathbb{N} \mid i \leq n\}$. L'ensemble des intervalles réels non vides dont les extrémités sont rationnelles (respectivement des nombres naturels) ou infinies est noté $\mathcal{I}_{\mathbb{Q}_{\geq 0}}$ (respectivement $\mathcal{I}_{\mathbb{N}}$). Pour $I \in \mathcal{I}_{\mathbb{Q}_{\geq 0}}$, $\underline{I}$ désigne son extrémité gauche et $\overline{I}$ désigne son extrémité droite si I est borné et ∞ sinon.

De plus, pour tout $d \in \mathbb{R}_{\geq 0}$, nous désignons $I \ddot{-} d$ comme l'intervalle défini par $\{\theta - d \mid \theta \in I \wedge \theta - d \geq 0\}$.

Pour F et F', deux systèmes d'inégalités linéaires sur un ensemble de variables X, nous notons $F \equiv F'$ lorsqu'ils ont des ensembles de solutions égaux sur X. Nous notons par $F_{|Y}$ (avec $Y \subseteq X$) la projections de F sur Y (en utilisant l'élimination de Fourier-Motzkin des variables Z avec $Y \cup Z = X$ et $Y \cap Z = \emptyset$).

2.3.2 Réseau de Petri

Un réseau de Petri (PN), également connu sous le nom de réseau de places/transitions, est un langage de modélisation permettant de représenter des systèmes. Un PN se compose de quatre éléments : les places, les transitions, les arcs et les jetons. Les arcs connectent les places avec les transitions et les transitions avec les places. On ne peut connecter deux places ou deux transitions ensemble. Les places peuvent avoir des jetons qui sont consommés et produits lorsque les transitions sont tirées. Une transition est sensibilisée s'il y a assez de jeton dans les places en amont de la transition pour être consommés.

Définition 2.1 (Réseau de Petri (PN))**.** *Un PN est un n-tuplet* $(P, T, {}^\bullet., .^\bullet, m_0)$ *avec :*

- P *un ensemble fini non nul de* places*,*
- T *un ensemble fini de* transitions *tel que* $T \cap P = \emptyset$*,*
- ${}^\bullet. : T \to \mathbb{N}^P$ *est la fonction d'incidence arrière,*
- $.^\bullet : T \to \mathbb{N}^P$ *est la fonction d'incidence avant,*
- $m_0 : P \to \mathbb{N}$ *est le marquage initial.*

Les fonctions d'incidence avant et arrière sont les fonctions décrivant les relations entre les places et les transitions. La fonction d'incidence arrière décrit les conditions pour sensibiliser chaque transition alors que la fonction d'incidence avant décrit le résultat du tir de chaque transition du réseau.

Représentation

Un PN est représenté dans la figure 2.1. Il est composé de 10 places symbolisées par les cercles et de 7 transitions symbolisées par les carrés. Dans le marquage initial, 3 jetons sont présents avec un seul jeton dans les place p_1, p_2 and p_3.

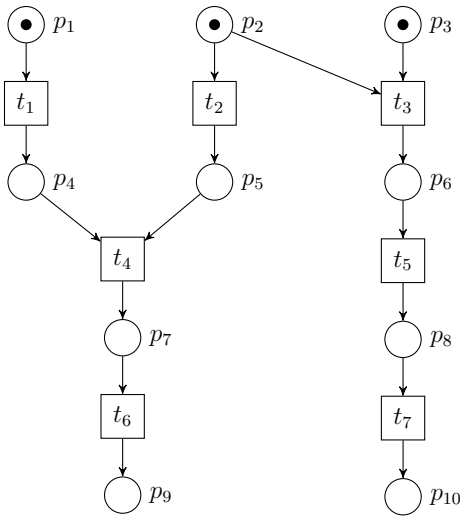


FIGURE 2.1 – Exemple d'un réseau de Petri

Un marquage est une correspondance entre P et $\mathbb{N}$. Pour un marquage $m \in \mathbb{N}^P$, $m(p)$ désigne le nombre de jetons dans la place p. Un PN $\mathcal{N}$ est dit k-borné ou simplement borné si le nombre de jetons dans chaque place ne dépasse pas un nombre fini k pour tout marquage atteignable à partir de m_0.

— Une transition $t \in T$ est dite *sensibilisée* par un marquage $m \in \mathbb{N}^P$ si m fournit t avec au moins autant de jetons que l'exige la fonction d'incidence arrière $^\bullet$. Nous définissons l'ensemble $En(m)$ des transitions activées par le marquage m comme $En(m) = \{t \in T \mid m \geq {}^\bullet(t)\}$

— Une transition $t' \in T$ est dite *nouvellement sensibilisée* par le tir d'une transition t depuis un marquage $m \in \mathbb{N}^P$ si elle est sensibilisée par $m - {}^\bullet t + t^\bullet$ mais pas par m. L'ensemble des transitions qui sont nouvellement sensibilisées par le déclenchement de t depuis le marquage m est $\mathcal{N}ewlyEn(m,t) = \left\{t' \in En(m - {}^\bullet t + t^\bullet) \;\middle|\; t' \notin En(m - {}^\bullet t) \text{ or } t = t'\right\}$

Les réseaux de Petri temporel (TPNs), introduis par Merlin [Mer74], sont une extension des PNs, où les transitions sont associées à des intervalles de temps. Dans un TPN, il y a une horloge implicite par transition sensibilisée et chaque transition a un intervalle de

temps, appelé intervalle de tir, pendant lequel la transition est tirable. En supposant que la transition ait été sensibilisée pour la dernière fois à un instant d et que les bornes de son intervalle de tir sont α et β, alors la transition ne peut pas être tirée avant $d + \alpha$ et doit être tirée avant $d + \beta$ à moins qu'elle n'ait été désactivée par le tir d'une autre transition. Le tir d'une transition se fait en un temps nul. Pour pouvoir tirer une transition dans un TPN, la partie logique et la partie temporelle du modèle doivent être conformes, *i.e.* les jetons nécessaires à la transitions doivent être présents dans la ou les places en amont et l'horloge de la transition sensibilisée doit être dans son intervalle de tir.

Définition 2.2 (Réseau de Petri temporel (TPN))**.** *Un TPN est un n-tuplet* $(P, T, {}^\bullet., .^\bullet, m_0, I_s)$ *avec :*

- $(P, T, {}^\bullet., .^\bullet, m_0)$ *est un PN*
- $I_s : T \to \mathcal{I}_{\mathbb{N}}$ *est une fonction assignant un intervalle de tir à chaque transition.*

La figure 2.2 montre la représentation d'un TPN.

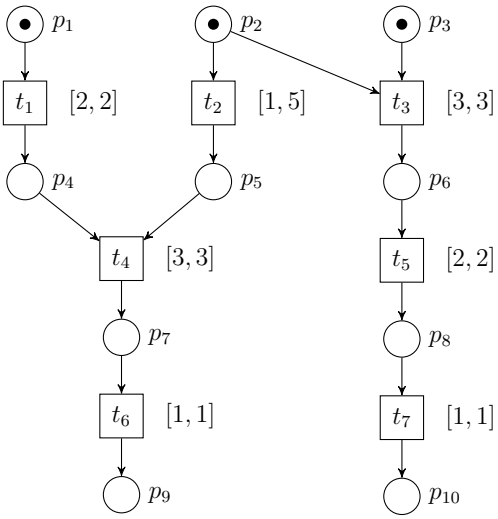


FIGURE 2.2 – Exemple d'un TPN avec pour PN sous-jacent celui de la figure 2.1.

Définition 2.3 (État d'un TPN)**.** *Un* état *du réseau* $\mathcal{N}$ *est une paire* (m, I) *dans* $\mathbb{N}^P \times \mathcal{I}^T_{\mathbb{Q}_{\geq 0}}$*, avec* m *un marquage de* $\mathcal{N}$ *et* I *une fonction appelée fonction de temporisation.* $I : T \to \mathcal{I}_{\mathbb{Q}_{\geq 0}}$ *associe un intervalle de temps à chaque transition sensibilisée par* m*.*

La sémantique d'un TPN consiste à changer d'état soit en déclenchant une transition,

soit en laissant s'écouler du temps.

Définition 2.4 (Sémantique d'un TPN)**.** *La sémantique d'un TPN est un système de transition temporisée* $(Q, q_0, \to)$ *avec :*

- $Q \subseteq \mathbb{N}^P \times \mathcal{I}^T_{\mathbb{Q}_{\geq 0}}$
- $q_0 = (m_0, I_0)$ *avec* $\forall t \in En(m_0) \;\; I_0(t) = I_s(t)$
- $\to$ *est un ensemble composé de deux types de transitions :*
  - *Une transition discrète :* $(m, I) \xrightarrow{t} (m', I')$ *si et seulement si :*
    - $m \geq {}^\bullet t$, $m' = m - {}^\bullet t + t^\bullet$ *et* $\underline{I(t)} = 0$,
    - $\forall t' \in En(m')$
      - $I'(t') = I_s(t')$ *si* $t' \in \mathcal{N}ewlyEn(m, t)$,
      - $I'(t') = I(t')$ *sinon*
  - *Une transition temporelle :* $(m, I) \xrightarrow{\delta \in \mathbb{Q}_{\geq 0}} (m, I \ddot{-} \delta)$ *si et seulement si* $\forall t \in En(m)$, $\overline{(I \ddot{-} \delta)(t)} \geq 0$.

Une séquence de transition d'un TPN $\mathcal{N}$ est une séquence (finie ou infinie) dans sa sémantique commençant par l'état initial q_0.

On note $(m, I) \xrightarrow{t@\delta} (m', I')$ pour la séquence écoulant un temps δ suivi du tir de la transition t : $(m, I) \xrightarrow{\delta} (m, I \ddot{-} \delta) \xrightarrow{t} (m', I')$.

L'ensemble des séquences de transitions d'un TPN est noté Runs.

On définit $sequence(\rho)$ (respectivement $trace(\rho)$) la projection de la séquence de transitions ρ sur T (resp. $T \times \mathbb{Q}_{\geq 0}$). La séquence σ (resp. la trace τ) correspondant à la séquence de transitions $\rho = q_0 \xrightarrow{t_0@\delta_0} q_1 \xrightarrow{t_1@\delta_1} q_2 \xrightarrow{t_2@\delta_2} q_3$ est $\sigma = sequence(\rho) = t_0 t_1 t_2$ (resp. $\tau = trace(\rho) = t_0@\delta_0.t_1@\delta_1.t_2@\delta_2$).

L'ensemble des séquences Runs d'un TPN $\mathcal{N}$ peut être représenté dans un graphe où les états sont les marquages et les arcs sont les transitions du réseau. On appel ce graphe, le graphe d'état de $\mathcal{N}$.

Définition 2.5 (Graphe d'état d'un TPN)**.** *Le graphe d'état discret (DSG) d'un réseau de Petri temporisé (TPN) est la structure* $DSG = (S, s_0, \hookrightarrow)$ *où* $S \in \mathbb{N}^P \times \mathcal{I}^T_{\mathbb{Q}_{\geq 0}}$, $s_0 = (m_0, I_s)$ *et* $s \overset{t}{\hookrightarrow} s'$ *si et seulement si* $\exists \delta \in \mathbb{Q}_{\geq 0} \mid s \xrightarrow{t@\delta} s'$

Tout état du DSG est un état de la sémantique du TPN et tout état de la sémantique qui n'est pas dans le graphe d'état est atteignable depuis un état du graphe avec une transition temporelle. Le DSG représente un espace d'état dense dans le sens où les états

du DSG peuvent avoir une infinité de successeur par $\overset{t}{\hookrightarrow}$. La représentation finie d'espaces d'états denses implique le regroupement de certains ensembles d'états.

Classe d'État

Pour une séquence arbitraire de transitions $\sigma = t_1 \dots t_n \in T^*$, soit C_σ l'ensemble de tous les états qui peuvent être atteints par la séquence σ depuis $s_0 : C_\sigma = \{s \in S \mid s_0 \overset{t_1}{\hookrightarrow} s_1 \cdots \overset{t_n}{\hookrightarrow} s\}$. Tous les états de C_σ partagent le même marquage et peuvent donc être écrit comme une paire (m, D) avec m le marquage en commun des états et D l'union des points appartenant à l'ensemble des intervalles de tir. L'union D est appelée le domaine de tir. Soit $\cong$, la relation satisfaite par deux ensembles d'états lorsqu'ils ont le même marquage et le même domaine de tir.

> **Définition 2.6.** *Soit $C_\sigma = (m, D)$ et $C'_{\sigma'} = (m', D')$ deux ensemble d'états, $C_\sigma \cong C_{\sigma'}$ si et seulement si $m = m'$ et $D \equiv D'$.*

Si $C_\sigma \cong C_{\sigma'}$, tout tir possible depuis un état dans C_σ est également possible depuis un état dans $C_{\sigma'}$ et inversement. On définit les classes d'états (comme présentées dans les travaux de Berthomieu [BM83 ; BD91]) comme les ensembles C_σ définis précédemment modulo l'équivalence $\cong$.

> **Définition 2.7.** *Le graphe de classe (SCG) de [BM83 ; BD91] est défini comme l'ensemble des classes d'états avec une relation de transition : $C_\sigma \overset{t}{\to} X$ si et seulement si $C_{\sigma.t} \cong X$.*

Ainsi, le SCG calcule le plus petit ensemble C des classes d'états par rapport à $\cong$. Le SCG est fini si et seulement si le réseau est borné. De plus, le SCG est une abstraction complète de l'espace d'état du TPN.

Étant donnée une classe d'état $C = (m, D)$, un point $x = (\delta_1, \delta_2, ..., \delta_n) \in D$ est composé des valeurs des variables $\theta_1, \theta_2, ..., \theta_n$ qui se réfèrent aux instants de tir dans C des transitions $t_1, t_2 ... t_n$ sensibilisées par m. Le domaine de tir peut être décrit par des inégalités linéaires de la forme $\theta_i \leq k$ ou $\theta_j - \theta_i \leq k'$ avec $k \in \mathbb{N}$ et $k' \in \mathbb{Z}$.

Par exemple, le SCG du TPN de la figure 2.2 est donné en figure 2.3.

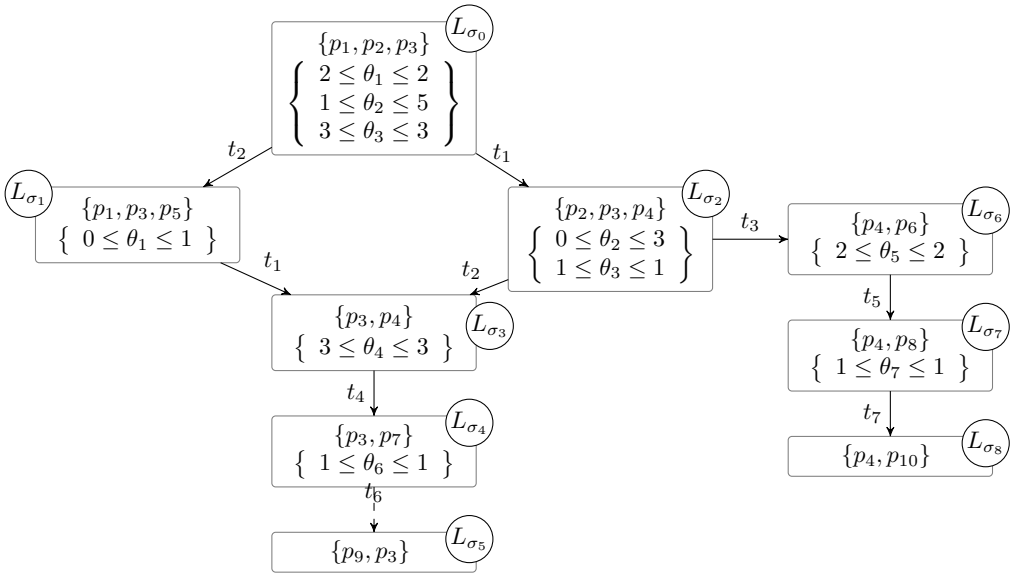


FIGURE 2.3 – Graphe des classe du TPN de la figure 2.2

2.4 Réseau de Petri temporel à coût (cTPN)

Les cTPN sont une extension des TPNs où une fonction de coût est associée aux transitions et au marquage. Dans notre modèle, nous séparons le coût lié aux transitions et le coût lié au marquage. Le coût lié aux transitions est utilisé pour calculer la *récompense*. Lorsqu'une transition est tirée, la *récompense* est mise à jour en ajoutant le coût discret associé à la transition. Le coût lié marquage est le prix par unité de temps d'un marquage. Chaque place est associée à une augmentation du coût par unité de temps et à chaque unité de temps, le *coût* est mis à jour en fonction du marquage.

Définition 2.8 (Réseau de Petri temporel à coût (cTPN))**.** *Un cTPN est un n-tuplet* $\mathcal{N}_c = (P, T, {}^\bullet., .^\bullet, m_0, I_s, \omega, cr)$ *avec :*

- $\mathcal{N} = (P, T, {}^\bullet., .^\bullet, m_0, I_s)$ *est un TPN.*
- $\omega : T \to \mathbb{N}$ *est la fonction de récompense discrète.*
- $cr : \mathbb{N}^P \to \mathbb{Z}$ *est la fonction de taux de coût. cr est une fonction linéaire sur les marquages avec des coefficients entiers.*

Définition 2.9 (Sémantique d'un cTPN)**.** *La sémantique d'un cTPN* $\mathcal{N}_c = (P, T, {}^\bullet., .^\bullet, m_0, I_s, \omega, cr)$ *est la même sémantique qu'un TPN* $\mathcal{N} = (P, T, {}^\bullet., .^\bullet, m_0, I_s)$.

L'état d'un cTPN est un n-tuplet $(m, I, \mathcal{R}, \mathcal{C}) \in \mathbb{N}^P \times \mathcal{I}^T_{\mathbb{Q}_{\geq 0}} \times \mathbb{N} \times \mathbb{R}$, avec (m, I) un état du TPN et $\mathcal{R}$ et $\mathcal{C}$ sont respectivement l'accumulation à partir de l'état initial du coût lié aux transitions discrètes (c'est-à-dire la récompense) et le coût lié au marquage d'une séquence qui conduit à l'état (m, I).

La récompense du tir d'une transition $(m, I, \mathcal{R}, \mathcal{C}) \xrightarrow{t} (m', I', \mathcal{R}', \mathcal{C}')$ est $\mathcal{R}' - \mathcal{R} = \omega(t)$.

Le coût lié à l'écoulement $(m, I, \mathcal{R}, \mathcal{C}) \xrightarrow{\delta} (m, I', \mathcal{R}', \mathcal{C}')$ est $\mathcal{C}' - \mathcal{C} = \delta \times cr(m)$.

Définition 2.10 (Coût d'une séquence de transition (*cost*))**.** *Le coût d'une séquence de transition* $\rho = (m_0, I_0, \mathcal{R}_0, \mathcal{C}_0) \xrightarrow{t_0@\delta_0} (m_1, I_1, \mathcal{R}_1, \mathcal{C}_1) \xrightarrow{t_1@\delta_1} (m_2, I_2, \mathcal{R}_2, \mathcal{C}_2) \cdots \xrightarrow{t_{n-1}@\delta_{n-1}} (m_n, I_n, \mathcal{R}_n, \mathcal{C}_n)$ *est :*

$$cost(\rho) = \mathcal{C}_n = \sum_{i=0}^{n-1} \delta_i \times cr(m_i)$$

Définition 2.11 (Récompense d'une séquence de transition (*reward*))**.** *La récompense d'une séquence de transition* $\rho = (m_0, I_0, \mathcal{R}_0, \mathcal{C}_0) \xrightarrow{t_0@\delta_0} (m_1, I_1, \mathcal{R}_1, \mathcal{C}_1) \xrightarrow{t_1@\delta_1} (m_2, I_2, \mathcal{R}_2, \mathcal{C}_2) \cdots \xrightarrow{t_{n-1}@\delta_{n-1}} (m_n, I_n, \mathcal{R}_n, \mathcal{C}_n)$ *est :*

$$reward(\rho) = \mathcal{R}_n = \sum_{i=0}^{n-1} \omega(t_i)$$

2.4.1 Exemple

Considérons le cTPN de la figure 2.4. En admettant que la fonction de taux de coût soit $cr = 2 \times p_2 + 3 \times p_3 + 3 \times p_5 + 5 \times p_6 + 5 \times p_7 + 5 \times p_8 + 5 \times p_9$ avec p_i une abréviation de $m(p_i)$ et $m(p_i)$ le nombre de jeton dans la place p_i. Le tir des transitions t_2, t_1, t_4 et t_6 respectivement aux instants absolus 1.4, 2, 5 et 6 donne la séquences de transition suivante : $\rho = q_0 \xrightarrow{t_2@1.4} q_1 \xrightarrow{t_1@0.6} q_2 \xrightarrow{t_4@3} q_3 \xrightarrow{t_6@1} q_4$ avec

- $q_0 = (m_0, I_0, \mathcal{R}_0, \mathcal{C}_0) = \left(\left\{ \begin{array}{c} p1 \\ p2 \\ p3 \end{array} \right\}, \begin{array}{l} t_1 : [2, 2] \\ t_2 : [1, 5] \\ t_3 : [3, 3] \end{array}, 0, 0 \right)$
- $q_1 = \left(\left\{ \begin{array}{c} p1 \\ p5 \\ p3 \end{array} \right\}, \ t_1 : [0.6, 0.6] \ , \mathbf{2}, 1.4 \times (2 + 3) = \mathbf{7} \right)$
- $q_2 = \left(\left\{ \begin{array}{c} p4 \\ p5 \\ p3 \end{array} \right\}, \ t_4 : [3, 3] \ , 2 + 0 = \mathbf{2}, 7 + 0.6 \times (3 + 3) = \mathbf{10.6} \right)$
- $q_3 = \left(\left\{ \begin{array}{c} p7 \\ p3 \end{array} \right\}, \ t_6 : [1, 1] \ , 2 + 3 = \mathbf{5}, 10.6 + 3 \times (3 + 3) = \mathbf{28.6} \right)$

— $q_4 = \left(\left\{ \begin{matrix} p9 \\ p3 \end{matrix} \right\}, \ none \ , 5+1 = \mathbf{6}, 28.6 + 1 \times (5+3) = \mathbf{36.6} \right)$

À la fin de la séquence de transition, nous obtenons $reward(\rho) = 6$ et $cost(\rho) = 36.6$

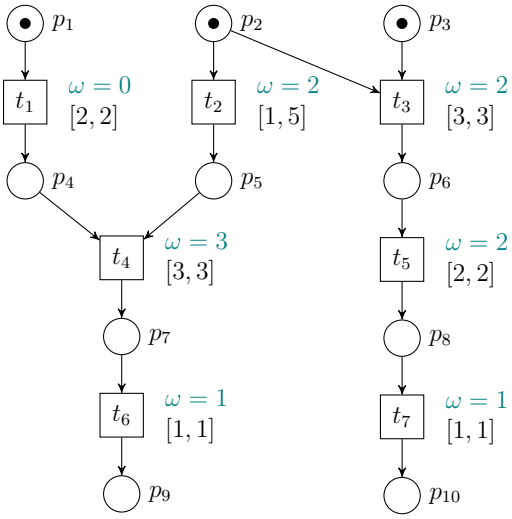


FIGURE 2.4 – Réseau de Pétri T-temporel à coût, avec une fonction de taux de coût $cr = 2 \times p_2 + 3 \times p_3 + 3 \times p_5 + 5 \times p_6 + 5 \times p_7 + 5 \times p_8 + 5 \times p_9$

2.5 Problème sous contrainte de coût

Soit un cTPN $\mathcal{N}$ et une limite supérieure à la variable de coût, le problème sous contrainte de coûts qui nous intéresse peut être énoncé comme suit :

1. Quelle est la récompense maximale qui peut être atteinte sans dépasser la limite supérieure de coût ?
2. Quelle est la séquence de transitions qui mène à cette récompense optimale et qui minimise le coût ?

Nous définissons l'ensemble des séquences de transition sous contrainte de coût comme toutes les séquences de transition qui satisfont le critère *cost* sous la limite supérieure.

Définition 2.12 (Ensemble des séquences de transition sous contrainte de coût ($\mathsf{Runs}_{c \leq c_{max}}$))**.** *L'ensemble des séquences de transition d'un cTPN sous la contrainte de coût $c \leq c_{max}$ est l'ensemble des séquences (finies ou infinies) dans sa sémantique commençant par le marquage initial q_0 tel que tous les états de la séquence respectent la contrainte : $\rho = q_0 \xrightarrow{t_0@\delta_0} q_1 \xrightarrow{t_1@\delta_1} q_2 \cdots \xrightarrow{t_{n-1}@\delta_{n-1}2} q_n$ tel que $\forall k \leq n$, $\sum_{i=0}^{k-1} \delta_i *$*

$cr(m_i) \leq c_{max}$.
On note cet ensemble $\mathsf{Runs}_{c \leq c_{max}}$.

À partir de cet ensemble des séquences de transition sous contrainte de coût ($\mathsf{Runs}_{c \leq c_{max}}$), nous sommes intéressés par les séquences de transition qui maximisent la valeur de la récompense.

Définition 2.13 (Récompense optimale d'un cTPN sous contrainte de coût)**.** *La récompense optimale sous contrainte de coût* $c \leq c_{max}$ *est :*

$$OptReward(c \leq c_{max}) = reward(\rho)$$

tel que $\rho \in \mathsf{Runs}_{c \leq c_{max}}$ *et* $\nexists \rho' \in \mathsf{Runs}_{c \leq c_{max}} \mid reward(\rho') > reward(\rho)$.

On définit la séquence de transition optimale sous contrainte de coût comme la séquence de transition qui maximise la récompense (*reward*) et qui minimise le coût (*cost*) tout en respectant la contrainte de coût.

Définition 2.14 (Séquence de transition optimale sous contrainte de coût)**.** *Une séquence de transition d'un cTPN sous contrainte de coût* $c \leq c_{max}$ *est définie telle que :*

$$OptRun(c \leq c_{max}) = \rho \in \mathsf{Runs}_{c \leq c_{max}}$$

tel que $reward(\rho) = OptReward(c \leq c_{max})$ *et* $\nexists \rho' \in \mathsf{Runs}_{c \leq c_{max}} \mid reward(\rho') = reward(\rho)$ *et* $cost(\rho') < cost(\rho)$.

2.5.1 Espace d'état sous contrainte de coût

Nous étendons maintenant la classe d'état de [BM83 ; BD91] avec le coût et la récompense. On appel *classes d'états avec coût* ces classes d'état étendues.

Soit une séquence σ de transitions conduisant à une classe d'état $C_\sigma = (m, D)$. Le domaine de tir D est un polyèdre convexe contraignant les délais de tir des transitions sensibilisées par m. Pour une transition sensibilisée t_i, on notera θ_i la variable correspondante sur D. Ces temps de tir sont relatifs à la date absolue du tir de la dernière transition de σ (ou 0 pour la classe initiale).

Les classes d'états avec coût $L_\sigma = (m, \mathcal{R}, F)$ étendent les classes d'états comme suit :

— l'état discret est maintenant donné par le marquage m et la récompense $\mathcal{R}$ obtenus par la séquence de transitions σ,
— le domaine de tir est étendu avec une variable de coût c, initialement égale à c_0, et évoluant comme décrit dans la sémantique ci-dessus, et en utilisant l'observation suivante : les dates de tir étant relatives à la dernière transition tirée, le temps passé dans une classe avant le tir d'une transition donnée t_i est exactement θ_i.

Le calcul des classes d'états avec coût successifs prolonge alors naturellement le calcul classique des classes d'états de coûts de [BM83 ; BD91] comme suit :

— la classe d'état avec coût initial est : $L_\varepsilon = (m_0, 0, \{\theta_i \in I_s(t_i) \mid t_i \in En(m_0)\} \wedge \{c = c_0\})$.
— Une transition t_f est tirable depuis une classe $L_\sigma = (m, \mathcal{R}, F)$ sous contrainte de coût $c \leq c_{max}$ si et seulement si :
  — t_f est sensibilisée par m ;
  — $(F \wedge \bigwedge_{i \neq f} \theta_f \leq \theta_i \wedge c + \theta_f \times cr(m) \leq c_{max}) \neq \emptyset$.

  $\mathcal{F}irable(L_\sigma, c \leq c_{max})$ désigne l'ensemble des transitions t_f tirables depuis L_σ sous contrainte de coût $c \leq c_{max}$
— Le successeur $L_{\sigma t_f}$ de la classe d'état avec coût L_σ par une transition t_f tirable depuis L_σ est donné par l'Algorithme 1.

Algorithme 1 Successeur $L_{\sigma t_f} = (m', \mathcal{R}', F')$ de $L_\sigma = (m, \mathcal{R}, F)$ en tirant t_f : $L_{\sigma t_f} = Next(L_\sigma, t_f)$

1: $m' \leftarrow m - {}^\bullet t_f + t_f^\bullet$;
2: $\mathcal{R}' \leftarrow \mathcal{R} + \omega(t_f)$;
3: $F' \leftarrow F \wedge \bigwedge_{i \neq f} \theta_f \leq \theta_i \wedge c + \theta_f \times cr(m)$;
4: for all $i \neq f$, add variable θ'_i to F', constrained by $\theta_i = \theta'_i + \theta_f$;
5: add variable c' to F', constrained by $c' = c + \theta_f \times cr(m)$;
6: eliminate (by projection) variables c, θ_i for all i, and θ'_j for all t_j disabled by firing t_f, from F';
7: for all newly enabled transition t_j, add variable θ'_j, constrained by $\theta'_j \in I_s(t_j)$.

En calculant itérativement les classes d'état étendues, nous obtenons un graphe éventuellement infini dont les arêtes sont étiquetées par les transitions tirées et les nœuds par les classes.

Soit une contrainte $c \leq c_{max}$ où c_{max} est un nombre entier fini, l'Algorithme 2 consiste en une exploration classique de l'espace d'état symbolique, tout en vérifiant la contrainte de coût dans les conditions de tir d'une transition. L'algorithme utilise une liste, PASSED,

afin de garder en mémoire les états symbolique déjà visités.

Le coût n'est pas borné dans la liste PASSED, on utilise donc un opérateur de comparaison symbolique spécifique $\sqsubseteq$ entre les états symboliques [RLS06 ; Bou+17].

Définition 2.15. *Soit $L = (m, \mathcal{R}, F)$ et $L' = (m', \mathcal{R}', F')$ deux classes d'états de coûts. On dit que L est subsumée par L', que l'on note par $L \sqsubseteq L'$ si et seulement si $m = m'$, $\mathcal{R} = \mathcal{R}'$ et $\uparrow F \subseteq \uparrow F'$ où $\uparrow F$ est le polyèdre convexe obtenu à partir de F en supprimant toutes les contraintes de limite supérieure sur la variable de coût c.*

Nous supposons qu'au départ le coût est $c_0 \leq c_{max}$.

Algorithme 2 Algorithme symbolique pour l'exploration de l'espace d'état sous contrainte de coût

```
1: ConstrainedStateSpace ← ∅
2: Passed ← ∅
3: Waiting ← {(m_0, 0, F_0 ∧ c = c_0)}
4: while Waiting ≠ ∅ do
5:     select L = (m, R, F) from Waiting
6:     if for all L' ∈ Passed, L ⋢ L' then
7:         add L to Passed
8:         for all t_f ∈ Firable(L, c ≤ c_max), add Next(L, t_f) to Waiting
9:     end if
10: end while
11: for each L = (m, R, F) ∈ Passed do
12:     add (m, R, F ∧ c ≤ c_max) to ConstrainedStateSpace
13: end for
14: return ConstrainedStateSpace
```

Les conditions de tir $t_f \in \mathcal{F}irable(L, c \leq c_{max})$ vérifient $(F \wedge c + \theta_f \times cr(m) \leq c_{max}) \neq \emptyset$, donc sur tout le domaine de tir F' calculé par l'algorithme, il existe au moins un point dans F' qui respecte la contrainte $c \leq c_{max}$. Cependant, nous ne prenons pas en compte la contrainte de coût dans le calcul du successeur $Next(L, t_f)$ Il peut donc exister un domaine F de classes atteignables $(m, \mathcal{R}, F)$ contenant des points dans $F \wedge (c > c_{max})$. Il est évident qu'aucune transition n'est tirable sous contrainte de coût $c \leq c_{max}$ depuis un tel point. Donc la liste PASSED calculée par l'algorithme 2 est correcte en ce qui concerne le marquage et la récompense mais est une approximation du domaine de tir, certains points dans les domaines de tir ne sont pas atteignable mais tous les points atteignable sont présent. Pour obtenir le domaine de tir CONSTRAINEDSTATESPACE vérifiant la contrainte

de coût dans la liste PASSED, il faut intersecter le domaine de tir avec la contrainte $c \leq c_{max}$.

L'algorithme 2 ne terminera pas si le nombre de marquages ou de récompenses atteignables n'est pas fini alors que la contrainte de coût est respectée.

Nous allons maintenant prouver la terminaison de l'algorithme dans le cas où les marquages et les récompenses sont bornés.

Lemme 1 ($F_{\sigma|\theta} \equiv D_\sigma$)**.** *Soit $\mathcal{N}$ un cTPN et $\mathcal{N}_U$ son TPN (c'est-à-dire sans le coût et la récompense). Soit σ une séquence de transitions tirables dans $\mathcal{N}$ depuis un état initial, conduisant à $L_\sigma = (m, a, F_\sigma)$ et $C_\sigma = (m, D_\sigma)$ respectivement pour $\mathcal{N}$ et $\mathcal{N}_U$. La projection de F_σ sur les variables θ est $F_{\sigma|\theta} \equiv D_\sigma$.*

Démonstration. Par induction sur la longueur n de la séquence de transition σ : pour n = 0, la propriété est triviale. Supposons que le lemme soit valable pour σ avec $L_\sigma = (m, a, F)$, $C_\sigma = (m, D)$ et $F_{|\theta} \equiv D$. En considérant t une transition tirable depuis C_σ conduisant à $C_{\sigma.t} = (m', D')$ alors $F \wedge \bigwedge_{i \neq f} \theta_f \leq \theta_i \neq \emptyset$ et t_f est tirable depuis L_σ. De plus, aucune contrainte sur c n'est ajoutée par le calcul de $Next(L_\sigma, t_f)$ (l'Algorithme 1). L'équation $c' = c + \theta_t \times cr(m)$ et l'élimination de la variable c ne change pas l'espace des solutions sur θ donc $F'_{|\theta'} \equiv D'$. □

Par conséquent, en ce qui concerne le graphe des classes d'état de la définition 2.7, un domaine de tir $F_{\sigma|\theta}$ peut être décrit par des inéquations linéaires de la forme $\theta_i \leq k$ où $\theta_j - \theta_i \leq k'$ avec $k \in \mathbb{N}$ et $k' \in \mathbb{Z}$.

Théorème 1. *Si les taux de coût sont des nombres entiers, l'Algorithme 2 termine si le TPN est borné et si la récompense est bornée.*

Démonstration. Si le TPN est borné alors le nombre de classes d'état atteignables $C = (m, D)$ est fini. De plus, le nombre de récompenses est également fini (nombre entier positif borné). Supposons qu'il existe dans le graphe de classe avec coût un nombre infini de classes d'états avec coût, alors il existe un nombre infini de classes d'états avec coût partageant le même marquage, la même récompense et, en utilisant le lemme 1, le même domaine de tir $F_{|\theta}$. Or ces classes d'états avec coût ne diffèrent que par les contraintes sur la variable de coût c. Il a été prouvé dans des travaux précédents [Bou+17 ; LRS21] que le domaine $\uparrow F$ d'une classe d'état d'un cTPN, peut être divisé en une union de polyèdres plus simples avec exactement une contrainte sur la variable de coût de la forme $c \geq c_{min}$ avec $c_{min} = \ell(\theta_1, \ldots, \theta_n)$ où ℓ est une fonction linéaire à coefficients entiers.

Si les taux de coût sont entiers, ces polyèdres simples ont des sommets entiers de la forme $(\theta_1, \ldots, \theta_n, \ell(\theta_1, \ldots, \theta_n))$ et comme ℓ a des coefficients entiers, alors c_{min} est un nombre entier supérieur à zéro. La condition de tir garantit que pour tout F calculé par l'Algorithme 2, $F \cap (c \leq c_{max}) \neq \emptyset$ donc c_{min} est un entier compris entre 0 et c_{max} ce qui contredit l'hypothèse de départ. □

2.6 Récompense optimale et séquences de transitions optimales

Rappelons que nous cherchons à maximiser la récompense en respectant la contrainte de coût, mais que parmi les solutions, nous retiendrons celle où le coût est minimal.

Le coût optimal de la séquence de transitions σ conduisant à $L_\sigma = (m, a, F)$ est $inf(F_{|c})$ (c'est-à-dire la valeur minimale de c dans F). Puisque pour toutes les classes d'état de PASSED, nous avons $inf(F_{|c}) = inf((F \cap c \leq c_{max})_{|c})$, alors les listes CONSTRAINEDSTATESPACE et PASSED partagent le même coût minimal. Nous pouvons donc nous satisfaire de la liste PASSED pour calculer la solution du problème défini dans la section 2.2.

On a donc : $OptReward(c \leq c_{max}) = \mathcal{R}_{max} = max(\mathcal{R} \mid (m, \mathcal{R}, F) \in \text{PASSED})$

Il peut y avoir plusieurs classes d'état qui ont la récompense optimale et le même coût minimal. Cet ensemble est : $OptL = \{L_\sigma = (m, \mathcal{R}, F_\sigma)$ tel que $\mathcal{R} = \mathcal{R}_{max}$ et $inf(F_{\sigma|c}) = \min(inf\ (F_{|c}) \mid (m, \mathcal{R}_{max}, F) \in \text{PASSED})\}$

Enfin, il peut y avoir plusieurs séquences de transition optimales :

$OptRun(c \leq c_{max}) = \{\rho$ tel que $L_\sigma \in OptL$, $sequence(\rho) = \sigma$ et $cost(\rho) = inf(F_{\sigma|c})\}$

2.6.1 Exemple

Retournons au cTPN de la figure 2.4. Le graphe des classes d'état sous contrainte de coût $c \leq 30$ est donnée en figure 2.5 (nous omettons dans la figure le détail des classes). La classe d'état initiale est L_{σ_0}.

Le tir des séquences de transition $\sigma_3 = t_1 t_2$ et $\sigma'_3 = t_2 t_1$ conduisent à deux classes différentes $L_{\sigma_3} = (m_3, \mathcal{R}_3, F_3)$ et $L_{\sigma'_3} = (m_3, \mathcal{R}_3, F'_3)$ partageant le même marquage, récompense et domaine de tir , avec des valeurs de coût différent mais avec la même valeur de coût minimale. Nous donnons ici le détail des classes L_{σ_0}, L_{σ_1}, L_{σ_2}, L_{σ_3} et $L_{\sigma'_3}$:

$$L_{\sigma_0} = \Big(\left\{ \begin{array}{l} p1 \\ p2 \\ p3 \end{array} \right\}, 0, \left\{ \begin{array}{l} \theta_1 \in [2,2] \\ \theta_2 \in [1,5] \\ \theta_3 \in [3,3] \\ -1 \leq \theta_2 - \theta_1 \leq 3 \\ -2 \leq \theta_3 - \theta_2 \leq 2 \\ 1 \leq \theta_3 - \theta_1 \leq 1 \\ c = 0 \end{array} \right. \Big)$$

$$L_{\sigma_1} = \Big(\left\{ \begin{array}{l} p2 \\ p3 \\ p4 \end{array} \right\}, 0, \left\{ \begin{array}{l} \theta_1 \in [0,1] \\ c \in [5,10] \\ c \geq 10 - 5 * \theta_1 \end{array} \right. \Big)$$

$$L_{\sigma_2} = \Big(\left\{ \begin{array}{l} p1 \\ p3 \\ p5 \end{array} \right\}, 2, \left\{ \begin{array}{l} \theta_2 \in [0,3] \\ \theta_3 \in [1,1] \\ -2 \leq \theta_3 - \theta_2 \leq 1 \\ c = 10 \end{array} \right. \Big)$$

$$L_{\sigma_3} = \Big(\left\{ \begin{array}{l} p3 \\ p4 \\ p5 \end{array} \right\}, 2, \left\{ \begin{array}{l} \theta_4 \in [3,3] \\ c \in [10,15] \end{array} \right. \Big) \quad L_{\sigma'_3} = \Big(\left\{ \begin{array}{l} p3 \\ p4 \\ p5 \end{array} \right\}, 2, \left\{ \begin{array}{l} \theta_4 \in [3,3] \\ c = 10 \end{array} \right. \Big)$$

On a $F'_{3|c} \subseteq F_{3|c}$ et $\uparrow F_3 = \uparrow F'_3$ donc $L_{\sigma'_3} \sqsubseteq L_{\sigma_3}$ et $L_{\sigma_3} \sqsubseteq L_{\sigma'_3}$. Ces classes d'état seront fusionnées par l'Algorithme 2 mais selon l'ordre d'exploration, la classe d'état sélectionnée peut être soit L_{σ_3} ou $L_{\sigma'_3}$.

La classe d'état L_{σ_5}, en pointillée sur la figure 2.5, n'est pas dans le graphe d'état car la contrainte de coût c_{max} (qui est égale à 30) est dépassée.

On obtient $OptReward(c \leq c_{max}) = \mathcal{R}_{max} = 5$

Les classes L_{σ_4} et L_{σ_8} atteignent cette valeur de récompense. Comme $\mathcal{R}_4 = \mathcal{R}_8 = 5$ et $\inf(F_{4|c}) < \inf(F_{8|c})$, la classe d'état optimale est L_{σ_4} avec $\inf(F_{4|c}) = 28$.

Enfin, à partir de ce graphe de classes d'états et de la classe d'états optimale, nous pouvons calculer deux solutions équivalente pour $OptRun(c \leq 30)$:

— $\rho_1 = q_0 \xrightarrow{t_2@2} q_1 \xrightarrow{t_1@0} q_2 \xrightarrow{t_4@3} q_3 \xrightarrow{t_6@1} q_4$
— $\rho_2 = q_0 \xrightarrow{t_1@2} q'_1 \xrightarrow{t_2@0} q_2 \xrightarrow{t_4@3} q_3 \xrightarrow{t_6@1} q_4$

On note que dans la séquence ρ_1, t_2 est tirée à la date 2 (immédiatement avant la transition t_1) car le coût par unité de temps dans la place p_2 est de 2, ce qui est inférieur au coût par unité de temps dans la place p_5 (qui est de 3). Dans ρ_2, t_2 est aussi tirée à la date 2 (immédiatement après t_1) permettant de réduire la durée de la séquence de transition et le coût.

En imaginant que pour le système modélisé, nous pouvons contrôler le tir de t_2, la stratégie pour obtenir une séquence de transitions optimale consiste à tirer t_2 à la date 2.

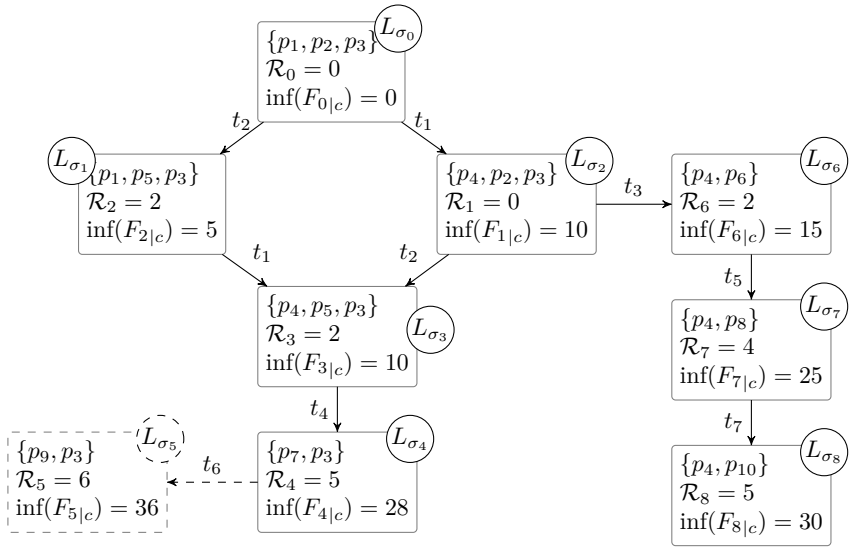


FIGURE 2.5 – Graphe d'état sous contrainte de coût du cTPN de la figure 2.4

2.7 Heuristiques pour *OptRun*

Nous avons présenté un algorithme qui calcule l'ensemble des exécutions optimales d'un cTPN sur la base d'une exploration exhaustive de son espace d'état. Cet algorithme peut souffrir du problème d'explosion de l'espace d'état. Nous allons donc nous concentrer sur un problème plus simple : la recherche d'une classe d'états optimale, *i.e.* la classe d'état finale d'une séquence de transition optimale (*OptRun*) comme définie en section 2.6. Pour ce faire, nous présentons d'abord un algorithme *glouton* qui calcule une "bonne" solution avec une faible complexité de calcul puis, un algorithme qui calcule la solution optimale mais en ajoutant quelques hypothèses.

La recherche d'une séquence de transition optimale en termes de récompense dans l'espace d'état contraint par le coût s'apparente à la recherche du chemin le plus court dans un graphe orienté. En fait, l'algorithme que nous proposons adopte une approche similaire à celle de A^*, un algorithme célèbre [HNR68] pour résoudre le problème du chemin le plus court dans un graphe orienté sans avoir à explorer l'ensemble du graphe. A^* utilise une fonction pour estimer le coût du chemin le plus court entre la racine du graphe et la destination qui passe par un nœud donné n. Cette fonction est généralement notée f, avec $f(n) = g(n) + h(n)$ où g est la fonction qui donne le coût (connu) entre la source et le n, et h est une heuristique qui estime le coût (inconnu) entre n et la destination. A^* explore le graphe en classant les nœuds par valeur croissante de f et s'arrête dès qu'il atteint sa destination. Si h est pessimiste, elle donne une limite supérieure au coût réel,

alors A^* est optimale (il est garanti qu'il trouvera le chemin le plus court).

Cependant, notre problème n'est pas exactement un problème de chemin le plus court. Tout d'abord, dans notre cas, la destination n'est pas identifiée à l'avance. Deuxièmement, notre problème d'optimisation comporte non pas une mais deux variables : la récompense que nous voulons maximiser et le coût que nous voulons minimiser. Comme dans A^*, nous proposons de guider l'exploration de l'espace d'état à coût contraint à l'aide d'une fonction f que nous utilisons pour ordonner les classes d'état en fonction de leur pertinence par rapport à notre problème. Cependant, dans notre cas, cette fonction ne donne pas d'estimation du coût d'une séquence de transition (car nous avons deux variables), et n'est pas séparée en deux parties g et h. Les deux techniques que nous proposons dans les paragraphes suivants sont toutes deux basées sur la traversée de l'espace d'état à coût contraint, comme décrit dans l'algorithme 3. Ils se distinguent par :

— f, la fonction utilisée pour diriger l'exploration,
— et NEEDTOEXPLORE, la procédure permettant de choisir les successeurs d'une classe d'états donnée qui doivent être examinés.

Algorithme 3 Algorithme basé sur des heuristiques pour l'exploration de l'espace d'état sous contrainte de coût

```
1: PASSED ← ∅
2: WAITING ← {(m_0, 0, F_0 ∧ c = c_0)}
3: while WAITING ≠ ∅ do
4:     select L = (m, A, F) whose value of f is the smallest from WAITING
5:     if Firable(L, c ≤ c_max) = ∅ then
6:         return L
7:     end if
8:     remove L from WAITING
9:     add L to PASSED
10:    for t_f ∈ Firable(L, c ≤ c_max) do
11:        L' = Next(L, t_f)
12:        if NEEDTOEXPLORE(L', PASSED) then
13:            add L' to WAITING
14:        end if
15:    end for
16: end while
```

2.7.1 Ratio récompense/coût

Dans la première version de l'analyse, l'ordre d'exploration est déterminé par le ratio récompense/coût. Soit R_n la récompense accumulée entre la classe source et la classe d'état n, et c_n le coût minimal pout attiendre n. Alors, $f(n)$ est calculé comme suit :

$$f(n) = \begin{cases} \frac{c_n}{\mathcal{R}_n} & \text{si } \mathcal{R}_n \neq 0 \\ +\infty & \text{sinon} \end{cases} \tag{2.1}$$

L'espace d'état à coût contraint sera parcouru en explorant les successeurs de la classe d'état qui a la meilleure « dynamique » (le meilleur gain de récompense par coût) parmi les classes présentes dans la liste WAITING. Tous les successeurs doivent être examinés, à l'exception de ceux qui ont déjà été rencontrés (voir l'Algorithme 4)

Algorithme 4 Fonction NEEDTOEXPLORE pour l'heuristique gloutonne

```
procedure needToExplore(L, Passed)
    return L ∉ Passed
end procedure
```

Étant donné que l'algorithme suit avec avidité le rapport récompense/coût le plus élevé, il n'est pas garanti qu'il renvoie une classe d'états faisant partie d'une séquence de transitions optimale. Cependant, suivre le rapport récompense/coût le plus élevé est une stratégie de bon sens pour se déplacer rapidement dans l'espace d'état à coût contraint afin d'obtenir une bonne solution approximative, en un temps de calcul raisonnables.

D'après le cTPN de la figure 2.4, en utilisant cette première version de l'analyse, la partie de l'espace d'état à coût contraint qui est explorée est donnée en figure 2.6. Dans cet exemple, l'analyse trouve la solution optimale, mais ce n'est pas toujours le cas. Par exemple, si $f(L_{signa_2}) = +\infty$, ses successeurs ne seront jamais explorés, même si la solution optimale se trouvait parmi eux.

2.7.2 Discrétisation du coût

Dans le problème défini à la section 2.2, la récompense est fortement liée au coût. N'oublions pas que nous modélisons un système électronique et que tout progrès dans l'application ne peut se faire sans utiliser de l'énergie. Dans le modèle, cela se traduit par : la valeur de la récompense est augmentée si et seulement si le coût est aussi augmenté. Cependant, le coût peut être incrémenté sans qu'il y ait de changement dans la valeur de

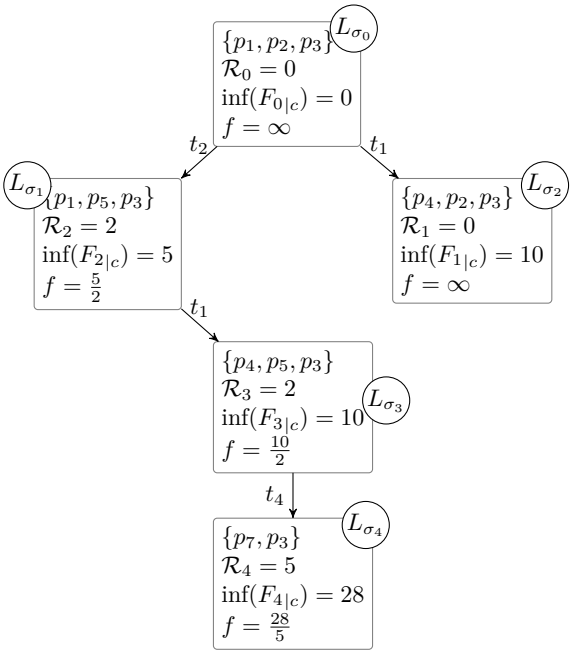


FIGURE 2.6 – Graphe des classes d'état du cTPN de la figure 2.4 en utilisant l'algorithme "glouton".

la récompense. Ainsi, pour tout coût, il y a une récompense maximale associée telle que $\mathcal{R}_i \leq k \times c_i$, avec k un entier. Comme c'est le cas pour toutes les paires de points $(\mathcal{R}, c)$, il en va de même pour $(\mathcal{R}_{max}, c_{max})$ et en connaissant $\mathcal{R}_{max}$ on peut en déduire k car c_{max} est une valeur connue. Dans un tel modèle, sur un graphique récompense/coût, la récompense évolue selon une fonction en escalier (chaque fois qu'une transition avec une récompense non nulle est tirée), et ne dépassera jamais $k \times c$, où c est le coût accumulé jusqu'à présent. Cette situation est illustrée par la figure 2.7. Dans cet exemple, J est un successeur de I et K est un successeur de J. La pente de la droite passant par I et J, ainsi que la pente de la droite passant par J et K est inférieure ou égale à la pente $\frac{\mathcal{R}_{max}}{c_{max}}$.

Cette seconde analyse est conçue pour les systèmes où la récompense est une discrétisation du coût. Pour ces systèmes, nous définissons $\mathcal{R}_{max}$ comme limite supérieure de la récompense pour toutes les séquences de transition. Cette valeur est telle que, dans chaque état, le ratio global de récompense par coût est inférieur ou égal à $\frac{\mathcal{R}_{max}}{c_{max}}$.

Ce nouvel élément d'information peut maintenant être utilisé pour guider l'exploration de l'espace d'état à coût contraint. Pour ce faire, nous calculons $f(n)$ comme suit :

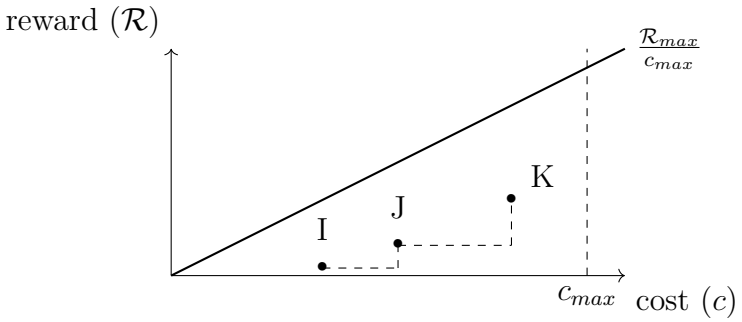


FIGURE 2.7 – Exemple de l'évolution de la récompense pour différents nœuds dans le cas de la discrétisation du coût

$$f(n) = \begin{cases} \frac{1}{\mathcal{R}_n + \frac{\mathcal{R}_{max}}{c_{max}} \times (c_{max} - c_n)} & \text{si } c_n \neq c_{max} \text{ et } \mathcal{R}_n \neq 0 \\ +\infty & \text{sinon} \end{cases} \tag{2.2}$$

Grâce à cette fonction, la procédure de recherche est désormais pilotée par la récompense réelle de la classe d'état et la meilleure récompense hypothétique que l'on puisse obtenir en passant par cet état. Ainsi, lors de l'exploration de l'espace des états à coût contraint, nous pouvons ignorer sans risque les successeurs de la classe d'états Lj lorsque le critère suivant est respecté :

$$\exists L_n \in \text{PASSED}, L_n = (m_n, \mathcal{R}_n, F_n), \quad \frac{1}{\mathcal{R}_n} < \frac{1}{\mathcal{R}_j + \frac{\mathcal{R}_{max}}{c_{max}} \times (c_{max} - c_j)} \tag{2.3}$$

Lorsque le critère est valide, cela signifie que la meilleure récompense possible qui pourrait être obtenue en explorant les successeurs de L_j sera plus petite que la récompense d'une classe déjà explorée. Depuis le critère (2.3), on peut prouver trivialement que : $\mathcal{R}_n > \mathcal{R}_j + \frac{\mathcal{R}_{max}}{c_{max}} \times (c_{max} - c_j)$.

Sur un graphique de récompense/coût, ce point peut être mis en évidence en traçant une ligne droite horizontale à partir de l'intersection de la ligne verticale c_{max} et la projection de j par la droite parallèle de pente $\frac{\mathcal{R}_{max}}{c_{max}}$. Sur la figure 2.8, tout nœud ayant un point dans le triangle formé par$\triangle ABC$ (par exemple le nœud i) est déjà une meilleure solution que n'importe quel successeur du nœud j.

Le critère (2.3) est ajouté dans l'Algorithme 3 dans la fonction NEEDTOEXPLORE comme décrit dans l'Algorithme 5. Contrairement à l'heuristique définie pour l'algorithme

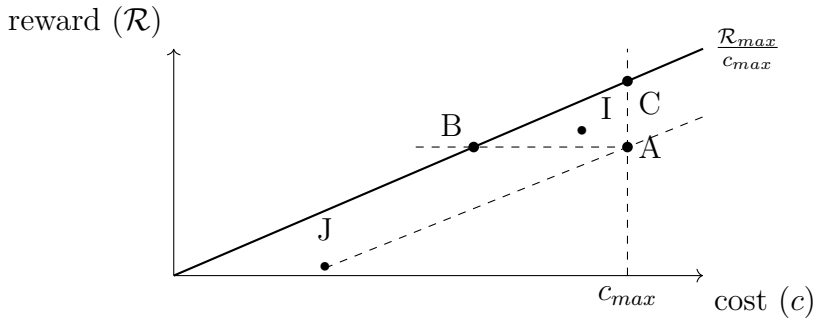


FIGURE 2.8 – Graphique montrant l'élimination des successeurs d'un nœud avec le critère (2.3)

"greedy", nous garantissons que l'état optimal est atteint puisqu'il explore tous les états à l'exception de ceux qui ont été supprimés par le critère. Dans le pire des cas, la procédure explore toutes les classes du graphe d'états à coût contraint.

Algorithme 5 Fonction NEEDTOEXPLORE pour l'heuristique discrétisation du coût

```
1: procedure NEEDTOEXPLORE(L, PASSED)
2:     if L ∈ PASSED then
3:         return False
4:     end if
5:     for M in PASSED do
6:         if 1/R_M < f(L) then
7:             return False
8:         end if
9:     end for
10:    return True
11: end procedure
```

En utilisant l'exemple de la figure 2.4, l'exploration sera comme le graphe d'état présenté dans la figure 2.9. Dans ce cas, l'heuristique ne coupera aucune branche car le critère n'est jamais satisfait. L'heuristique couperait les états après L_{σ_6} dans le cas où la valeur de f pour L_{σ_6} serait inférieur à $\frac{1}{5}$ (depuis l'état L_{σ_4} où $\frac{1}{\mathcal{R}_n} = \frac{1}{5}$).

2.8 Conclusion du deuxième chapitre

Dans ce chapitre, nous avons montré comment formaliser et obtenir des solutions pour le problème d'ordonnancement de tâche faisant progresser au maximum une application sous contrainte d'énergie. La sémantique des réseaux de Petri temporels à coût est utilisée

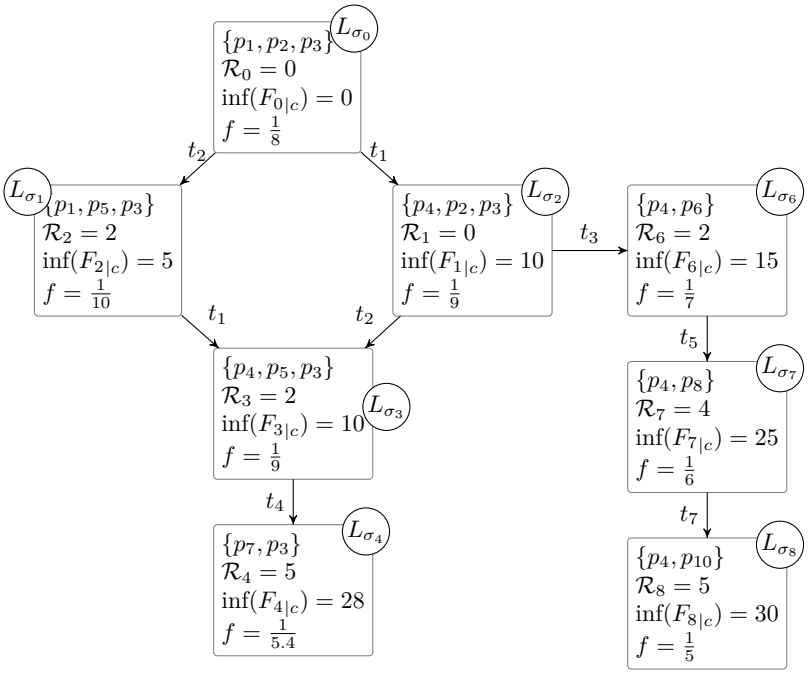


FIGURE 2.9 – Graphe des classes d'état du cTPN de la figure 2.4 en utilisant l'heuristique (2.2)

et nous avons étendu les classes d'états afin d'intégrer la dimension de récompense introduite lors de la modélisation de notre problème. Ce chapitre a aussi introduit plusieurs algorithmes afin d'explorer l'espace des solutions de manière partielle et exhaustive.

Chapitre 3

Conception du support d'éxécution

Dans ce chapitre, nous allons présenter la conception et l'implémentation de *RESURRECT*, un support d'exécution pour les systèmes intermittents qui prend en compte la consommation d'énergie de l'application qu'il héberge. L'objectif principal de *RESURRECT* est d'assurer l'exécution de l'application à travers les pertes d'alimentation inévitables dans le contexte intermittent. De plus, nous cherchons à minimiser le plus possible le surcoût lié à la gestion de l'alimentation intermittente. Nous introduisons en section 3.1 les concepts clés régissant notre proposition pour *RESURRECT*. Nous présentons ensuite son architecture en section 3.2 et son modèle d'exécution en section 3.3. Les différents points pour la gestion du paradigme intermittent sont présentés en section 3.4. Une extension pour intégrer la gestion des événements est introduite en section 3.5. Pour finir, l'intégration de *RESURRECT* dans un système d'exploitation temps réel sous licence libre nommé Trampoline [Bec+06] est décrite en section 3.6 et l'évaluation de ses performances est en section 3.7.

Tout le long de ce chapitre, nous utiliserons le modèle linéaire d'énergie présenté à la section 1.2.2 qui permet de déterminer la quantité d'énergie restante à partir d'une mesure de tension du supercondensateur alimentant le système.

3.1 Concepts de *RESURRECT*

La conception de *RESURRECT* se base sur un certain nombre de concepts que nous allons développer.

La plateforme : La plateforme est une abstraction du système intermittent. Elle décrit uniquement l'aspect énergétique du système, c'est-à-dire l'énergie disponible et utilisable. Comme on se place dans le cas où le système est alimenté via une supercapacité, l'énergie disponible peut être caractérisée par une différence de tension. L'énergie de la plateforme utilisable pratiquement est comprise entre V_{min} et V_{max} où V_{max} représente l'énergie maximale emmagasinable et V_{min} la tension minimale définie de manière à pou-

voir mettre le système en hibernation avant d'atteindre le seuil où le MCU n'est plus suffisamment alimenté et passe en brow out (BOR). Lors de l'exécution de l'application, les décisions pour la gestion des phénomènes liés au contexte intermittent se basent sur l'énergie disponible de la plateforme. Enfin, pour une plateforme, l'énergie disponible n'est pas continue mais quantifiée dans un ensemble $\mathcal{V}$ de n valeurs tel que : $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ avec $\forall i, V_{min} < v_i \leq V_{max}$. Ce choix de quantifier les niveaux d'énergie est un choix de modélisation afin de contrôler la complexité des modèles. Plus on ajoute de niveaux d'énergie, plus le modèles possède un grain fin mais sa complexité augmente.

Mode : Un mode de fonctionnement ou mode, est une paire composée d'une fréquence de fonctionnement et d'une capacité de charge qui reflète l'ensemble des circuits actifs parmi ceux du système. On note l'ensemble des modes du système $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$. Dans le cas du modèle sans régulateur de tension externe, comme défini dans l'équation (1.7), pour un mode donné, la tension d'alimentation décroît de manière linéaire. Pour le cas du modèle de consommation d'énergie avec un régulateur de tension externe, c'est la tension d'alimentation au carré qui décroît de manière linéaire suivant l'équation (1.9). Ainsi, en fonction de la présence ou non d'un régulateur de tension, chaque mode m est associé avec une fonction f_m, où $\frac{df_m}{dt}$ est constante et f_m décrit la pente de tension (ou la pente de la tension au carré) aux bornes du supercondensateur alimentant le système lorsque seul le mode m est actif et que la récolte d'énergie est nulle.

Plus d'un mode peut être actif à la fois et lorsque c'est le cas, la pente de tension caractérisant la consommation du système est calculée en additionnant les pentes de tension de chaque mode actif. Si des circuits sont communs à plusieurs modes, la pente de tension totale peut être plus petite que la somme des pentes de chaque mode. C'est par exemple le cas des horloges qui alimentent le CPU mais aussi tous les périphériques internes. En ce qui concerne *RESURRECT*, nous négligeons ces effets et modélisons la pente de tension résultant d'un ensemble de modes par la somme des pentes des modes. Cette modélisation est acceptable car elle est pessimiste : on surestime la consommation de notre système lorsque plusieurs modes avec des circuits en commun sont actifs. Elle est également acceptable, car la consommation des circuits communs qui sont comptés plusieurs fois correspond la plupart du temps à une faible consommation par rapport à la consommation globale. Par exemple, la consommation d'une horloge utilisée par le CPU et l'ADC est très faible devant la consommation d'énergie du CPU ou celle de l'ADC. Si ce n'était pas le cas, il suffirait de caractériser précisément le circuit en question en l'isolant comme un mode et en effectuant des mesures selon le protocole décrit dans la

section 1.2.3.

Activité : Une activité est le pendant côté applicatif d'un mode. Une activité est une fonction logicielle (par exemple le traitement de données avec un algorithme) ou matérielle (une copie de donnée en utilisant le DMA) dont l'exécution active un nombre fini de modes du système. L'ensemble des activités est $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. Chaque activité a est caractérisée par :

- un ensemble de mode $\mathcal{M}_a$ tel que $\mathcal{M}_a \subseteq \mathcal{M}$,
- un pire temps d'exécution $WCET_a$ tel que $WCET_a \in \mathbb{N}$,
- une pire chute de tension $\Delta V_a = WCET_a \times \sum_{\forall m \in \mathcal{M}_a} \frac{df_m}{dt}$,
- une récompense notée $R_a \in \mathbb{N}$ qui quantifie le progrès réalisé par l'application à la terminaison de l'activité.

Chaque fois qu'une activité est terminée avec succès, l'application progresse. Le but de *RESURRECT* est de maximiser la progression de l'application sous les contraintes liées à l'intermittence de l'alimentation. La récompense R_a d'une activité quantifie cette progression pour chaque activité. On considère les activités comme étant atomiques, c'est-à-dire qu'une activité commencée et interrompue par une perte d'alimentation devra être entièrement recommencée. Toutefois, *RESURRECT* est conçu de façon à ne pas commencer une activité si elle n'est pas garantie de finir sans être interrompue par une perte d'alimentation. Ainsi, on ne commencera une activité que si l'énergie disponible est suffisante pour la terminer, cela se traduit par : $\forall a \in A$, $\Delta V_a \leq V_{courant} - V_{min}$ où $V_{courant}$ représente la tension actuelle d'alimentation. L'exécution d'une activité peut également être activée par des conditions liées à l'état du système. Dans ce cas, les conditions sur l'énergie disponible et l'état du système doivent être vérifiées afin d'exécuter l'activité.

Step : Une étape de calcul, que l'on appelle *step*, est une suite d'activités de telle sorte que pour une énergie de départ disponible donnée, l'exécution de ce *step* maximise la valeur de la récompense tout en assurant que la quantité d'énergie disponible après son exécution soit strictement supérieure à l'énergie minimale requise pour mettre le système en hibernation. Le *step* s se compose des éléments suivants :

- un ensemble n_s de séquences d'activités seq_s,
- un seuil de tension V_s : la tension minimale aux bornes du composant alimentant le système afin d'exécuter s,
- une récompense R_s qui quantifie l'avancement de l'application à la terminaison de son exécution,
- un critère Pr_s qui selon qu'il soit validé ou non, active ou inhibe l'exécution du

step en se basant sur des conditions liées à l'état du système.

L'ensemble fini des *steps* du système est $\mathcal{S}$. Une même activité peut être présente plusieurs fois dans la même séquence d'activités, dans différentes séquences d'un même *step* ou dans différents *steps*. Un *step* est composé de plusieurs séquences d'activités pour deux raisons :

— cela permet la préemption des séquences d'activités, c'est utile par exemple, lorsqu'une activité utilisée pour échantillonner un signal est exécutée en concurrence avec une activité intensive en calcul qui traite un morceau précédent du signal.

— cela permet d'exploiter le parallélisme naturel entre les activités purement matérielles et les activités purement logicielles.

On a donc, $seq_s = \{seq_s^1, seq_s^2, \ldots, seq_s^{n_s}\}$, avec $\forall i$, $seq_s^i = [a_s^i[0], \ldots a_s^i[k_s^i]]$, et $\forall a \in seq_s^i$, $a \in \mathcal{A}$.

Un *step* est construit de telle sorte que si la tension aux bornes du tampon d'énergie n'est pas inférieure à son seuil de tension, le système peut effectuer toutes les activités des séquences associées à celui-ci avant d'atteindre V_{min}. Ainsi, le seuil de tension V_s d'un *step* est égale à V_{min} plus la chute de tension la plus défavorable résultant de l'exécution des séquences d'activités. En pratique, le seuil de tension d'un *step* est fixé au seuil de tension de la plateforme immédiatement supérieur.

Formellement, $V_s = \min_{v \in V}\{v - V_{min} \geq \sum_{a \in seq_s^i, seq_s^i \in seq_s} \Delta V_a\}$.

Bien entendu, la condition $V_s \leq V_{max}$ doit être respectée, sinon le système ne sera jamais en mesure d'exécuter le *step* s.

Enfin, la récompense obtenue en exécutant avec succès le *step* est simplement la somme des récompenses des activités exécutées au cours du *step* : $R_s = \sum_{a \in seq_s^i, seq_s^i \in seq_s} R_a$.

Cycle d'Exécution : Un cycle d'exécution sur une plateforme commence depuis une tension $v \in \mathcal{V}$. Il se termine lorsque la tension disponible pour la plateforme est tombée à une valeur inférieure au plus petit niveau dans l'ensemble $\mathcal{V}$. Au cours d'un cycle d'exécution, il est tout à fait possible d'effectuer plusieurs *steps*, notamment lorsque la récolte d'énergie est non nulle.

3.2 Architecture de *RESURRECT*

La conception proposée pour *RESURRECT* a pour objectif de permettre l'exécution des *steps* au sein de cycles d'exécution.

RESURRECT est constitué des éléments présentés en figure 3.1 et détaillés ci-dessous :

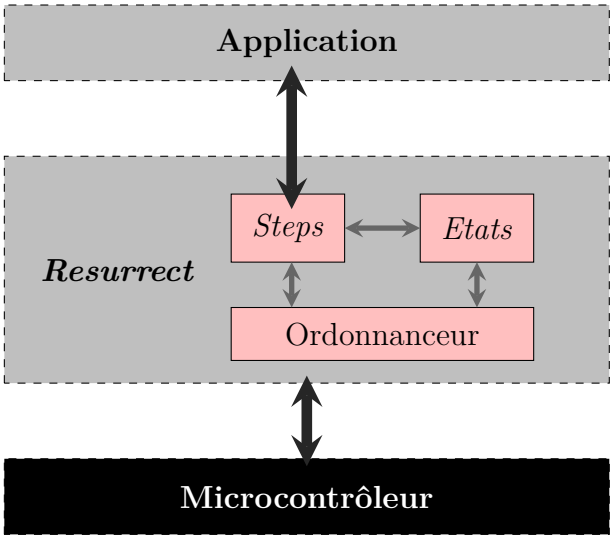


FIGURE 3.1 – Architecture de *RESURRECT*

— **les *steps* :** ils ont été présentés dans la section précédente, et sont générés à partir du modèle de l'application,
— **les états du système :** les états du système sont définis par les valeurs prises par un certain nombre de variables dépendantes de l'application. Les *steps* sont des transitions entre les états du système,
— **un ordonnanceur :** l'ordonnanceur permet de choisir au moment de l'exécution le prochain *step* à exécuter. À la fin d'un *step*, l'état courant du système est mis à jour et l'ordonnanceur se base sur l'énergie disponible et sur l'état du système pour sélectionner le *step* suivant.

3.2.1 Construction des *steps*

Nous avons vu dans le chapitre 2 une méthode permettant d'obtenir un ordonnancement optimal en terme de récompense sous contrainte de coût à partir d'un état de départ. En utilisant cette méthode, nous pouvons générer une séquence de transitions (potentiellement plusieurs en combinant des cTPNs indépendants). En faisant correspondre une partie des transitions dans la modélisation du système par un cTPN avec les activités décrites en section 3.1, nous avons tous les éléments afin de construire un *step*. Les séquences de transition correspondent aux séquences d'activités du *step*, son seuil de tension V_s correspond à la valeur de la contrainte sur le coût, la récompense R_s correspond à la valeur de la variable de récompense pour la classe d'état optimale *OptRun* et finalement,

le critère Pr_s représentant la condition pour l'exécution du *step* s correspond au marquage initial du cTPN.

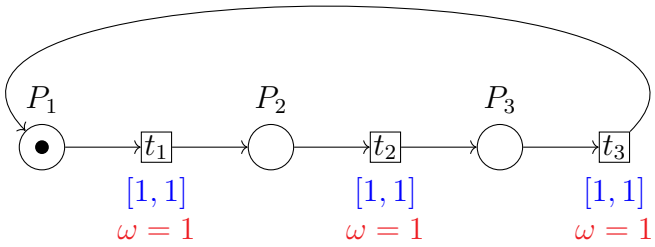
On remarque que pour un état de départ donné (c'est-à-dire un marquage initial donné) en fonction de l'énergie disponible, plusieurs *steps* différents peuvent être générés. En effet, en fonction de l'énergie disponible, des séquences d'activités différentes peuvent être optimales. Au niveau de la modélisation, cela se traduit par l'utilisation de l'algorithme 2 avec différentes valeurs de contrainte sur le coût pour un même marquage initial. Pour correspondre aux concepts de *RESURRECT*, ces contraintes de coût prennent les valeurs d'énergie possibles pour la plateforme, c'est-à-dire parmi l'ensemble $\mathcal{V}$. Ainsi, pour un état de départ donné, on obtient au maximum autant de *steps* que de valeurs dans l'ensemble $\mathcal{V}$ et chacun de ces *steps* a potentiellement un état final différent (un marquage final différent pour les classes d'état optimales).

L'application s'exécutant sur un système sans batterie intermittent est une application cyclique sans état final. Pour être complet, il faut donc générer pour chaque état du système les *steps* pour chaque niveau d'énergie possible de la plateforme de manière itérative. La liste des *steps* calculée par l'algorithme 6 est complète en ce sens.

Pour pouvoir embarquer ces séquences d'activité dans le système intermittent, il ne reste plus qu'à chaîner les *steps* via les états du système. On obtient ainsi un graphe où les sommets sont les états du système et les arêtes les *steps*. Toute cette procédure de génération de *step* est résumé avec un exemple dans la figure 3.2.

Algorithme 6 Algorithme itératif pour la génération des *steps* d'un système à partir de la modélisation avec un cTPN.

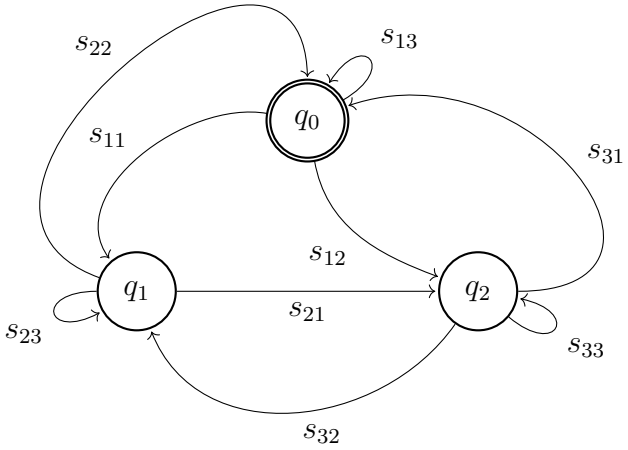
```
1: PASSED ← ∅
2: LIST STEP ← ∅
3: WAITING ← {L_0}
4: while WAITING ≠ ∅ do
5:     select l from WAITING
6:     if l ∉ PASSED then
7:         add l to PASSED
8:         for each v ∈ V do
9:             LIST STEP ← sequence(ρ) such as ρ ∈ OptRun(c ≤ v)
10:            WAITING ← L' such as L' = Next(l, sequence(ρ))
11:        end for
12:    end if
13: end while
14: return LIST STEP
```



(a) Exemple d'un cTPN modélisant une application contenant 3 activités. On pose une fonction de taux de coût $cr = 1 \times p_1 + 1 \times p_2 + 1 \times p_3$. L'application s'exécute de façon cyclique, dès que la dernière activité (t_3) est finie, l'activité de départ (t_1) est redémarrée. Pour cet exemple, on supposera également que l'ensemble $\mathcal{V}$ des valeurs d'énergie disponibles est : $\mathcal{V} = 1, 2, 3$. Ainsi on a $V_{max} = 3$ et $V_{min} = 1$.

$s_{11} = t_1$
$s_{12} = t_1 \rightarrow t_2$
$s_{13} = t_1 \rightarrow t_2 \rightarrow t_3$
$s_{21} = t_2$
$s_{22} = t_2 \rightarrow t_3$
$s_{23} = t_2 \rightarrow t_3 \rightarrow t_1$
$s_{31} = t_3$
$s_{32} = t_3 \rightarrow t_1$
$s_{33} = t_3 \rightarrow t_1 \rightarrow t_2$

(b) Enumération des *step* possibles à partir du cTPN de la Figure 3.2a.



(c) Graphe représentant les états du système ainsi que leur relation via les *steps* générés.

FIGURE 3.2 – Processus de génération des *steps*. Les *steps* sont calculés en utilisant l'algorithme 6 avec trois niveaux d'énergie de départ disponibles ($\mathcal{V} = \{1, 2, 3\}$) et un marquage initial $m_0 = \{p_1\}$. Concernant la nomenclature, le *step* s_{ab} correspond à la séquence de transition avec pour marquage initial un jeton dans la place p_a et une contrainte sur le coût tel que $c \leq b$

3.3 Modèle d'exécution

À l'exécution, *RESURRECT* enchaîne l'exécution des *steps* en fonction de l'énergie disponible. Plus précisément, au moment du démarrage, la tension aux bornes du supercondensateur est mesurée. Ensuite, *RESURRECT* sélectionne le *step* avec la récompense la plus élevée parmi les *steps* qui peuvent être exécutés depuis l'état actuel du système. Étant donnée la tension mesurée $V_{courant}$, l'ensemble des *steps* qui peuvent être exécutés est : $\mathcal{E} = \{s | s \in \mathcal{S} \wedge Pr_s \wedge (V_s \leq V_{courant})\}$. Le *step* choisi est celui qui offre la récompense la plus élevée parmi les *steps* exécutables :

$$elected \in \{s \mid s \in \mathcal{E} \wedge (\forall s' \in \mathcal{E},\ R_s \geq R'_s)\}$$

Si $\mathcal{E} = \emptyset$, cela signifie que la tension courante est trop faible pour l'exécution d'un *step*. *RESURRECT* enregistre les données du programme qui ont été stockées dans la mémoire volatile et entre en hibernation. Le MCU est mis en sommeil jusqu'à ce que la tension aux bornes du supercondensateur soit suffisante pour exécuter un *step*. Il existe plusieurs stratégies possibles pour sortir de ce sommeil. La plus simple consiste à réveiller périodiquement le système après un délai fixé, généralement en programmant l'horloge temps réel (RTC). La procédure de choix d'un *step* est formalisée dans l'algorithme 7.

Algorithme 7 Algorithme pour l'élection du prochain *step* exécuté

```
1: E ← ∅
2: while E = ∅ do
3:     V_current ← MeasureBufferVoltage()
4:     E ← {s | s ∈ S ∧ Pr_s ∧ (V_s ≥ V_current)}
5:     if E = ∅ then
6:         Hibernate
7:     end if
8: end while
9: elected ← PickMaxRewardStep(E)
10: return elected
```

Une fois qu'un *step* est sélectionné, *RESURRECT* commence à l'exécuter. Soit V_{before} (resp. V_{after}) la tension d'alimentation du système avant (resp. après) l'exécution du *step* s. Par construction :

$$V_{after} \geq V_{before} - \sum_{\substack{a \in seq_s^i \\ seq_s^i \in seq_s}} \Delta V_a$$

Par définition, $\sum_{a \in seq_s^i, seq_s^i \in seq_s} \Delta V_a$ est la chute de tension la plus défavorable, rencontrée lorsque toutes les activités de s consomment le plus d'énergie et la puissance d'entrée fournie par la collecte d'énergie est nulle. Il s'agit d'une situation inhabituelle. La plupart du temps, la tension alimentant le système à la fin de l'exécution du *step* élu sera plus élevée que dans le pire des cas, et peut même être suffisante pour exécuter un autre *step*. Ainsi, une fois qu'un *step* a été exécuté, *RESURRECT* tente à nouveau d'exécuter un autre *step*.

Il est bien sûr possible que la tension d'alimentation tombe en dessous de la tension seuil V_{bor} où le système s'éteindra pendant que le système est en hibernation. Dans ce cas, le système s'arrête mais comme il est en hibernation, toutes les données dans la mémoire volatile ont été préalablement sauvegardées (à l'étape *checkpoint* dans la figure 3.3). Il redémarrera simplement lorsque la tension d'alimentation sera suffisante. La seule différence entre un démarrage à la suite d'une perte d'alimentation et un démarrage à froid (c'est-à-dire un démarrage après une hibernation du système) est que dans le premier cas, les données sauvegardées dans la mémoire NVM avant l'entrée en hibernation doivent être restaurées et les périphériques doivent être reconfigurés. Ce modèle d'exécution est résumé dans la figure 3.3.

3.3.1 Propriétés et robustesse de *RESURRECT*

En adoptant une approche pire cas et un modèle de consommation d'énergie, *RESURRECT* fournit un certain nombre de garanties sur l'exécution du système. En particulier, si les hypothèses concernant le temps d'exécution des activités dans le pire des cas et la consommation d'énergie des modes sont correctes, le modèle garantit que :

1. Chaque activité s'exécute intégralement à l'intérieur d'un unique cycle d'exécution.
2. Les coupures d'alimentation ne surviennent pas pendant l'exécution d'un *step*.

Ces deux propriétés simplifient grandement la gestion des *checkpoints*. La première propriété implique qu'il n'est pas utile d'inclure les variables internes aux activités dans le *checkpoint* (par exemple la pile pour une activité logicielle, où les registres et l'état des périphériques pour une activité matérielle). La deuxième propriété implique que, s'il y a

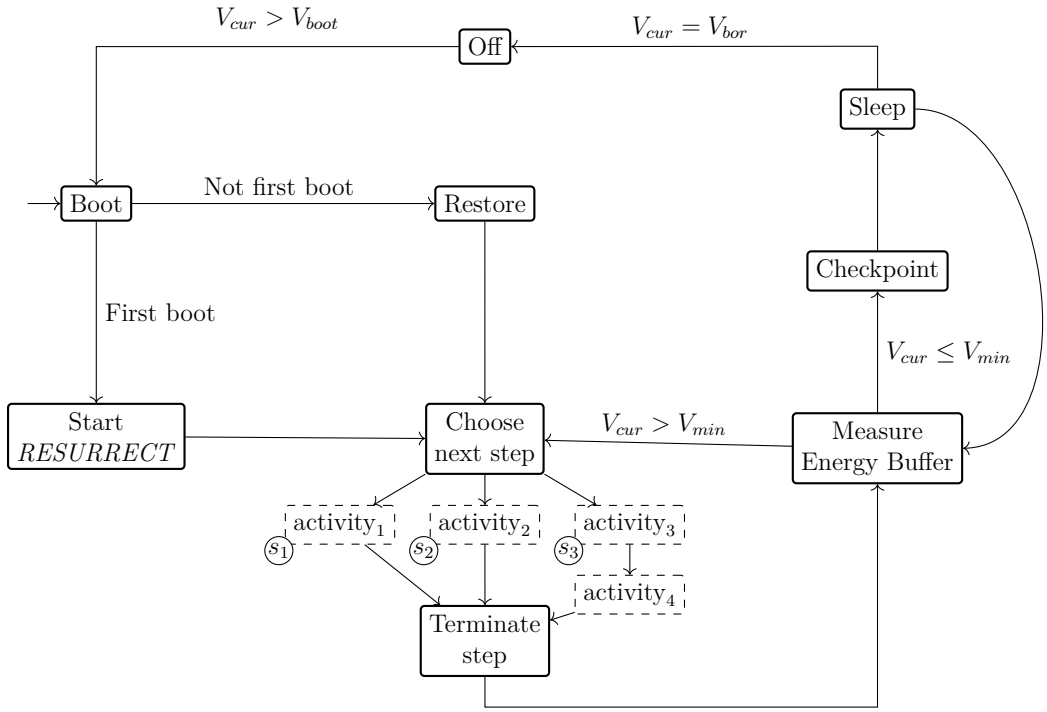


FIGURE 3.3 – Gestion des *steps* dans *RESURRECT*. Ici, 3 *steps* sont représentés par s_1, s_2 et s_3.

une interruption d'alimentation, tout l'avancement dans l'application réalisé au cours du cycle d'exécution actuel a déjà été validé par un *checkpoint*.

Cependant, les systèmes parfaits n'existent pas. Des erreurs peuvent être introduites lors de la conception du système, de sa mise en œuvre ou même au cours de sa durée d'utilisation (par des défaillances matérielles, ou le vieillissement des composants, en particulier le supercondensateur).

Dans le cas de *RESURRECT*, de telles erreurs peuvent conduire à une défaillance sous la forme d'une violation des propriétés 1 et/ou 2. Dans ce cas, l'implémentation va détecter l'état erroné lorsque le système est redémarré et, lancer un démarrage à froid. Cela peut se faire via une variable booléenne en mémoire non volatile qui est mise à l'état haut à chaque début d'exécution d'un *step* et à l'état bas à la fin de chaque *step*. Ainsi, au redémarrage, si la variable est à l'état haut, le système est dans un état erroné. L'avancement de l'application réalisé lors du cycle d'exécution précédent sera perdu, mais le système sera rétabli dans un état valide et pourra poursuivre l'exécution de l'application. Cette stratégie est viable si l'apparition de ces défaillances reste un événement rare, ce qui semble être une hypothèse raisonnable pour un système conçu à l'aide de méthodes

et d'outils de l'état de l'art.

3.4 Gestion de l'intermittence

Sauvegarde de l'état du système : La sauvegarde de l'état du système (aussi appelée *checkpointing*) se fait uniquement à la fin d'un *step* s'il n'y a pas assez d'énergie disponible dans la plateforme pour en exécuter un autre.

Les sauvegardes (aussi appelées *checkpoints*) sont manipulées en utilisant un double-tampon afin d'avoir toujours une sauvegarde valide à disposition. Ce double-tampon consiste à avoir deux espaces mémoires différents pour les sauvegardes, le premier espace est utilisé pour l'écriture de la sauvegarde en cours et le deuxième garde en mémoire la sauvegarde précédente. Lorsque l'écriture de la sauvegarde en cours est finie, les deux espaces mémoires sont échangés. Cette technique est couramment utilisé dans le domaine intermittent [Ber+19 ; ML18] et permet de prévenir la corruption d'une sauvegarde s'il y a perte d'alimentation alors que la sauvegarde est écrite en mémoire. Même si par construction, le modèle d'exécution de *RESURRECT* empêche la perte d'alimentation lorsqu'un *checkpoint* est sauvegardé, cela permet d'ajouter un niveau supplémentaire de robustesse.

Gestion des périphériques : La sauvegarde de l'état du système est insuffisante pour assurer la bonne continuation d'une application dans un contexte intermittent lorsque le système comprend des périphériques. En ce qui concerne les périphériques internes, les registres internes sont généralement liés au bus d'adresse, ce qui permet un suivi précis de leurs états par le CPU. Pour les périphériques externes les registres internes sont inaccessible pour le CPU et des opérations sur les interfaces d'entrées/sorties sont nécessaires pour obtenir l'état du périphérique. Dans les deux cas, nous présupposons que les périphériques sont modélisés par une machine d'état, et que tout changement d'état du périphérique nécessite une intervention soit côté applicatif, soit côté support d'exécution. Ainsi, tout changement d'état est enregistré par le support d'exécution et lors d'un redémarrage, l'état de chaque périphérique est mis à jour. Ces hypothèses sont appropriées car les pilotes des périphériques externes utilisent généralement des machines à états pour suivre leurs modes de fonctionnement [Ber+20a ; Ber+19].

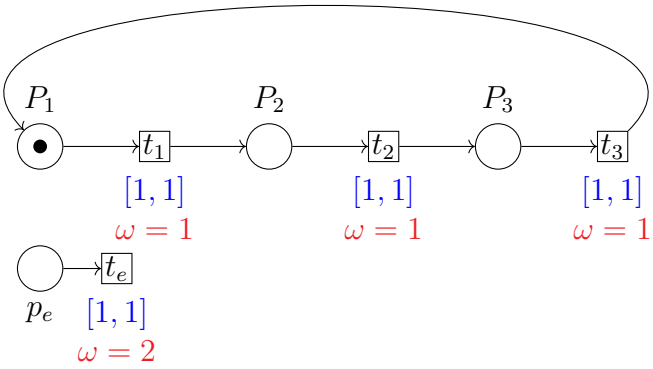
3.5 Extension pour la gestion d'événements

Le modèle d'exécution de *RESURRECT* suppose qu'une limite supérieure de l'énergie nécessaire à l'exécution d'un *step* est connue avant le début de l'exécution.

Dans certains cas, cette approche peut être désavantageuse, notamment lorsque le cas le plus défavorable est très éloigné du cas nominal. Dans le contexte d'application qui nous intéresse, celui des capteurs sans fil autonomes intermittents, cela peut être le cas lorsqu'un événement implique une réponse gourmande en énergie (par exemple, effectuer un calcul complexe et/ou envoyer une trame sur une liaison sans fil). Pour faire face à ce type de situation, nous proposons d'étendre le modèle présenté ci-dessus de la manière suivante.

Tout d'abord, lorsqu'un tel événement est détecté (soit par un dispositif matériel, soit à la suite d'un traitement logiciel), il est stocké en mémoire. Il n'est pas traité dans le *step* en cours. Ensuite, pour chaque *step* du système, nous ajoutons des variantes qui impliquent le traitement d'un ou plusieurs événements en attente. L'événement est donc traité lorsqu'un *step* incluant son traitement est sélectionné. Cela implique un temps de latence plus ou moins important, qui est fonction du temps d'exécution restant du *step* au cours duquel l'événement est détecté. Au niveau de la modélisation, initialiser le système avec un événement en attente revient à ajouter un jeton dans la place correspondante. Cette politique peut être facilement encodée dans le prédicat associé aux *steps* et ne nécessite donc aucune extension de la formalisation proposée ci-dessus. Pour que le système conserve une taille raisonnable, il faut supposer que le nombre d'événements en attente est limité et que cette limite est faible. Cette hypothèse semble raisonnable pour les événements qui nécessitent une réponse énergivore par rapport aux autres fonctionnalités du système.

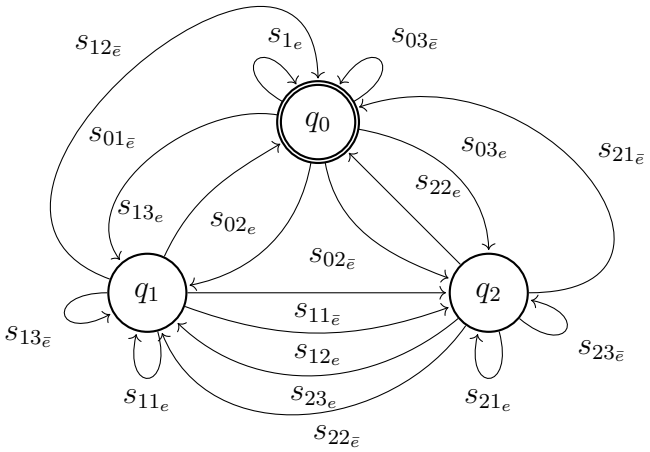
En reprenant l'exemple de la figure 3.2 et en y ajoutant un événement, cela donne le processus de génération illustré dans la figure 3.4.



(a) Exemple d'un cTPN modélisant une application contenant 3 activités et un évènement. On pose une fonction de taux de coût $cr = 1 \times p_1 + 1 \times p_2 + 1 \times p_3 + 1 \times p_e$. Pour cet exemple, on supposera également que l'ensemble $\mathcal{V}$ des valeurs d'énergie disponibles est : $\mathcal{V} = 1, 2, 3$. Ainsi on a $V_{max} = 3$ et $V_{min} = 1$. Ici, on suppose que l'évènement est traité de manière indépendante par rapport à la chaîne d'activité.

$s_{01_e} = t_e$
$s_{01_{\bar{e}}} = t_1$
$s_{02_e} = t_1 \parallel t_e$
$s_{02_{\bar{e}}} = t_1 \rightarrow t_2$
$s_{03_e} = t_1 \parallel t_e \rightarrow t_2$
$s_{03_{\bar{e}}} = t_1 \rightarrow t_2 \rightarrow t_3$
$s_{11_e} = t_e$
$s_{11_{\bar{e}}} = t_2$
$s_{12_e} = t_2 \parallel t_e$
$s_{12_{\bar{e}}} = t_2 \rightarrow t_3$
$s_{13_e} = t_2 \parallel t_e \rightarrow t_3$
$s_{13_{\bar{e}}} = t_2 \rightarrow t_3 \rightarrow t_1$
$s_{21_e} = t_e$
$s_{21_{\bar{e}}} = t_3$
$s_{22_e} = t_3 \parallel t_e$
$s_{22_{\bar{e}}} = t_3 \rightarrow t_1$
$s_{23_e} = t_3 \parallel t_e \rightarrow t_1$
$s_{23_{\bar{e}}} = t_3 \rightarrow t_1 \rightarrow t_2$

(b) Énumération des *steps* possibles à partir du cTPN de la Figure 3.4a. Le symbole $\parallel$ signifie que les deux transitions sont tirées en même temps.



(c) Graphe représentant les états du système ainsi que leur relation via les *steps* générés. Comme il y a un évènement possible, par rapport à la Figure 3.2c, il y a le double de transition possible entre deux états, avec et sans l'évènement traité.

FIGURE 3.4 – Processus de génération des *steps* étendu pour la gestion des événements. Pour la nomenclature, le *step* s_{ab_e} correspond à la séquence de transition avec pour marquage initial un jeton dans la place p_a et p_e et une contrainte sur le coût tel que $c \leq b$. Un *step* $s_{ab_{\bar{e}}}$ n'a pas de jeton dans la place p_e dans son marquage initial.

3.6 Implémentation

3.6.1 Trampoline

Trampoline [Bec+06] est un système d'exploitation temps réel (RTOS). Son interface de programmation est alignée sur la norme automobile OSEK/VDX et son successeur AUTOSAR. Il est développé depuis 2005 au sein de l'équipe *Système Temps Réels* du laboratoire des sciences du numérique de Nantes (LS2N). Trampoline cible principalement les petits systèmes embarqués en terme de mémoire et de puissance de calcul. Ce RTOS est dit statique car il ne permet pas de créer des objets (toutes entités manipulable par le RTOS) en cours d'exécution, tous ces objets doivent être décrits lors de l'étape de compilation. Trampoline utilise un langage dédié pour la description de ces objets mais également pour décrire leurs relations. Dans le contexte des systèmes intermittents construits autour d'un petit microcontrôleur, un RTOS statique offre plusieurs avantages. Tout d'abord, il peut être adapté en fonction des objets présents dans la description de l'application. Ainsi, seul le code nécessaire est généré et intégré. Par exemple, si une application n'utilise pas de messagerie, tous les services du RTOS et les structures de données associées pour gérer les messages peuvent être omis dans la construction de l'exécutable. Ensuite, après avoir fini la description de l'application, un compilateur vérifie la cohérence de la description et génère une grande partie du code de bas niveau spécifique au système d'exploitation, soulageant ainsi le développeur de cette tâche fastidieuse et généralement sujette aux erreurs. Enfin, dans le cas de Trampoline, le langage de configuration et son compilateur sont extensibles. Cela facilite l'ajout de nouveaux objets lors de l'extension du système d'exploitation, ce que nous avons fait ici pour y intégrer notamment les *steps*, partie centrale de *RESURRECT*.

Architecture de trampoline

Comme illustré en figure 3.5, l'architecture de Trampoline est composée de trois blocs. Tout d'abord, le Board Support Package (BSP) constitue le premier bloc et est dépendant de la plateforme. En effet, les composants du BSP (partiellement écrit en assembleur) décrivent la gestion des appels système via le *System call handler*, la gestion des interruptions via le *External interrupt handler*, la protection mémoire via le *Memory protection manager* et l'exécution des changements de contexte via le *Context switch manager*. Le deuxième bloc, appelé Kernel, contient toutes les fonctions nécessaires aux différents services. Ces composants sont portables, et ne nécessitent donc pas d'être écrits spécifiquement pour

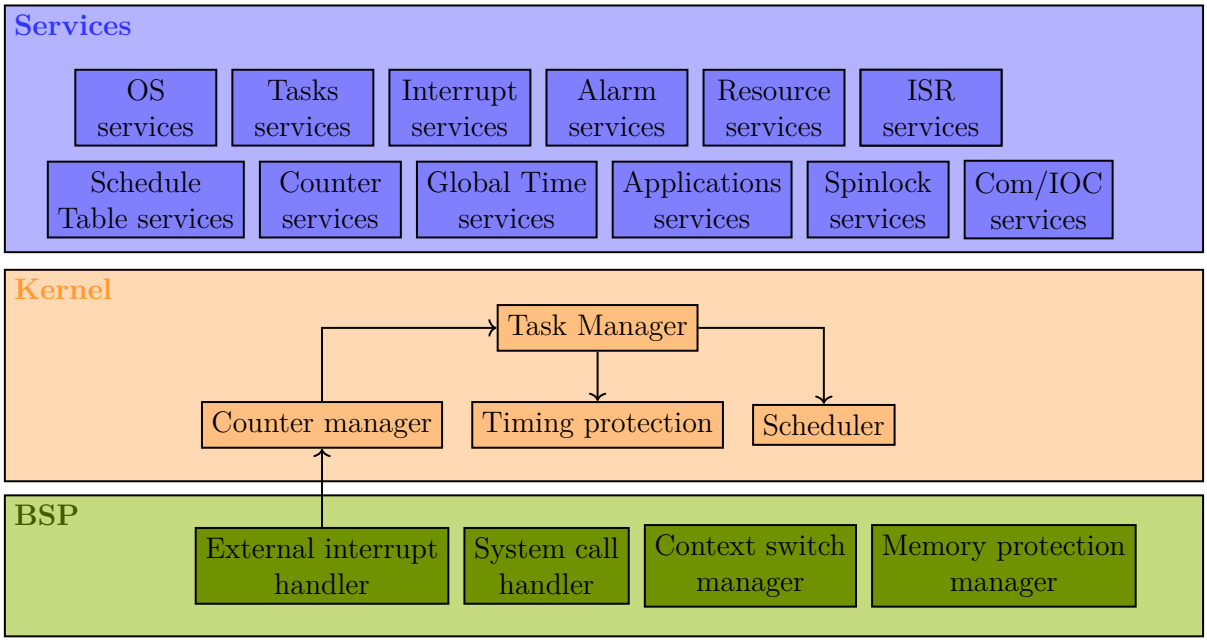


FIGURE 3.5 – Architecture logicielle de Trampoline

chaque plateforme. Enfin le troisième bloc, appelé Services, est constitué des différents services utilisables par l'application. À travers cette interface de programmation, le développeur peut utiliser les services de Trampoline. Comme Trampoline est un OS statique, les services non utilisés ne sont pas intégrés à l'application lors de l'étape de compilation.

Mise en œuvre de trampoline

La construction d'une application via Trampoline se déroule en plusieurs étapes. Comme énoncé précédemment, les objets utilisables par l'OS sont décrits dans un langage appelé OIL (OSEK Implementation Language). Un exemple de la syntaxe est proposé en figure 3.6 avec un exemple d'implémentation d'une tâche. À partir du fichier de description de ces objets, le processus de génération de l'application consiste en une première compilation des objets de l'OS en utilisant un compilateur appelé GOIL afin de générer les structures de données du noyau qui seront nécessaire à l'exécution de l'application. Une deuxième étape de compilation est nécessaire afin de construire l'application à partir des fichiers sources de l'application, des structures de donnée générées et des fichiers sources de l'OS. Ce processus est décrit en figure 3.7.

```
1  TASK t1{
2      PRIORITY p1;
3      SCHEDULE = FULL;
4      AUTOSTART = FALSE;
5      ACTIVATION = a1;
6  };
```

```
1   #include <stdio.h>
2   #include "tpl_os.h"
3
4   int main(void)
5   {
6       StartOS(OSDEFAULTAPPMODE);
7       return 0;
8   }
9
10  TASK(t1)
11  {
12    printf("Hello␣World\r\n");
13    TerminateTask();
14  }
```

FIGURE 3.6 – Exemple de la description d'une tâche en utilisant le langage OIL pour décrire les méta-données. Ici, la priorité de la tâche est définie par le champ *PRIORITY*, la possibilité de préemption de la tâche est définie par le champ *SCHEDULE*, l'activation de la tâche lors du lancement de l'OS est défini par le champ *AUTOSTART* et le nombre maximum de job simultané de la tâche est défini par le champ *ACTIVATION*. Un exemple d'écriture de la tâche *t1* côté application est également donné. Les appels systèmes *TerminateTask* et *StartOS* font parties de l'API de Trampoline.

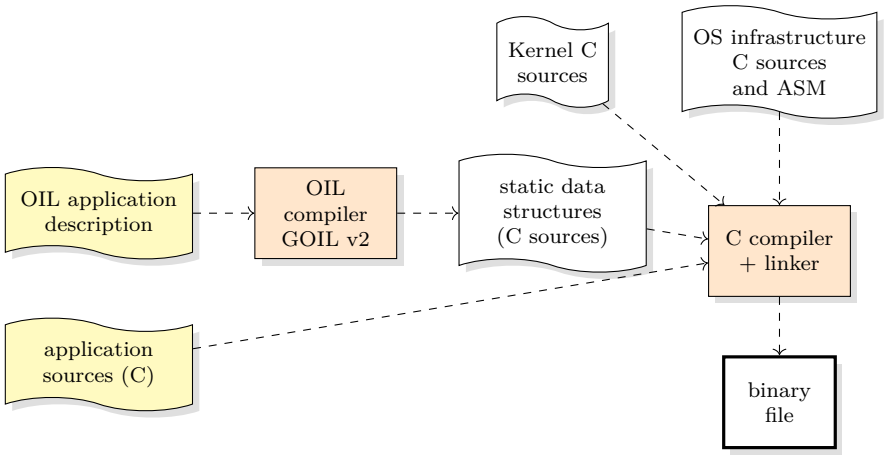


FIGURE 3.7 – Processus de génération d'une application via Trampoline

3.6.2 Intégration de *RESURRECT* dans trampoline

Notre implémentation de *RESURRECT* dans Trampoline vise à minimiser le nombre d'objets utilisés afin de minimiser la taille du code et des données ainsi que le nombre d'appels systèmes car ils sont plus coûteux que de simples appels de fonction. L'implémentation fonctionne de la manière suivante : le système contient une tâche spéciale appelée *resurrect*, qui est activée chaque fois qu'un *step* est élu. Cette tâche sert de fonction principale, c'est-à-dire de point de départ pour l'exécution du *step*. Quand un *step* est élu, le point d'entrée de cette tâche est modifié pour pointer sur la fonction associée au *step* élu. Cette fonction à son tour va appeler les sous-fonctions associées à chaque activités du *step* en respectant l'ordonnancement généré hors-ligne. Les activités matérielles asynchrones qui s'exécutent en parallèle sont déclenchées par l'une de ces sous-fonctions. Lorsqu'elles se terminent, elles déclenchent une interruption et la routine associée définit un événement qui est ensuite lu par la tâche principale. Il n'y a qu'une seule exception : les alarmes utilisées pour des activités périodiques ne déclenchent pas d'événement, mais activent une tâche auxiliaire qui exécutera une séquence d'activités périodiques. Dans ce cas, le *step* est composé de la tâche principale et d'une ou plusieurs tâches auxiliaires (une par chaîne d'activité périodique). Par construction, les *steps* ont une durée finie. Le nombre de périodes de chaque chaîne périodique inclus dans un *step* est connu au moment de la compilation. Un *step* se termine lorsqu'il atteint la fin de la tâche principale ou lorsque toutes les tâches auxiliaires ont été exécutées le nombre de fois prévu, selon ce qui arrive en dernier.

Extension de l'architecture

Au niveau de l'intégration de *RESURRECT*, l'architecture de Trampoline est légèrement modifiée afin d'intégrer la gestion des *steps*. Le bloc BSP reste inchangé. Les modifications se font au niveau du bloc Kernel et du bloc Services. Nous avons implémenté des services supplémentaires pour l'utilisation de *RESURRECT*. Comme le montre la figure 3.8, deux nouveaux types de services sont disponibles. Les services liés à au démarrage et à l'arrêt de *RESURRECT* sont similaires aux services de démarrage et d'arrêt de Trampoline. La principale différence est l'étape de restauration de la mémoire volatile à partir d'un *checkpoint*. Les services liés à la gestion des *steps* sont similaires aux services de gestion des tâches. Contrairement à l'architecture originelle de Trampoline, avec *RESURRECT*, l'ordonnancement se fait à deux niveaux. D'une part entre les *steps*, à

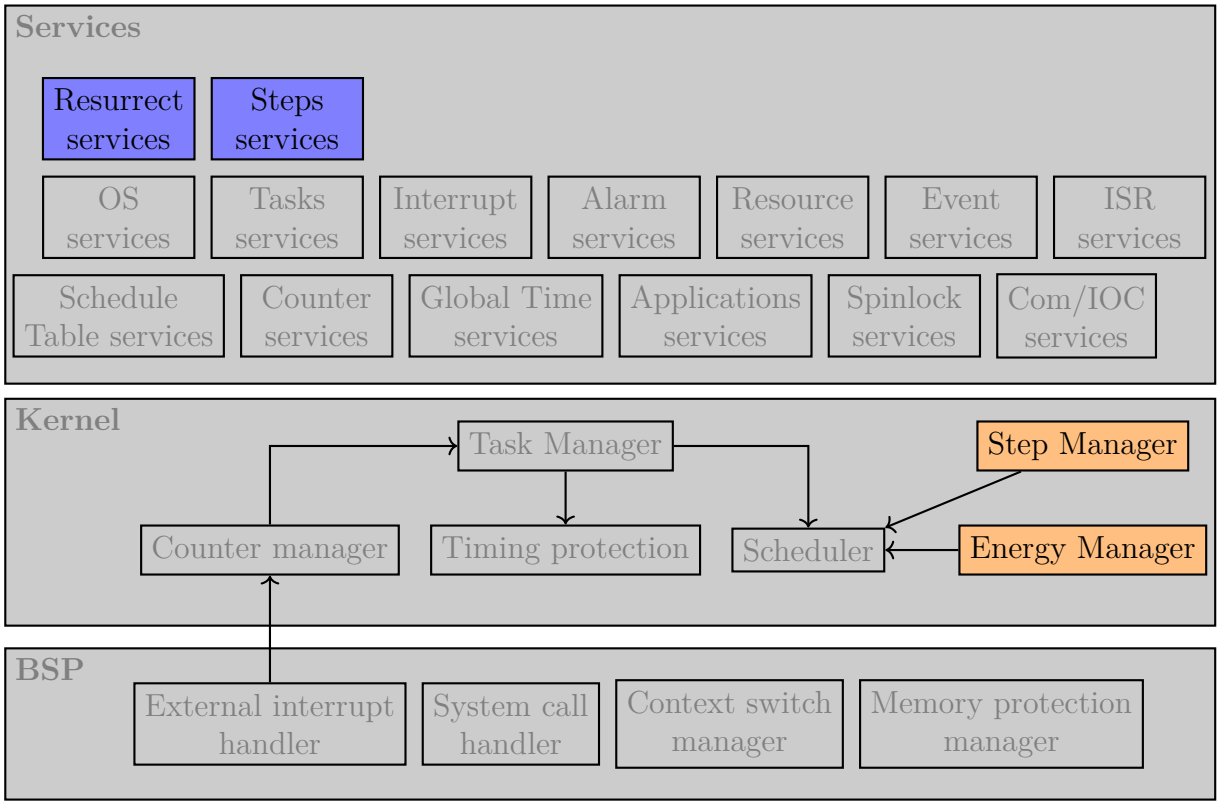


FIGURE 3.8 – Architecture logicielle de Trampoline avec *RESURRECT*

chaque terminaison d'un *step*, le *step* suivant est sélectionné en fonction du niveau actuel d'énergie et de l'état courant du système. Si dans un *step*, des tâches se font concurrence pour les ressources partagées ou si elles sont périodiques, l'ordonnanceur à priorité fixe de Trampoline est utilisé. Des fonctions de supervision de l'énergie disponible sont également fournies dans le noyau pour permettre la sélection des *steps*.

Extension du langage de configuration

La manière naturelle de décrire les *steps* est d'étendre le langage de configuration avec 2 types : *state* et *transition*. Ensemble, ces objets permettent de décrire le type de graphe illustré dans les figures 3.2c et 3.4c. La figure 3.9 montre comment utiliser ces objets avec OIL, le langage de configuration utilisé par *Trampoline RTOS*. L'état *E0* est l'état initial du système lors du premier démarrage.

Une structure de données est générée par le compilateur OIL pour stocker l'état actuel du système et la transition en cours d'exécution. Cette structure est stockée dans une mémoire non volatile afin de conserver la trace de son contenu en cas de perte d'alimentation.

```
1 STATE E0 {INITIAL_STATE = TRUE;};
2 STATE E1 {};
3
4 TRANSITION E0_E1 {
5         ENERGY = 2000;
6         FROM_STATE = E0;
7         TO_STATE = E1;
8         ENTRY = E0_E1_FUNC;
9 };
```

```
1   void E0_E1_FUNC(void)
2   {
3       Activity1();
4       Activity2();
5       Activity3();
6       return;
7   }
```

FIGURE 3.9 – Description en langage OIL des états et des *steps*. Les états ont un seul paramètre pour définir l'état initial du système. Les transitions ont plusieurs paramètres pour définir l'énergie nécessaire pour les compléter, l'état initial et final de la transition ainsi que la suite d'activité exécutée. Cette dernière est regroupée dans le champ *ENTRY*. Un exemple de suite d'activité est également donné sur la partie droite de la figure.

Une deuxième structure, en lecture seule, est également générée pour stocker le graphe qui relie *états* et *transitions*. Elle est utilisée par le noyau pour calculer l'ensemble des *steps* éligibles.

Extension de l'OS

Nous avons apporté plusieurs modifications au système d'exploitation. Tout d'abord, nous avons modifié la routine de démarrage qui est appelée au premier démarrage et après chaque perte d'alimentation. Cette routine appelle une fonction pour choisir le prochain *step* à exécuter (voir algorithme 7). Lors de l'élection d'un *step*, le point d'entrée de la tâche principale est mis à jour. Ensuite, le flot de contrôle retourne à l'ordonnanceur classique de Trampoline, qui démarre la tâche principale. Le *step* est alors exécuté comme décrit ci-dessus. Lorsqu'il est terminé, un service dédié est appelé. Ce service appelle également la fonction de sélection du *step* suivant, effectue la mise à jour du point d'entrée de la tâche principale et transfère le flot de contrôle à l'ordonnanceur si un *step* est élu. Chaque fois que la fonction de sélection du *step* est appelée, une mesure de tension aux bornes du supercondensateur alimentant le système est effectuée pour déterminer la quantité d'énergie restante. Cette mesure est effectuée à l'aide de l'ADC. Si la tension est trop faible, aucun *step* n'est éligible. Dans ce cas, le service appelle la fonction *hibernate*. Cette fonction effectue un checkpoint et met le système en mode hibernation, c'est-à-dire à faible consommation d'énergie, pendant une période définie. Le checkpoint n'inclut pas les piles de tâches car, à cet instant, toutes les tâches sont terminées et leur pointeur de

pile sera réinitialisé lors de leur prochaine activation. Il n'est pas non plus nécessaire de sauvegarder les registres du CPU tels que le *program counter* pour les mêmes raisons. Nous ne sauvegardons donc que les sections de données des tâches allouées en SRAM. Une fois le point de contrôle effectué, la RTC est réglée pour réveiller le MCU et le système passe en mode veille à faible consommation d'énergie. Le mode de veille à faible consommation et la période de réveil peuvent être ajustés au moment de la compilation. Nous avons défini des valeurs par défaut pour cibler un mode veille tel que le CPU et la plupart des périphériques internes sont éteints, mais la SRAM et les registres sont toujours alimentés. En ce qui concerne la période pour sortir de l'hibernation, nous l'avons fixé par défaut à 20 secondes.

3.7 Évaluation de *RESURRECT*

3.7.1 Système et outils

Nous avons utilisé la carte d'évaluation MSP430FR5994 launchpad de Texas Instrument. Ce MCU possède 256 kB de mémoire FRAM et 8 kB de SRAM. Il peut fonctionner avec des fréquences allant jusqu'à 24 MHz. Pour les résultats donnés, nous avons fixé la fréquence à 1 MHz pour pouvoir se comparer avec les supports d'exécution pour les systèmes intermittents de l'état de l'art [MCL17 ; Yı+18 ; YCY]. Les applications ont été compilées en utilisant la chaîne de compilation GNU GCC v9.3.1.11 [1]. Nous avons également utilisé un analyseur logique pour obtenir les mesures de temps des exécutions. Le modèle utilisé est le Saleae Logic Pro 16 [2]. En ce qui concerne la récolte d'énergie, nous avons utilisé de la récolte d'énergie par radio fréquence en combinant le transmetteur Powercast TX91501 et la carte d'évaluation Powercast P2110. Le transmetteur TX91501 émet 3 W en continu sur une gamme de fréquence centrée sur 915 MHz.

3.7.2 Évaluation en alimentation continue

Dans un premier temps, nous évaluons les performances de *RESURRECT* par rapport à *Alpaca* [MCL17], *Ink* [Yı+18] et *ImmortalThreads* [YCY] en alimentation continue.

1. En utilisant le flag -Og pour l'optimisation afin d'avoir la même configuration que ImmortalThreads [YCY].
2. `https://usd.saleae.com/products/saleae-logic-pro-16`

Alpaca est un support d'exécution sans *checkpoint* pour les systèmes intermittents. Il se base sur une approche par tâche et assure le progrès de l'application à cette granularité.

InK fait partie des premiers support d'exécution intermittent intégrant un modèle d'exécution réactif. Comme pour *Alpaca*, il se base sur un modèle *task-based*. Pour assurer la cohérence mémoire, *InK* utilise un système de double tampon pour chaque variable en mémoire non-volatile.

Enfin, *ImmortalThread*, qui est un support d'exécution avec des checkpoints, promet un très faible surcoût lié au checkpointing en ne sauvegardant que le *program counter*. Leur modèle d'exécution permet de supporter complètement le modèle d'exécution réactif, notamment en ajoutant la possibilité de faire du multi-threading.

Nous utilisons trois applications différentes avec des complexités algorithmiques variées. Le benchmark *bitcount* (BC) consiste à compter le nombre de 1 dans une chaîne de bits en utilisant plusieurs méthodes. Le benchmark *activity recognition* (AR) consiste à détecter si le système est à l'arrêt ou en mouvement en utilisant un accéléromètre. Les données d'entrée sont simulées en utilisant une génération aléatoire avec un *seed* fixe de manière à ce que la séquence de nombres aléatoires soit toujours la même. Enfin le benchmark DNN consiste à reconnaître un chiffre sur une image en utilisant un réseau de neurone profond. L'image d'entrée est stockée en mémoire.

InK et Alpaca sont des supports d'exécution intermittente dits *task-based*, c'est-à-dire que l'application est décomposée en tâches atomiques et idempotentes. Pour pouvoir se comparer, nous avons construit les activités de *RESURRECT* sur les tâches définis par InK et Alpaca (celles-ci sont identiques entre InK et Alpaca). De son côté, ImmortalThreads n'utilise pas de tâches mais des mini-checkpoints. Pour InK, son modèle d'exécution nécessite d'avoir les poids du réseau de neurone profond dans une zone mémoire appelée *task-shared* utilisant un double buffering. Hors le MSP430FR5994 n'a pas assez de mémoire pour accueillir ces poids deux fois. Nous n'avons donc pas pu implémenter cette application avec InK mais nous avons récupéré les temps d'exécution issus de [YCY] pour la figure 3.10. Concernant Alpaca, nous avons utilisé l'implémentation issue de [GLB19] qui utilise des techniques de poursuite de boucle (loop-continuation) en enlevant les écritures après lecture (write-after-read (WAR)) à la main. Cela viole le modèle d'exécution de Alpaca.

La figure 3.10 illustre le temps d'exécution de chaque benchmark pour les différents support d'exécution et met en évidence le surcoût dû au support d'exécution intermittent. Le temps d'exécution lié à l'application est en violet (il est le même pour tous) alors que

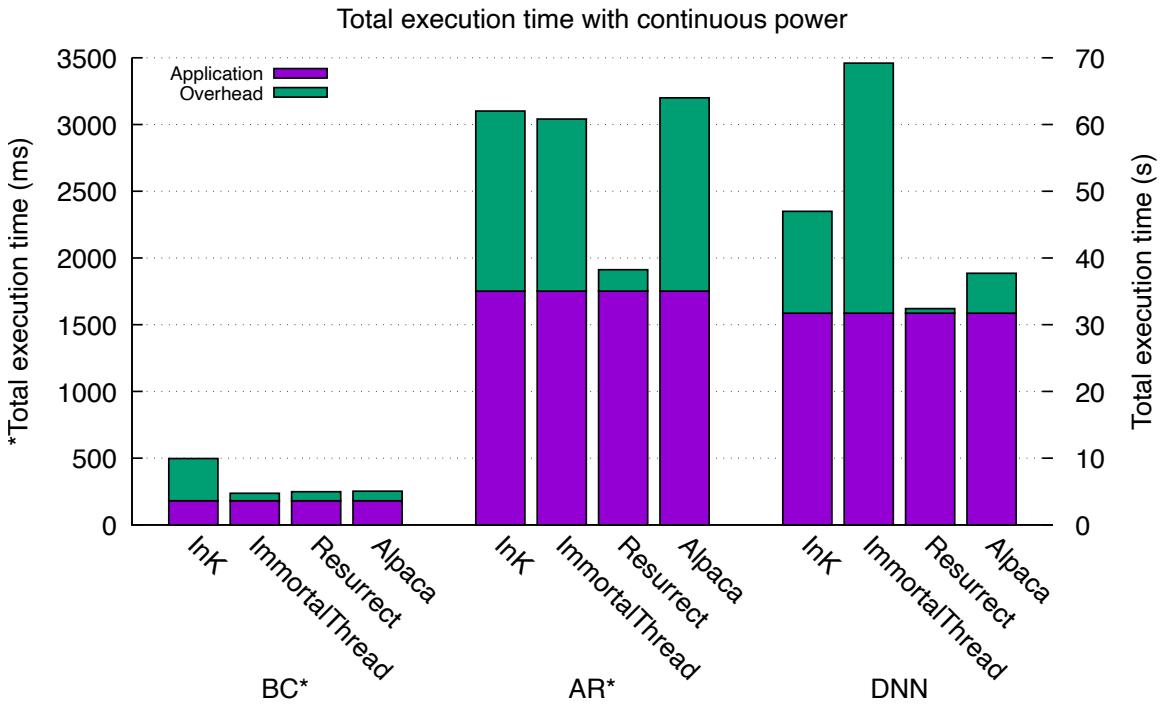


FIGURE 3.10 – Temps d'exécution et contribution des différents support d'exécution pour les 3 benchmarks (BC, AR, DNN). Pour les benchmarks BC et AR, l'échelle de temps est en milliseconde (échelle de gauche et annotée par le symbole *) alors que pour le benchmark DNN, l'échelle de temps est en seconde (échelle de droite). La partie en violet est le temps d'exécution de l'application alors que la partie en vert est le surplus lié au support d'exécution intermittent.

le temps d'exécution lié au support d'exécution est en vert.

On observe que *RESURRECT* a globalement un surcoût plus faible que les différents support d'exécution. Il n'y a que pour le benchmark BC où *RESURRECT* reste à un temps d'exécution équivalent par rapport aux autres support d'exécution. Cette diminution du temps d'exécution est justifiée par l'ajout du modèle de consommation d'énergie dans le système. En effet, avec *RESURRECT*, on construit à l'exécution la plus grande tâche exécutable avec l'énergie disponible en exécutant les *steps* à la suite. Ainsi, de nombreux *checkpoints* ne sont pas effectués ce qui permet de passer plus de temps et d'utiliser plus d'énergie pour l'application. Pour Alpaca, InK et ImmortalThread, il n'y a pas cette connaissance de la consommation d'énergie et donc du temps restant possible avant une perte d'alimentation ce qui oblige le système à effectuer des checkpoints fréquemment (au niveau de chaque tâche pour Alpaca et InK, et de mettre le *program counter* en NVM

pour ImmortalThread). Cela limite le temps et l'énergie disponible pour l'application. Pour le benchmark BC, comme l'application est très limitée en terme de complexité, les tâches sont très petites en terme de temps d'exécution. En utilisant *RESURRECT* dans ce cas, le système passe environ 30% de temps dans la gestion des *steps*. On remarque que c'est également le cas pour les autres supports d'exécution.

3.7.3 Évaluation en alimentation intermittente

Nous procédons ensuite à l'évaluation dans le cadre d'une alimentation intermittente en nous concentrant sur le benchmark DNN. Concernant la configuration utilisée, nous commençons les mesures avec une supercapacité de 10 mF pleine, et nous mesurons la tension à ses bornes au cours du temps ainsi que le nombre d'inférences effectuées pour différentes distances entre le transmetteur et le système de récolte par radio fréquence.

Le tableau 3.1 présente le nombre d'inférences terminées ainsi que le pire temps d'exécution d'une inférence pour des distances entre le transmetteur radio et le récepteur de 130 cm et 150 cm sur une période de mesure de 1000 s. À ces distances, même si la récolte d'énergie se fait en continu, l'énergie récoltée n'est pas suffisante pour maintenir l'exécution de l'application ce qui entraîne des pertes d'alimentation.

Pour plus de détail, la figure 3.11 illustre l'évolution de la tension d'alimentation du système pour une distance entre le transmetteur et le circuit de récolte de 130 cm. On observe que pour *RESURRECT*, le système ne franchit jamais le seuil de tension où le système n'est plus alimenté. Cependant, des périodes d'hibernation sont présentes lorsqu'il n'y a pas assez d'énergie pour le prochain *step*. Lors de ces périodes, la récolte d'énergie est suffisante pour alimenter le système (qui est en mode hibernation) et pour remplir le condensateur alimentant le système. Concernant la configuration de l'application avec *RESURRECT*, nous avons utilisé quatre niveaux de tension pour la génération des *steps* (3.3 V, 2.95 V, 2.6 V et 2.25 V).

Avec Alpaca et ImmortalThread, le système atteint le seuil de tension de perte d'alimentation et à partir de ce moment, il redémarre dès qu'il est alimenté, et ainsi exécute très peu d'opérations liées à l'application avant de franchir de nouveau le seuil de perte d'alimentation. Avec cette exécution, des opérations non achevées vont être redémarrées au prochain cycle entraînant une consommation d'énergie inutile. *RESURRECT* est plus performant en ce sens car il n'y a pas de réexécution du code, un *step* n'est démarré que lorsqu'il peut se terminer.

FIGURE 3.11 – Évolution de la tension alimentant la plateforme pour l'application DNN avec le transmetteur radio placé à 130 cm. À gauche, l'application utilise *RESURRECT*, au milieu l'application utilise Alpaca et à droite, elle utilise ImmortalThread. Les lignes bleue et rouge correspondent aux tensions limites d'alimentation du MSP430FR5994 (3.3 V et 1.8 V). Les figures du dessus représentent les jobs de l'application DNN ainsi que l'état du système. Pour Alpaca et ImmortalThread, l'état du système signifie un système en fonctionnement ou non. Pour *RESURRECT*, comme le système ne perd pas l'alimentation, nous indiquons à la place les périodes d'hibernation, ainsi un état haut signifie que le système est en hibernation. Pour Alpaca et ImmortalThread, nous avons utilisé les codes sources fournis avec les travaux [GLB19] et [YCY] sans modification autre que la mesure du temps des inférences.

	Distance : 130cm			Distance : 150cm		
	# of inference Completed	AET of inference	WCET of inference	# of inference Completed	AET of inference	WCET of inference
RESURRECT	21	44.92	71.6	11	87.97	111.6
Immortal Threads	7	134.80	151.40	3	255.9	324.1
Alpaca	15	66.22	72.87	7	129,47	159.18

TABLE 3.1 – Nombre d'inférences complétées, moyenne du temps d'exécution (AET) et pire temps d'exécution (WCET) d'une inférence (en s) pour le benchmark DNN sous alimentation intermittente.

3.8 Conclusion du troisième chapitre

Dans ce chapitre, nous avons présenté *RESURRECT*, un support d'exécution pour les systèmes intermittents incluant un modèle de consommation d'énergie. Ce support d'exécution est pessimiste par défaut et ainsi assure le non rejeu de code en présence de pertes d'alimentations. Le modèle d'exécution de *RESURRECT* prend en entrée un modèle en réseau de Petri d'une application et génère toutes les stratégies maximisant le progrès de l'application. Nous avons présenté les concepts clés pour le développement de ce support d'exécution ainsi qu'une implémentation complète se basant sur le RTOS Trampoline. Pour cela, une modification du noyau est présentée afin de reproduire *RESURRECT* sur d'autre RTOS. Une évaluation de *RESURRECT* en condition d'alimentation continue et intermittente montre que l'ajout de cette connaissance de la consommation d'énergie permet de limiter les surcoût engendrés par la gestion de l'intermittence.

Deuxième partie

Application : Détection du chant des oiseaux

Chapitre 4

Étude de cas : Application de détection du chant des oiseaux

Dans ce chapitre, nous nous intéressons à une application de détection de chant d'oiseaux. Comme le contexte intermittent ne convient pas à toutes les applications, cette application de détection du chant des oiseaux nous semble être un choix pertinent du domaine d'intervention d'un capteur intermittent. En effet, le traitement du son nécessite plusieurs opérations, de l'acquisition à la détection et à la reconnaissance, et ces opérations peuvent être coûteuses en terme de temps de calcul. De plus, ces capteurs sont généralement placés dans des environnements difficilement accessible. Cela permet également d'être un vecteur de collaboration avec d'autres équipes de recherche, notamment en traitement du signal et en bioacoustique. De plus, cette application est une opportunité de développer une approche de la science au service de la science en décloisonnant les domaines de recherche. Dans un premier temps, nous discutons de la surveillance de la biodiversité et des différents avantages que les systèmes intermittents peuvent apporter à ce domaine. Cela sera présenté en section 4.1. Ensuite, nous rappelons quelques notions de traitement du signal nécessaire à la compréhension des algorithmes utilisés en section 4.2. Nous présentons ensuite deux types d'algorithme en section 4.3 pour détecter des évènements sonores.

4.1 Surveillance acoustique de la biodiversité

L'éco-acoustique est la science qui étudie le rôle du son (principalement produit par l'humain ou ses activités) dans les processus écologiques. Par exemple, à partir d'un paysage sonore et de son évolution, la dynamique des populations et leurs comportements mais aussi le déclin ou la restauration d'écosystèmes sont étudiés.

Le paysage sonore d'un environnement change constamment en fonction des différentes espèces qui l'habitent. Dans ce contexte, notre application se focalise sur une seule caté-

gorie d'animaux : les oiseaux. L'éco-acoustique appliquée aux oiseaux a déjà permis de démontrer certains comportements spécifiques liés à l'environnement sonore. Par exemple, certaines espèces commencent à chanter plus tôt dans un environnement bruyant [DFB22].

4.1.1 Bénéfices d'un système autonome

On se propose de définir l'autonomie d'un système selon trois critères :

— autonomie énergétique : cela implique que le système est sans-fil. Il utilise l'énergie de l'environnement pour s'alimenter sans être relié à une installation électrique ;
— autonomie calculatoire : cela implique que le système est capable de traiter les données à la volée afin de transmettre le résultat du traitement ;
— autonomie opérationnelle : cela implique qu'aucune intervention humaine n'est nécessaire lors du fonctionnement nominale du système ;

Historiquement, les capteurs éco-acoustiques n'étaient pas du tout autonomes : il s'agissait simplement d'enregistreurs alimentés par batterie qui effectuaient des conversions analogique-numérique et le stockage des données audio sans compression sur des cartes SD. C'est toujours le cas pour les deux capteurs éco-acoustiques les plus répandus, à savoir le SongMeter et l'AudioMoth [Hil+18].

En dehors de l'éco-acoustique, un exemple réussi de capteur acoustique temps réel utilisant de la récolte d'énergie est le RFCx Guardian. Cette plateforme conçu par Rain-Forest Connection [1] pour détecter l'exploitation forestière illégale réutilise des téléphones portables munis de panneaux solaire et de microphone directionnel. Cependant, les téléphones portables ne font aucun traitement sur le son enregistré et se contentent de le transmettre.

Une autonomie calculatoire permet de limiter les données transmises sur les liaisons radio en déportant une charge de calcul sur le noeud. Ainsi, les données reçues par les chercheurs en éco-acoustique peuvent être directement exploitées ou passées dans une autre pile logicielle nécessitant une plus grande puissance de calcul sans la nécessité d'un pré-traitement chronophage. De plus, réduire les données transmises permet également de réduire la consommation énergétique car l'opération de transmission sur ces systèmes est la plus énergivore. Cette autonomie calculatoire permet également une meilleure confidentialité des données, les données brutes ne sont plus transmises et des opérations de chiffrement peuvent être effectuées sur le capteur.

1. RainForest Connection est une association à but non lucratif, site web : `https://rfcx.org`

	Autonomie Énergétique	Autonomie Calculatoire	Autonomie Opérationelle
Audiomoth	✗	✗	✗
SongMeter	✗	✗	✗
RFCx Guardian	✓	✗	✗
BARD	✓	✓	✓

TABLE 4.1 – Comparaison des différents équipements pour l'éco-acosutique.

Les bénéfices d'un système complètement autonome pour l'acquisition, le traitement et l'envoi de données dans le contexte d'une application de détection du chant des oiseaux sont multiples. Tout d'abord limiter l'intervention humaine sur les sites d'écoute permet de diminuer l'impact sur la mesure liée à ces interventions. Par exemple, venir sur site régulièrement pour changer les batteries ou récupérer les cartes SD peut laisser un sillage facilitant l'entrée de prédateurs. De plus, le transport vers les sites d'observations est énergivore, ce qui est particulièrement le cas pour les sites d'observations de la biodiversité [Mai+22]. Enfin, les données reçues sont transmises régulièrement, permettant un suivi de l'évolution du paysage acoustique avec un délai borné.

Cette proposition de capteur, et nous appelons notre prototype du chapitre 5 *BARD* (Bio Acoustic Recognition Device), offre un nouvel axe de recherche prônant non pas la minimisation de la consommation de l'énergie mais la création de capteurs pérennes [Los+21].

Gestion de l'intermittence

La fréquence et la durée des périodes d'intermittence dépendent de deux facteurs : 1) l'énergie requise par le capteur autonome pour effectuer ses tâches (échantillonnage, numérisation, classification, transmission) et 2) l'énergie ambiante captée par le dispositif de collecte et stockée sous forme de charge électrique dans le supercondensateur. Si le premier facteur est relativement prévisible, le second ne l'est généralement pas. Si on prend l'exemple de l'énergie solaire, l'énergie récoltée est à peu près proportionnelle à l'ensoleillement. La prédiction fine de la capacité de récolte est un problème difficile qui sort du cadre de ces travaux. La figure 4.1 montre l'irradiance de deux jours du même mois (7 et 8 novembre 2019) à partir de l'ensemble de données d'Oak Ridge [MA07] et illustre la nature imprévisible de la récolte de l'énergie solaire. On observe que sur deux jours consécutifs l'irradiance est très variable et imprévisible, ainsi il est difficile d'obtenir

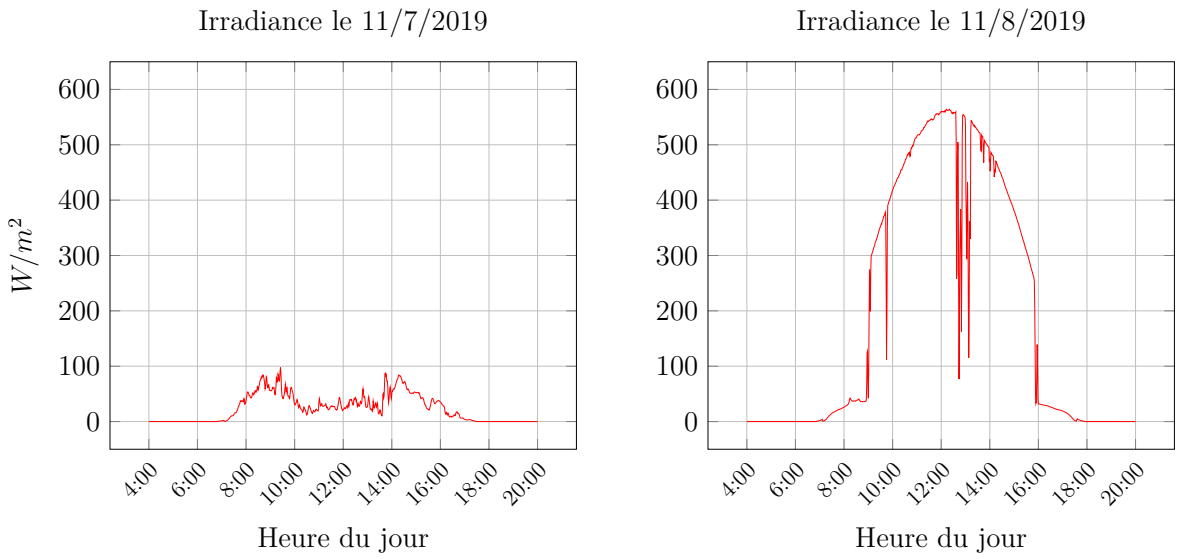


FIGURE 4.1 – Irradiation horizontale globale à partir de l'ensemble de données de Oak Ridge sur deux jours consécutifs, montrant la grande variabilité de la collecte de l'énergie solaire.

une autonomie opérationnelle sans une exécution intermittente.

4.2 Prérequis

On se propose dans cette section de définir les notions de base de traitement du signal qui vont être nécessaires pour la compréhension des algorithmes utilisés pour la détection du chant des oiseaux.

Le traitement du signal est la discipline se focalisant sur l'analyse et la manipulation de signaux. Un signal est une représentation mathématique d'une grandeur physique, cependant dans la plupart des cas, les signaux traités sont des signaux électriques ou devenus électriques par le biais d'un capteur. En se basant sur l'exemple de notre application de détection du chant des oiseaux, un microphone est utilisé afin de transformer l'onde acoustique en signal électrique qui sera alors numérisé puis analysé.

4.2.1 Domaine fréquentiel

Un signal temporel tel qu'un signal sonore, peut être décomposé en petite partie oscillant à une fréquence donnée. On appelle alors ces petites parties du signal les composantes de fréquence, et chaque partie possède une fréquence propre. Cette fréquence spécifique

détermine la vitesse d'oscillation de la composante du signal et est exprimée en Hertz (Hz). Ces composantes de fréquence possèdent également une amplitude qui détermine leurs intensités et une phase qui détermine où commence le cycle d'oscillation par rapport à une référence.

Le domaine fréquentiel d'un signal correspond à la représentation du signal selon toutes ces composantes de fréquence. On appelle spectre de fréquence d'un signal la représentation de l'amplitude et de la phase en fonction de la fréquence pour chaque composante de fréquence. Pour la suite, nous nous intéresserons uniquement à l'amplitude des composantes de fréquence.

La transformation d'un signal temporel vers sa représentation fréquentielle peut se faire en utilisant la transformation de Fourier, qui permet de décomposer le signal en une combinaison de fonctions trigonométriques simples. De plus, nous nous intéressons à la représentation discrète du domaine fréquentiel d'un signal. La pertinence de cette représentation discrète réside dans le fait qu'une part significative du traitement des signaux est désormais réalisée dans le domaine numérique. Cela implique que non seulement la représentation du signal dans le domaine temporel est discrétisée (par exemple, l'échantillonnage du son par l'ADC), mais également que les représentations dans le domaine fréquentiel le sont. On parle alors de transformation de Fourier discrète (TFD). L'algorithme de la transformée de Fourier rapide (FFT) permet d'avoir les mêmes résultats qu'une implémentation naïve de la TFD avec une complexité algorithmique moindre.

Enfin, la transformée de Fourier à court terme discrète (STFT) est une technique permettant de visualiser les composantes de fréquences d'un signal au fur et à mesure qu'il évolue dans le temps. En pratique, cela revient à découper un signal en segments de même durée et de calculer la transformée de Fourier sur chaque segment. Un exemple de STFT est donné en figure 4.2.

4.3 Algorithmes de détection

4.3.1 Per-Channel Energy Normalization (PCEN)

PCEN [Los+19] est un algorithme de détection conçu pour être robuste aux variations d'intensité sonore. Il fusionne la réduction du bruit de premier plan avec l'élimination du bruit de fond, augmentant ainsi la précision de la détection. PCEN est une méthode originellement utilisée dans la reconnaissance de la parole. Il permet notamment d'aider

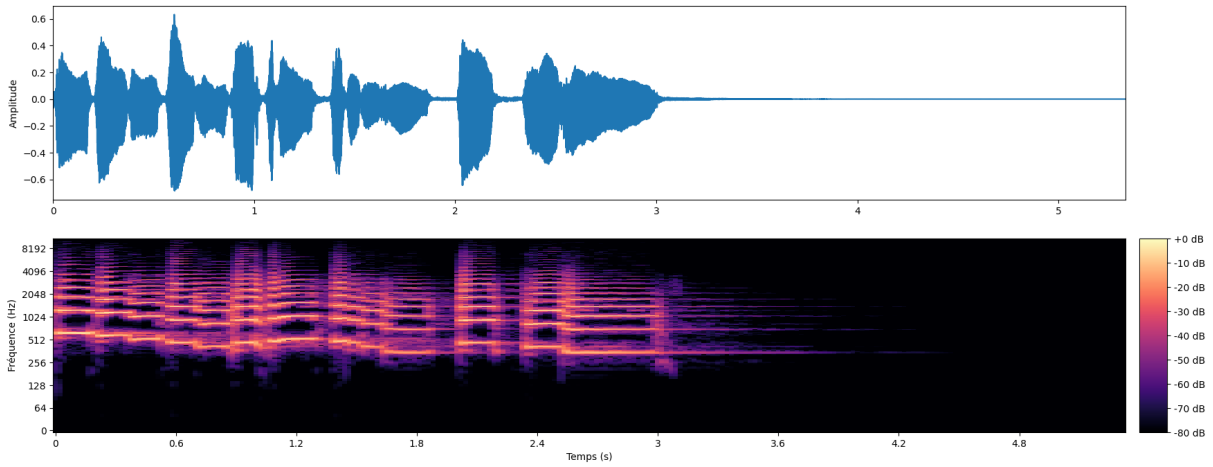


FIGURE 4.2 – Enregistrement de trompette monophonique et sa STFT. Le signal sonore enregistré est sur la figure du haut, alors que sa STFT est présentée sur la figure du bas. Pour la STFT, l'échelle des fréquences est logarithmique et l'amplitude est exprimée en décibel.

à la reconnaissance de mots clés dans un environnement sonore bruité [Wan+16]. Une analyse de cet algorithme [Los+19] montre qu'il peut être utilisé notamment comme traitement préliminaire avant un traitement type apprentissage automatique.

PCEN est originellement défini comme suit :

$$PCEN(t,f) = \left(\frac{E(t,f)}{(\epsilon + M(t,f))^{\alpha}} + \delta \right)^{r} - \delta^{r} \tag{4.1}$$

$$M(t,f) = (1-s) \times M(t-1,f) + s \times E(t,f) \tag{4.2}$$

avec $\epsilon, \alpha, \delta, r$ et s des constantes. *E(t,f)* est le spectre de puissance du signal (amplitude en fonction de la fréquence) et *M(t,f)* est une moyenne pondérée du spectre de puissance, avec le poids s qui est utilisé pour donner plus d'importance aux valeurs récentes. Cela permet à l'algorithme de s'adapter aux sons non-stationnaires.

Pour chaque période d'échantillonnage t et pour chaque fréquence f, l'indicateur *PCEN(t,f)* est comparé avec un seuil pour déterminer si un échantillon de son contient ou non un évènement. L'idée derrière PCEN est de comparer chaque échantillon avec une moyenne du bruit ambiant. Ainsi si l'échantillon est comparable au bruit ambiant, celui-ci est traité comme ne comportant pas d'évènement d'intérêt.

On remarque que PCEN contient de nombreux paramètres ajustables ce qui rend son implémentation difficile car son espace de configuration est vaste.

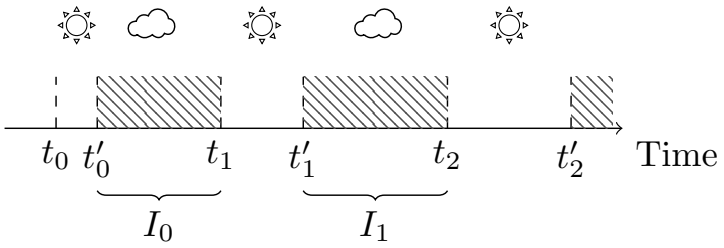


FIGURE 4.3 – Exemple d'exécution pour un capteur intermittent alimenté par énergie solaire. Le capteur est actif sur les périodes $[t_n, t'_n[$ et est éteint sur les périodes $[t'_n, t_{n+1}[$. Les périodes d'intermittence sont notées I_n.

Discussion sur l'implémentation

L'implémentation de PCEN sur un capteur intermittent autonome soulève quelques problèmes.

Premièrement, l'équation régissant PCEN implique t, le numéro d'échantillon. Dans un système non-intermittent, les mesures de t sont effectuées aux instants $time_{NI} = k \times t$, avec k la période d'échantillonnage. Dans un contexte intermittent, l'équation devient $times_I = k \times t + I(t)$, avec *I(t)* la durée cumulée des périodes d'intermittence sur l'intervalle [0,t] comme illustré sur la figure 4.3. Ainsi, dans un contexte intermittent, la validité des équations impliquant le passé en fonction de la valeur de $I(t)$ doit être questionnée car il n'est pas toujours possible d'avoir la durée des intermittences. De plus, le principe de l'algorithme est d'apprendre le bruit ambiant or, avec des périodes d'intermittences trop longues, cet apprentissage devient désuet.

Deuxièmement, le capteur autonome que nous envisageons est conçu autour d'un MCU prévu pour des applications très basse consommation, ce dernier ayant peu de puissance de calcul et une mémoire volatile limitée. Par exemple, une unité de calcul en virgule flottante (FPU) est rarement présente sur le type de MCU envisagé. C'est le cas pour le MSP430 de Texas Instrument qui est le MCU sélectionné pour le prototype de capteur intermittent. En l'absence d'une FPU, le calcul en virgule flottante peut se faire en utilisant des bibliothèques dédiées, mais cela a un coût conséquent en terme de temps d'exécution. Une opération MAC (Mutiply and ACumulate) nécessite 3 µs avec des variables entières et 201.8 µs avec des variables en virgule flottante, soit une multiplication par 67 pour le temps de calcul [2]. De plus, la consommation d'énergie étant proportionnelle au temps de calcul, l'utilisation du calcul en virgule flottante n'est pas compatible avec les objectifs

2. Testé sur un MSP430FR5994 à 8 MHz en utilisant le compilateur MSPGCC version 4.6.3 avec l'option -O0, c'est-à-dire sans optimisation.

de notre capteur intermittent sans circuit FPU intégré. Pour contourner cette difficulté, l'utilisation du calcul en virgule fixe est une solution mais possède quelques inconvénients. Tout d'abord, la précision sera fixée au préalable et les arrondis peuvent s'accumuler rapidement dégradant la précision du calcul. C'est un problème pour PCEN car le vecteur M demande une accumulation à chaque itération.

NV-PCEN

NV-PCEN (Non Volatile PCEN) est une version légèrement modifiée de PCEN. On enlève la plupart des paramètres qui sont consommateurs en temps de calcul. Ainsi, nous pouvons jouer sur le compromis entre la capacité des calculs et la précision des résultats.

$$\text{NV-PCEN(t, f)} = \frac{E(t,f)}{M(t-1,f)} \tag{4.3}$$

Avec M la moyenne des spectres de puissance des échantillons précédents.

$$M(t,f) = \frac{1}{t} \times \sum_{\tau=0}^{t-1} E(\tau,f) \tag{4.4}$$

Pour réduire la taille mémoire nécessaire au calcul de M car il faudrait garder en mémoire les t derniers échantillons du spectre de puissance, on peut utiliser une version récursive définie comme :

$$M(t,f) = \frac{1}{t} \times ((t-1) \times M(t-1,f) + E(t-1,f)) \tag{4.5}$$

Contrairement à l'équation (4.2), nous avons enlevé la pondération sur les valeurs moyennées. Nous appliquons à la place le même poids pour chaque valeur. Cela permet de s'affranchir du calcul du poids de chaque trames précédentes. C'est cette version de l'algorithme qui sera implémentée sur la plateforme et qui sera évaluée.

Évaluation de NV-PCEN

Pour évaluer les performances de NV-PCEN, nous considérons l'expérimentation suivante : la détection d'un signal de fréquence fixe mélangé avec du bruit blanc. Le signal à détecter est un signal sinusoïdal. Nous fixons la fréquence fondamentale du signal à 1500 Hz. Le bruit est généré avec une loi uniforme continue sur la plage [-1 ;1]. La fréquence d'échantillonnage est fixée à 8000 Hz. Avec 256 points par trame, une trame correspond à

une durée de 32 ms. Pour PCEN et NV-PCEN, nous faisons varier la taille de l'historique des trames utilisées pour le calcul de la variable M.

La comparaison est faites avec PCEN calculé avec des données en virgules flottantes sur 32 bits alors que NV-PCEN est calculé avec des données en virgule fixe sur 16 bits (1 bit de signe et 15 bits pour la partie décimale). Pour NV-PCEN, le calcul en virgule fixe comprend les opérations comme le FFT et le calcul de l'amplitude à partir du vecteur de nombre complexe résultant. Pour cela, nous avons utilisé une bibliothèque logicielle développée par ARM [3]. Pour PCEN, nous avons utilisé l'implémentation issue de la bibliothèque librosa [Lib], avec les paramètres utilisés dans [Wan+16].

Nous évaluons la détection du signal pour les deux algorithmes en comparant leurs résultats par rapport à un seuil prédéfini. Si à la fréquence de 1500 Hz les algorithmes donnent un résultat supérieur au seuil, nous considérons que le signal est détecté. Comme pour toutes évaluations de détection, les résultats peuvent être divisés en 4 catégories. Il peut s'agir d'une estimation correcte de la présence ou de l'absence de l'événement sonore, respectivement un *vrai positif* et un *vrai négatif*, ou il peut s'agir d'une estimation fausse que ce soit une détection de signal alors que le signal est absent (un *faux positif*) ou une détection manquée alors que le signal est présent (un *faux négatif*). Soit TP, TN, FP et FN respectivement le nombre de vrai positif, vrai négatif, faux positif et faux négatif. La précision d'un algorithme de détection définie la proportion d'estimation qui se révèle correcte et est calculée telle que :

$$precision = \frac{TP}{TP + FP} \tag{4.6}$$

Le rappel définit la proportion d'estimation correcte qui sont identifiés parmi toutes les entrées. Il est calculé tel que :

$$rappel = \frac{TP}{TP + FN} \tag{4.7}$$

Enfin, on considère la métrique F1 tel que la moyenne harmonique de la précision et du rappel. Cette métrique est utilisée pour synthétiser la précision et le rappel dans une seule valeur. Elle est calculée telle que :

$$F_1 = \frac{2 \times precision \times rappel}{precision + rappel} \tag{4.8}$$

3. `https://arm-software.github.io/CMSIS_5/DSP/html/index`

La figure 4.4 illustre la précision en fonction du rappel de PCEN et NV-PCEN pour différentes tailles d'horizon. Ces courbes sont obtenues en faisant varier le seuil de détection et avec un rapport signal sur bruit de $-13\,\mathrm{dBm}$. Pour chaque seuil, 1000 essais sont testés avec et sans le signal à détecter. On observe que pour PCEN, la taille du contexte n'influe pas sur le résultat de la précision et du rappel. Cela est surement dû au fait que le bruit ambiant est un bruit blanc (généré aléatoirement), et donc, l'algorithme l'apprend très vite. Pour NVPCEN, les performances sont légèrement inférieures à PCEN mais augmentent avec l'augmentation de la taille de l'horizon. Pour NVPCEN, on observe également que l'effet de l'augmentation de la taille de l'horizon sature vite, car pour un horizon de 1000 trames, le résultat est pratiquement égale à celui pour un horizon de 100 trames. Avec une fréquence d'échantillonnage de $8\,\mathrm{kHz}$ et 256 échantillons par trame, cela représente respectivement $32\,\mathrm{s}$ et $3.2\,\mathrm{s}$.

À partir de ces courbes, nous pouvons extraire un seuil optimal de détection en prenant le seuil tel que la métrique F_1 est la plus grande.

La deuxième évaluation consiste à tester la robustesse des algorithmes face à un bruit de plus en plus fort par rapport au signal. Pour cela, on fixe le seuil de détection au seuil optimal précédent mesuré à l'aide de la figure 4.4 et nous faisons varier le SNR. La figure 4.5 illustre la métrique F_1 en fonction du rapport signal sur bruit. On observe que NVPCEN suit la même dynamique que PCEN, lorsque le SNR diminue, la métrique F_1 diminue également. NVPCEN est légèrement moins performant que PCEN. Le score F_1 commence à se dégrader pour les deux algorithmes lorsque le SNR est inférieur à $-10\,\mathrm{dBm}$, ce qui correspond à une puissance du bruit 10 fois plus grande que la puissance du signal.

Ces deux évaluations permettent de montrer que la modification de PCEN pour pouvoir être calculé sur des systèmes à faible puissance de calcul n'altère que légèrement les performances. Cependant, l'effet de l'accumulation d'un horizon n'est pas décisif après 100 trames. Cela limite l'intérêt d'une exécution intermittente pour cet algorithme avec notamment, la sauvegarde de l'horizon à travers les cycles d'exécution, car les 100 trames sont rapidement acquises (en $3.2\,\mathrm{s}$ pour une fréquence d'échantillonnage de $8\,\mathrm{kHz}$).

4.3.2 Time Frequency Second Derivative (TFSD)

Contrairement à PCEN, l'indice acoustique TFSD permet d'avoir une vision statistique d'un environnement sonore. L'indice acoustique TFSD n'a pas été originellement conçu pour la détection d'évènement mais pour quantifier le caractère agréable d'un en-

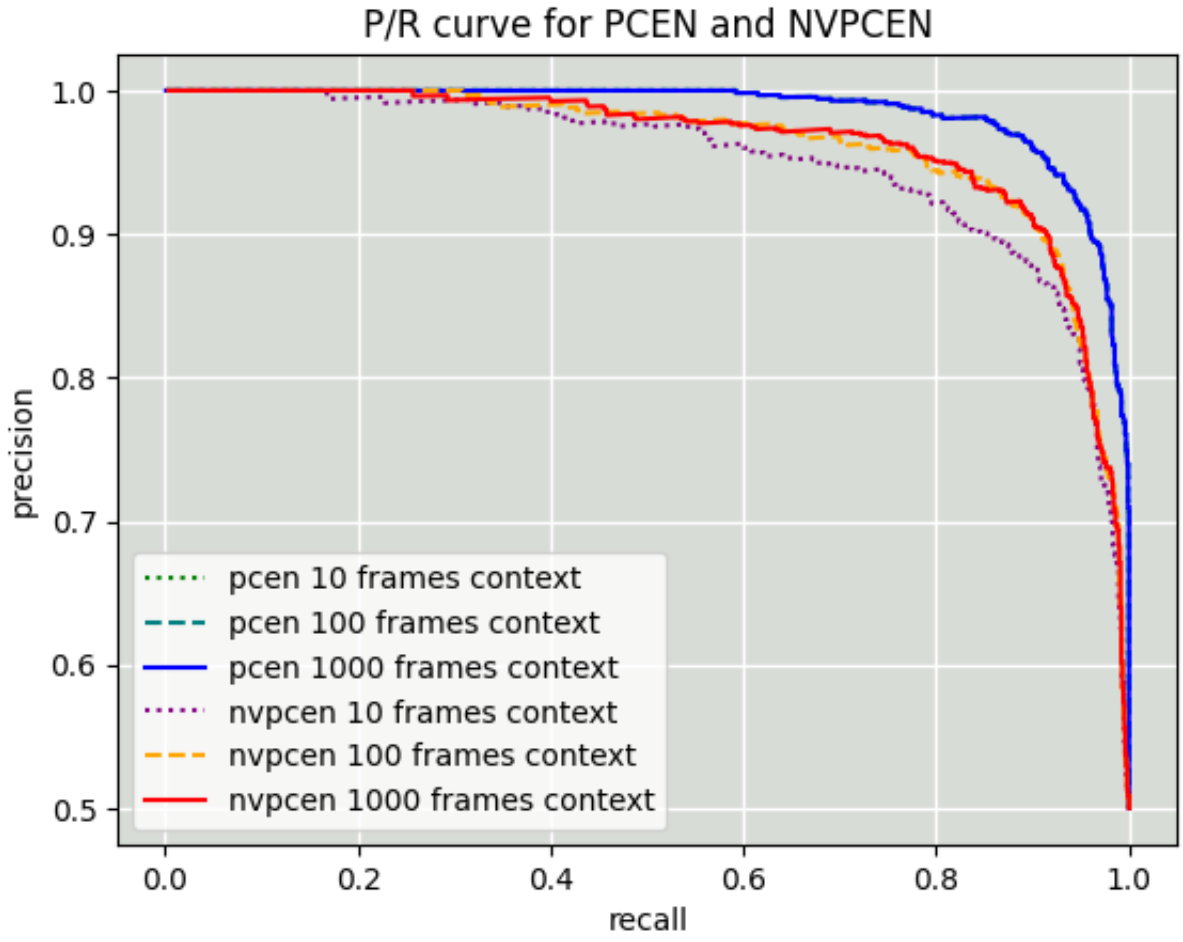


FIGURE 4.4 – Courbes de précision/rappel pour PCEN et NVPCEN pour différentes tailles d'horizon.

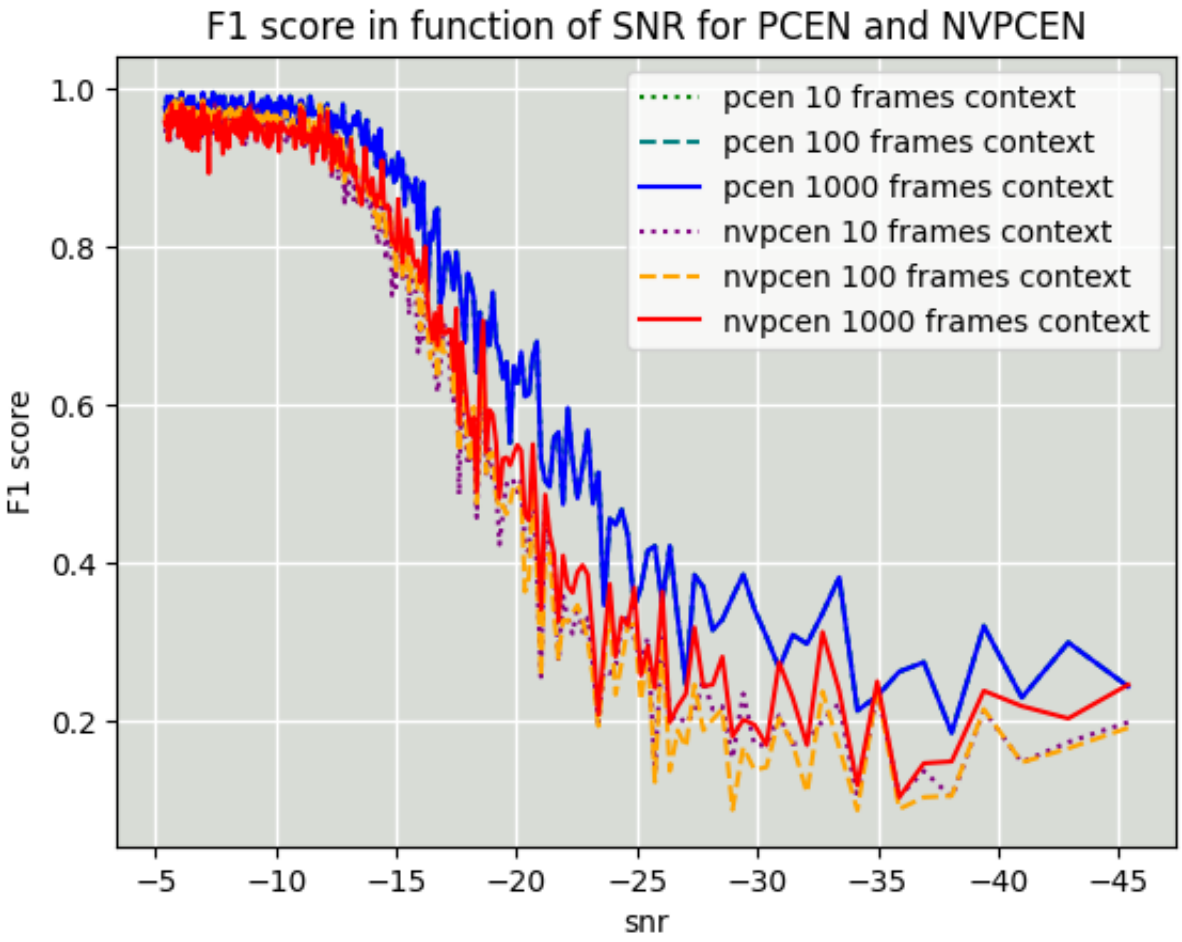


FIGURE 4.5 – Métrique F_1 en fonction du SNR de PCEN et NVPCEN pour différentes tailles d'horizon.

vironnement sonore. Il a été prouvé qu'il existe une forte corrélation entre cet indice acoustique et la présence d'oiseaux [Aum+17]. De plus, il est particulièrement intéressant de détecter et d'analyser des données telles que les chœurs d'oiseaux (le chœur de l'aube désigne le chant des oiseaux au matin) pour caractériser l'influence de certains paramètres sur leurs comportements [BMF14]. TFSD, comme d'autres indices bioacoustiques [Sue+14], est un indice utile pour saisir avec une seule valeur, de manière normalisée et objective, une communauté acoustique. Il peut être utilisé de différentes manières pour surveiller l'activité vocale des oiseaux dans le temps et l'espace, comme la synchronisation des chœurs d'oiseaux sur des rythmes circadiens ou saisonniers, ou la comparaison de différentes communautés d'oiseaux.

Le principe de TFSD est d'évaluer la variation d'énergie d'une bande de fréquence spécifique dans le temps, normalisée par la variation d'énergie totale. La bande de fréquence choisie peut varier en fonction de l'espèce à détecter. Le TFSD est calculé comme suit :

$$TFSD(t,f) = \frac{\sum_{f_{bandmin}}^{f_{bandmax}} |\frac{d^2E(t,f)}{dtdf}|}{\sum_{fmin}^{fmax} |\frac{d^2E(t,f)}{dtdf}|} \tag{4.9}$$

Avec E le spectre de puissance du signal. Dans la description originale de l'indice acoustique TFSD, les bandes de fréquence choisies sont celles entre 31.5 Hz et 16 kHz. Pour la détection de chant d'oiseau, ces bandes de fréquence peuvent être adaptées.

L'implémentation de TFSD est plus simple que celle de PCEN notamment car TFSD ne nécessite pas un historique de FFTs. Comme pour PCEN, l'utilisation du calcul en virgule fixe est une solution pour pallier au manque de FPU.

Sur la figure 4.6, l'indice acoustique TFSD détecte la plupart des chants d'oiseaux sans différenciation sur l'espèce (le son de 40 s comprend 5 espèces différentes et il est annoté par un ornithologue).

4.4 Conclusion du quatrième chapitre

Dans ce chapitre, nous avons présenté les différentes notions de bio-acoustique et de traitement du signal afin de concevoir un capteur éco-acoustique intermittent. Le traitement du son nécessite de travailler dans le domaine fréquentiel ce qui requiert potentiellement une capacité de calcul conséquente. Certains algorithmes de traitement du son comme PCEN peuvent être adaptés au paradigme intermittent, et d'autre plus simple comme TFSD peuvent être utilisés tels quels. Cette application de détection peut être

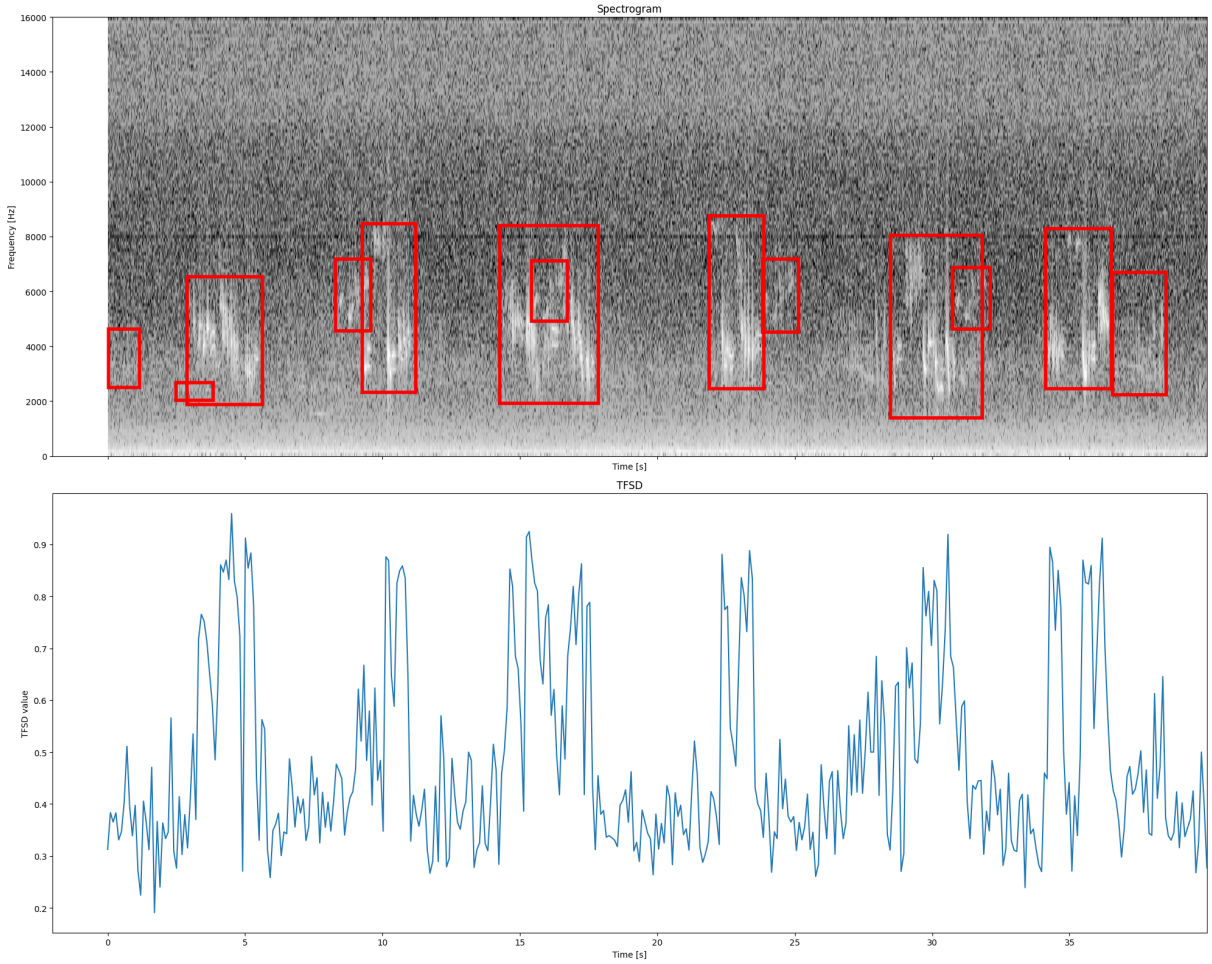


FIGURE 4.6 – STFT d'un son de 40 s ainsi que l'indice acoustique TFSD. La STFT se situe sur la figure du haut et les chants des oiseaux sont annotés par un rectangle rouge. Le TFSD est sur la figure du bas. Pour le TFSD, les bandes de fréquences visées sont entre 1.6 et 8 kHz. On remarque que lorsque l'énergie sur ces bandes de fréquence prédomine, la valeur de l'indice acoustique se rapproche de 1.

étendue pour inclure une étape de classification des espèces lorsque l'indice acoustique indique une forte probabilité de présence d'un événement. Les algorithmes de reconnaissance de l'état de l'art reposent sur l'analyse de la signature fréquentielle des chants d'oiseaux. Ceci peut notamment utiliser des traitements plus poussés comme des traitements d'apprentissage automatique.

Chapitre 5

Prototype de système intermittent

Dans ce chapitre, nous proposons de présenter un prototype de capteur sans-fil intermittent. Ce système embarque une application de détection de chant des oiseaux. Dans la section 5.1, nous décrivons ce prototype. Ensuite, en section 5.2, nous discutons des résultats obtenus à partir de ce prototype. Ce prototype met en œuvre le support d'exécution décrit en chapitre 3.

5.1 Description du prototype

Le capteur proposé est un système de surveillance de la population aviaire. L'objectif principal de ce capteur est de détecter la présence d'oiseaux, indépendamment de leur espèce. Pour cela, le capteur est muni d'un microphone et d'un MCU afin de calculer au plus près du capteur et de manière autonome un indice acoustique qui sera envoyé via une liaison sans-fil. Dans notre cas, cela sera en utilisant la technologie LoRa.

5.1.1 Architecture matérielle

Le prototype créé se compose de plusieurs éléments, avec au centre le MCU MSP430FR5994 pour l'unité de calcul. Ce MCU intègre un accélérateur matériel permettant de faire des opérations vectorielles. Cet accélérateur permet notamment de faire des FFTs sans l'intervention du CPU. Le prototype possède également un composant radio LoRa pour l'envoi de données et un microphone pour l'acquisition de son. Concernant les circuits de récolte d'énergie, le prototype utilise un panneau photovoltaïque et un module Maximum Power Point Control (MPPC) pour extraire l'énergie du panneau photovoltaïque. La sortie du MPPC est directement connectée au supercondensateur alimentant le système (MCU et périphériques externes). La Figure 5.1 illustre l'architecture matérielle du prototype. Le supercondensateur utilisé pour le tampon d'énergie est de 0.1 F.

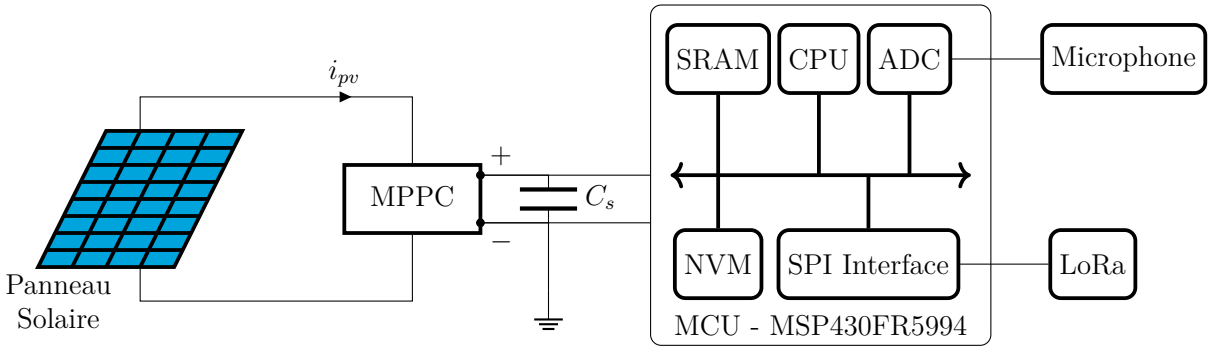


FIGURE 5.1 – Architecture matérielle du prototype de système intermittent développé, *BARD*. Le microphone et la radio sont des périphériques externes au MSP430FR5994.

5.1.2 Architecture logiciel

Ce prototype de système intermittent embarque une application de détection du chant des oiseaux à l'aide de l'indice acoustique TFSD décrit précédemment au chapitre 4. En plus de cette application, il embarque également le support d'exécution *RESURRECT* présenté au chapitre 3.

Application

Comme le montre la figure 5.2, l'application est constituée de deux tâches, l'une pour la chaîne de traitement des données récupérées par le microphone et l'autre pour la gestion de l'acquisition des données. La tâche de gestion de l'acquisition s'occupe de mettre en route l'ADC et le DMA. L'ADC, à l'aide d'un timer, va échantillonner le signal analogique reçu grâce au microphone. Le DMA est utilisé pour directement transférer les échantillons vers une zone mémoire prédéfinie. Cette tâche est activée périodiquement grâce à une alarme. La période est fixée à 33 ms, ce qui correspond à 256 échantillons avec une fréquence d'échantillonnage de 8 kHz plus un léger délai afin de s'assurer que tous les échantillons sont bien prêts (principalement afin d'attendre le dernier transfert DMA de l'ADC vers la mémoire volatile). À l'issue de l'acquisition des 256 échantillons de son, une interruption est levée par le DMA afin de prévenir que la zone mémoire dédiée au stockage du son est prête. Un système de double tampon pour les échantillons de son est mis en place afin de paralléliser l'acquisition et le traitement du son de l'échantillon précédent. Dans cette interruption, l'acquisition est interrompue et les tampons sont échangés. De plus, un évènement est initié afin de prévenir la deuxième tâche. Cette deuxième tâche, qui va effectuer le traitement du son, va accuser réception de l'évènement et procéder au calcul

de la FFT et de l'indice TFSD pour les échantillons précédemment prélevés. L'envoi des résultats par liaison radio via la puce LoRa n'est effectué que lorsque qu'un certain nombre d'indice acoustique a été calculé. Lorsque ce nombre est atteint, les indices acoustiques sont moyennés et la valeur maximale du TFSD est récupérée, puis envoyée. Nous avons fixé le nombre à 32, ce qui correspond à environ une seconde de son. Ainsi, toutes les secondes, 32 indices acoustiques TFSD sont calculés et la moyenne et la valeur maximale parmi ces 32 valeurs sont envoyées via le périphérique LoRa.

Support d'exécution

Le système utilise *RESURRECT* et les séquences d'activités sont générées hors ligne. Le modèle de l'application, donné en figure 5.3, est utilisé avec les algorithmes décrits au chapitre 2 afin de générer ces séquences. Au préalable, afin de nourrir le modèle formel de l'application, nous avons fait des mesures de pente de tension pour chaque activité présente selon le protocole présenté en section 1.2.3. Pour obtenir le pire temps d'exécution de chaque activité, nous avons mesuré à l'aide d'un analyseur logique le temps nécessaire pour effectuer chaque activité. Nous savons que cette méthode va donner un temps très optimiste pour le WCET des activités et qu'il existe des méthodes et outils beaucoup plus performant [Wil+08], mais cela est en dehors du champ des travaux de cette thèse. La capacité du supercondensateur utilisée étant relativement grande, nous avons utilisé un seul seuil de tension fixé à 2.2 V, soit une exécution de l'application dès que le tampon d'énergie est à au moins environ 20% de sa capacité. Ce choix d'un seul niveau de tension est également motivé par le fait que les activités sont très courtes et consomment très peu d'énergie, ce qui a pour effet de générer des traces d'exécution conséquentes. Le graphe états / transition obtenu est donné en figure 5.4.

5.2 Résultats

A l'aide de notre prototype de système intermittent, nous avons collecté les valeurs de l'indice acoustique TFSD sur une période de plusieurs jours. À titre de comparaison, nous avons en même temps et sur le même lieu placé un appareil passif d'écoute de l'environnement sonore. Cet appareil est un SongMeter 4 [1].

Nous comparons l'indice acoustique calculé en ligne par notre prototype et envoyé via liaison radio LoRa et l'indice acoustique calculé hors-ligne, après extraction des données

1. `https://www.wildlifeacoustics.com/products/song-meter-sm4`

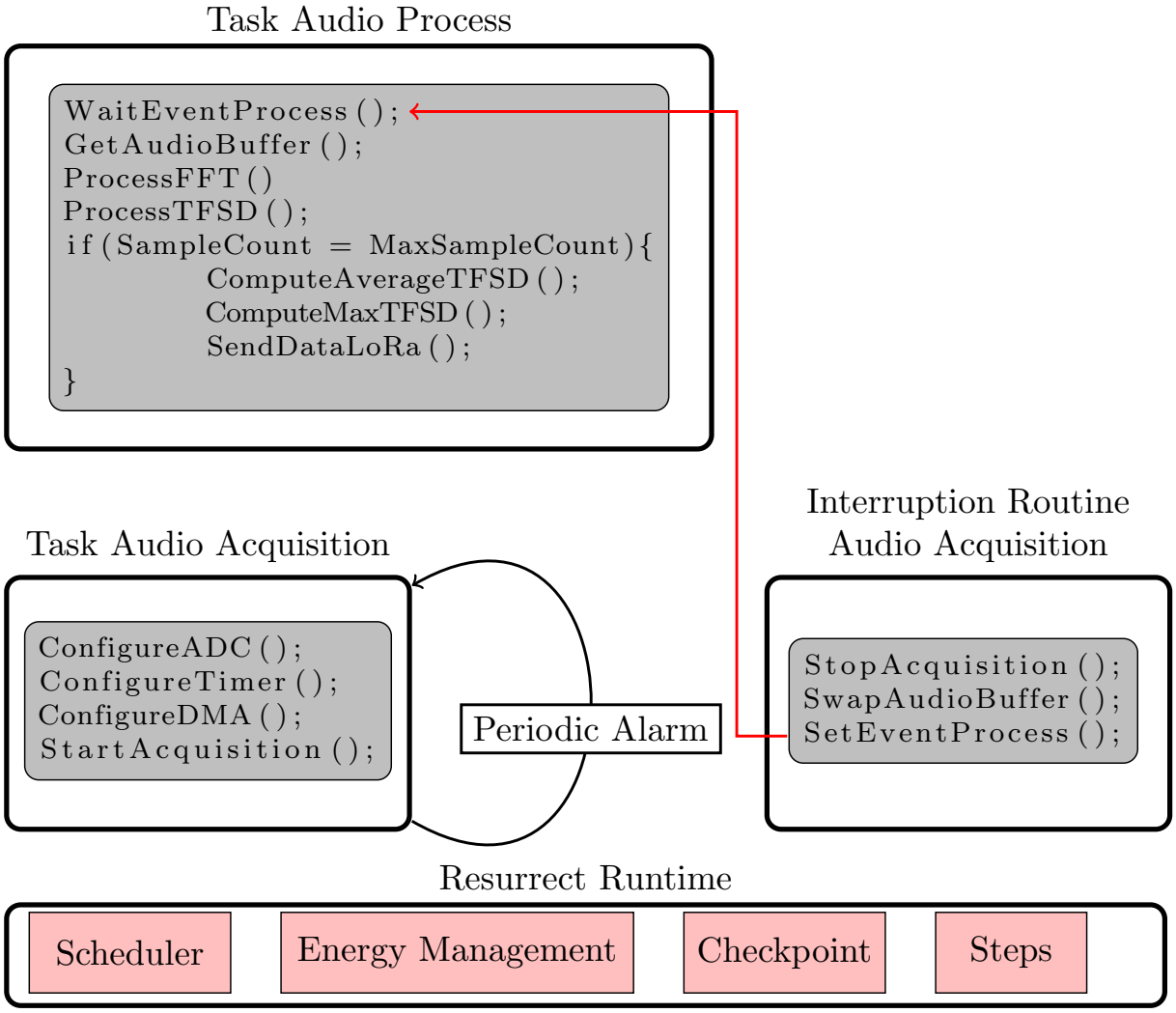


FIGURE 5.2 – Architecture logicielle du prototype de système intermittent développé, *BARD*.

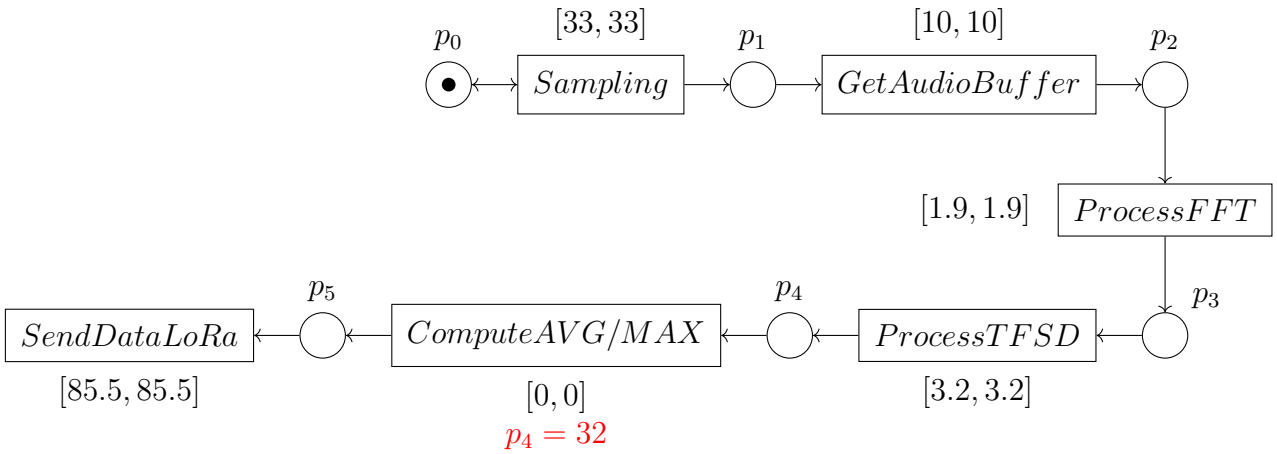


FIGURE 5.3 – Modèle en réseau de Petri de l'application. Pour les intervalles de temps, l'unités est la ms. Comme l'application est linéaire, nous n'avons pas représenté les récompenses pour chaque activités. Pour la transition *Compute AVG/MAX*, une garde (en rouge) conditionne son tir.

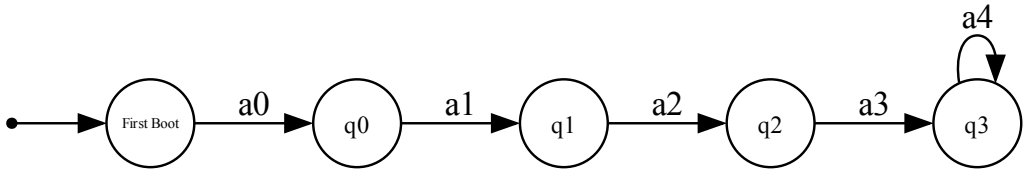


FIGURE 5.4 – Graphe d'états / transitions pour l'application du prototype.

audio sauvegardées sur les cartes SD par le SongMeter 4. Pour le calcul avec les données extraites du SongMeter 4, nous avons formaté les données en virgule fixe sur 16 bits et utilisé une bibliothèque afin d'effectuer la FFT également en virgule fixe car originalement, les données sont sauvegardés en virgule flottante. Les résultats sont donnés en figure 5.5 et figure 5.6. Ces figures représentent la moyenne et la valeur maximale de l'indice TFSD sur 5 jours consécutifs. La moyenne et la valeur maximale sont obtenues à partir de 32 valeurs de l'indice acoustique consécutives. On observe bien que le prototype intermittent est en fonctionnement sur certaines périodes de temps, reflétant le cycle circadien. Certaines valeurs obtenues avec notre prototype sont supérieures à 1, ce qui n'est pas possible avec l'équation (4.9). Cela résulte donc d'une erreur soit dans l'implémentation, soit au cours de la transmission. On observe également une grande différence entre les valeurs obtenues par les deux systèmes. Cela peut venir de la différence entre les microphones des deux systèmes mais également de l'ADC utilisé dans notre prototype qui ne possède que 12 bits de précision. Nous envisageons d'améliorer notre prototype en utilisant un autre microphone et un autre ADC pour augmenter la précision du calcul du TFSD.

5.3 Conclusion

Dans ce Chapitre, nous avons présenté un prototype de système intermittent pour la détection du chant d'oiseau. Ce système repose sur le support d'exécution intermittent *RESURRECT* introduit en chapitre 3. L'intelligence sous-jacente de l'application est rudimentaire, en particulier parce qu'elle ne tient pas compte des moments où les oiseaux sont particulièrement actifs. Cela se voit particulièrement dans la section 5.2. Cependant, le but premier de ce prototype est de fournir un démonstrateur pour le support d'exécution intermittent *RESURRECT* décrit au chapitre 3 dans le cadre d'une application concrète. L'objectif de ce démonstrateur est partiellement atteint avec l'utilisation de bout en bout de *RESURRECT* sur une application simple, mais mettant en œuvre des fonctionnalités importante comme la gestion du parallélisme matériel/logiciel.

Une piste pour un futur développement serait d'optimiser l'acquisition d'échantillon le matin et le soir, lorsque les oiseaux sont le plus actifs. Ce démonstrateur ouvre de nouveaux axes de recherche pour les systèmes intermittents comme la gestion du temps combinée avec une approche consciente de l'énergie utilisée.

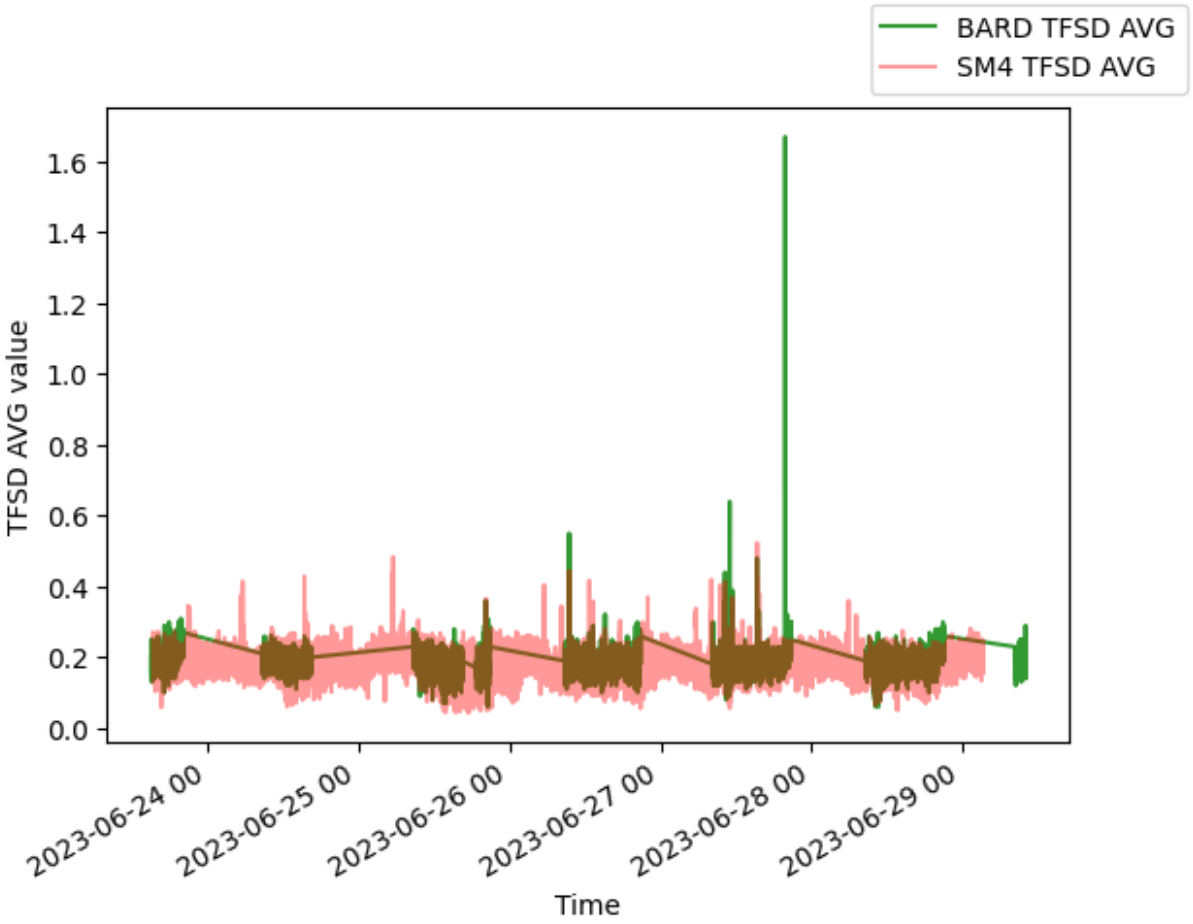


FIGURE 5.5 – Comparaison de la moyenne de l'indice acoustique TFSD obtenu avec un SongMeter et notre prototype de système intermittent.

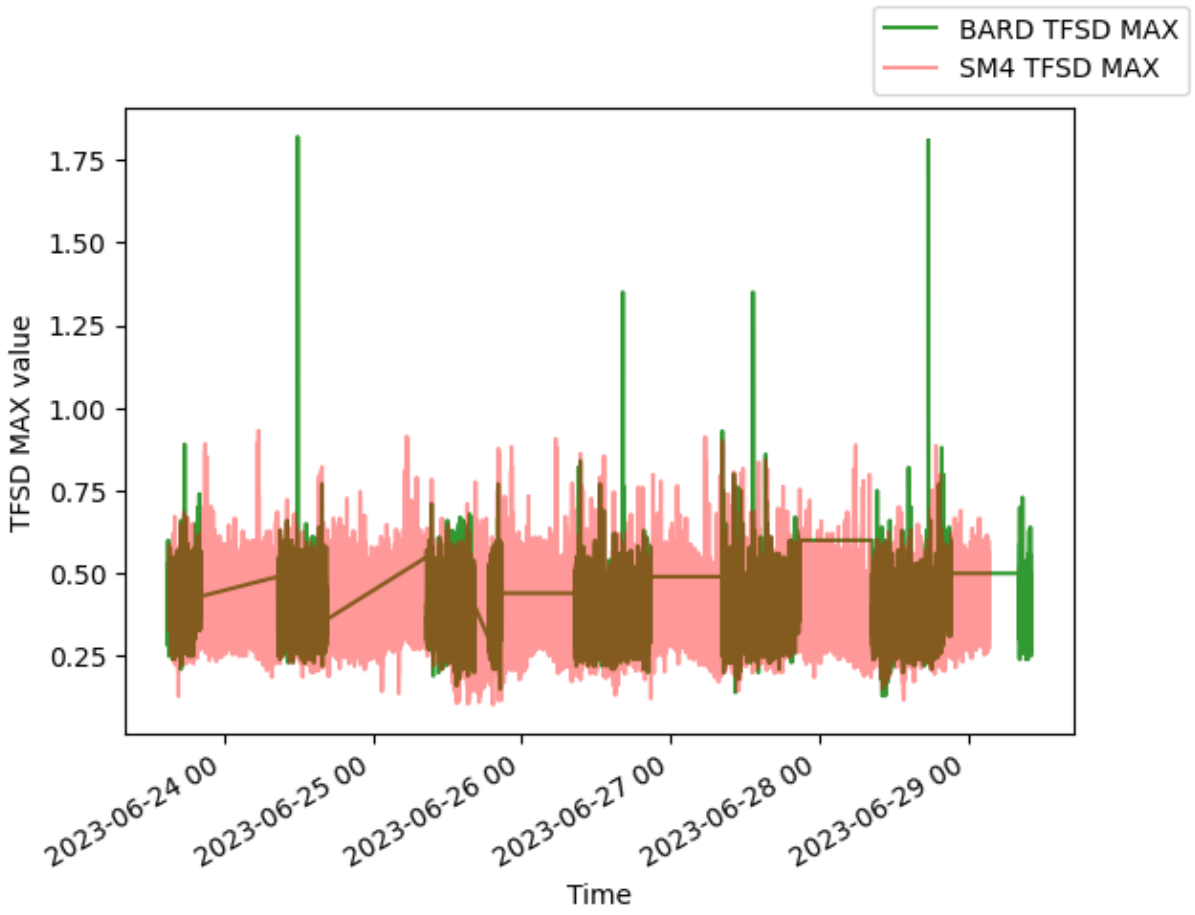


FIGURE 5.6 – Comparaison de la valeur maximale de l'indice acoustique TFSD obtenu avec un SongMeter et notre prototype de système intermittent.

Conclusion

Bilan

L'alimentation des systèmes informatique et plus particulièrement des objets connectés est un enjeu de taille. Pour limiter l'impact environnemental de ces objets et afin d'augmenter leur durée de vie, des modèles de calcul prenant en compte l'énergie sont apparus. La gestion de l'énergie apporte de nouvelles problématiques pour les systèmes informatiques. Dans ce contexte, la récolte d'énergie issue de l'environnement combinée avec le paradigme intermittent offre une alternative soutenable pour l'alimentation des nombreux objets connectés formant l'Internet des Objets. Les pertes d'alimentation ne sont alors plus des évènements anormaux rares mais font partie d'un fonctionnement normal. Ainsi une application s'exécute par intermittence. Même si ce paradigme ne convient pas à toutes les applications à cause de la variabilité de la qualité de service, ces techniques trouvent néanmoins leurs places.

Pour améliorer la prédictibilité de l'exécution d'une application sur un système intermittent, nous avons conçu un support d'exécution intermittent assurant l'aboutissement d'étapes de calcul sous contraintes d'énergie.

Pour cela, nous avons formalisé le problèmes de calcul de séquences optimales sous contraintes d'énergie en utilisant le formalisme des réseaux de Petri temporels à coût. Nous avons ainsi étendu les classes d'états afin d'intégrer la dimension du progrès de l'application. Un algorithme d'exploration exhaustif de l'espace des solutions est proposé pour la résolution de ce problème de maximisation de l'avancement de l'application sous contraintes d'énergie. Pour palier au problème d'explosion de l'espace des solutions, des algorithmes d'exploration partielle guidés par des heuristiques sont également proposés.

Une intégration de cet ordonnancement calculé hors-ligne est implémentée dans un support d'exécution intermittent basé sur un système d'exploitation temps-réel sous licence libre. Ce support d'exécution appelé *RESURRECT* montre l'intérêt d'adopter une vision consciente de l'énergie dans le développement des systèmes intermittents : cela permet de maximiser la quantité d'énergie utilisée pour faire progresser l'application en supprimant complètement les ré-exécutions de code et en minimisant le nombre de points

de sauvegarde.

Pour parfaire ces travaux, une étude de cas sur un système intermittent embarquant une application de surveillance de la biodiversité aviaire est étudiée. Des algorithmes pour la détection du chant des oiseaux dans le contexte intermittent sont présentés ainsi que les spécificités de leur implémentation sur des petits microcontrôleurs.

Enfin, un prototype de système intermittent dans le contexte de l'étude de cas étudiée vient démontrer la faisabilité d'un tel système. Ce prototype embarque le support d'exécution intermittent *RESURRECT* et valide l'utilisation de bout en bout des différents éléments proposés dans cette thèse.

Discussion des contributions

En ce qui concerne les modèles d'exécution intermittent, tous les modèles proposés jusque là ne prennent pas en compte le coût énergétique des portions de code exécutées pour les approches basées sur les *checkpoints* ou d'une tâche pour les approches dîtes *task-based*. Les travaux de cette thèse présentent *RESURRECT* un modèle d'exécution pour les systèmes intermittents qui considère ce coût énergétique afin de supprimer les surcoûts liés au rejeu de code et diminuer ceux dûs à la gestion de la mémoire. Contrairement aux supports d'exécution de l'état de l'art ayant des politiques *best-effort*, *RESURRECT* calcule hors ligne les différentes séquences d'activités optimales pour tous les états possibles de l'application et pour différents niveaux d'énergie disponibles. Ainsi, à l'exécution, *RESURRECT* choisit la meilleure séquence d'activités qu'il est possible d'exécuter de façon certaine avant la prochaine perte d'alimentation. Cependant, cela implique d'embarquer ces choix, utilisant ainsi une partie de la mémoire. Cela rend également le développement d'une application sous le paradigme intermittent dépendant des composants matériels utilisés comme le supercondensateur et les périphériques externes. Tout changement dans le matériel nécessite de repasser par l'étape de caractérisation de la plateforme, de modélisation, et de calcul hors-ligne.

Ce modèle d'exécution nécessite la mesure de la chute de tension pour les différents modes de fonctionnement ce qui peut représenter une contrainte et également une source d'erreur. De plus la granularité pour les modes de fonctionnement peut également être une source de variabilité. Dans notre modèle, nous avons défini un mode comme étant un état du système où les capacités de charge des transistors CMOS sont fixes. Dans les faits, cela peut représenter des temps très court. Dans ce cas, ils ne sont pas pris en compte dans le

modèle. Pour assurer un pessimisme nécessaire aux hypothèses du modèle de calcul, on peut surestimer les pentes de tension des modes définis.

L'algorithme de calcul des ordonnancements est présenté en utilisant le formalisme des réseaux de Petri temporels à coût. Le coût est ainsi utilisé pour décrire la consommation énergétique de bloc de code, et le progrès de l'application est abstrait à l'aide d'une variable incrémentée à chaque terminaison d'un de ces blocs d'instructions. La sémantique utilisée permet de décrire d'autres types de problèmes d'optimisation sous contrainte, à condition que la contrainte soit linéaire.

Perspectives

Les travaux réalisés dans cette thèse amènent de premiers résultats encourageants sur l'utilisation d'une approche consciente de l'énergie pour l'optimisation de système intermittent.

Réseaux de capteurs et mise à l'échelle. De nombreuses briques technologiques sont déjà utilisables pour mettre en œuvre des capteurs intermittents. Cependant, il pourrait être intéressant de faire communiquer ces capteurs entre eux afin d'obtenir une meilleure réactivité sur la détection d'évènements. La communication est l'opération la plus coûteuse en énergie, il faut donc développer des protocoles de communication pouvant assurer des transmission dans un contexte intermittent. Un second point intéressant à étudier serait la capacité de mettre à jour le logiciel s'exécutant sur ces capteurs intermittents. D'une part, cela permettrait de pouvoir corriger certaines failles matérielles ou logicielles découvertes après le déploiement et cela permettrait aussi de mutualiser les capteurs ou réseaux de capteurs de la même manière que les machines dans les centres de données. Un utilisateur pourrait envoyer son logiciel, et lorsqu'il le souhaite, passer la main à un autre utilisateur et ce sans la nécessité d'aller sur place physiquement.

Vieillissement du matériel. Les systèmes intermittents sont développés avec la volonté de prolonger le plus possible la durée de vie du capteur sans intervention humaine. Mais aujourd'hui, le vieillissement des composants n'est pas pris en compte dans les logiciels embarqués sur ces capteurs. Nous pensons qu'il est possible d'augmenter la qualité de service de ces capteurs en incorporant dans le support d'exécution une supervision du vieillissement de la plateforme. Dans le cas de *RESURRECT*, cela peut se traduire par enlever les *steps* demandant le plus d'énergie lorsque les performances de stockage d'énergie du supercondensateur diminuent.

Modèle de récolte d'énergie. La récolte d'énergie se fait lors du déploiement et à l'exécution, et il n'est pas possible d'anticiper fidèlement le comportement de cette récolte lors de l'étape de conception du système. Nous pensons tout de même qu'il est possible d'utiliser un modèle de récolte d'énergie lors de la conception du système. Cela pourrait se faire via une approche probabiliste qui pourrait être exploitée pour faire commuter l'application entre différents modes. En alliant cette modélisation probabiliste et des mesures de récolte lors de l'exécution, cela pourrait augmenter la qualité de service des systèmes intermittents. Pour aller plus loin, la conception pourrait également intégrer la dynamique des données captées du système.

Acronymes

Système

ADC Convertisseur analogique/numérique 64, 81, 93, 106, 107
CMOS Complementary Metal Oxide Semiconductor 26, 27, 29, 114
CPU Unité centrale de calcul 15, 16, 24, 26, 64, 73, 81, 105
DMA Accès directe à la mémoire (Direct Memory Access) 65, 106
DSP Processeur de signal numérique 24
FPU Unité de calcul en virgule flottante 95, 101
FRAM Ferroelectric random access memory 23, 82
IC Circuit intégré 24, 26, 29
LDO Régulateur à faible chute de tension 29–32
LED Diode électroluminescente 21
MCU Microcontrôleur 18, 24, 26, 29, 31, 33, 63, 70, 81, 82, 95, 105
MPPC Maximum Power Point Control 105
NVM Mémoire non volatile 23, 71, 84
RFID Radio Frequency Identification 15
SRAM Mémoire vive statique 23, 81, 82

Méthodes Formelles

cTPN Réseau de Petri temporel à coût 45–49, 52–54, 57, 60, 67
DSG Graphe d'état discret 44
PN Petri Net
PN Réseau de Petri 41–43, 85
SCG Graphe de classe 45
TPN Réseau de Petri temporisé 44, 45
TPN Réseau de Petri temporel 42–46, 52

Logiciel

API Interface de programmation d'application

BCET Meilleur temps d'exécution estimé 39

BOR Brow out 63

BSP Board Support Package 76

OS Système d'exploitation 76, 77

RTC Horloge temps réel 17, 70, 81

RTOS Système d'exploitation temps réel 76, 85

WAR Write-after-read 83

WCET Pire temps d'exécution estimé 39, 40, 107

BIBLIOGRAPHIE

[Aum+17] Pierre AUMOND et al., « Modeling Soundscape Pleasantness Using perceptual Assessments and Acoustic Measurements Along Paths in Urban Context », en, in : *Acta Acustica united with Acustica* 103.*3* (mai 2017), p. 430-443, ISSN : 1610-1928, DOI : 10.3813/AAA.919073, URL : http://www.ingentaconnect.com/content/10.3813/AAA.919073 (visité le 05/09/2022).

[Bak19] R. Jacob BAKER, *CMOS : Circuit Design, Layout, and Simulation*, Anglais, 4e édition, Piscataway, NJ : Wiley-IEEE Press, août 2019, ISBN : 978-1-119-48151-5.

[Bak+21] Abu BAKAR et al., « REHASH : A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing », en, in : *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5.*3* (sept. 2021), p. 1-42, ISSN : 2474-9567, DOI : 10.1145/3478077, URL : https://dl.acm.org/doi/10.1145/3478077 (visité le 22/08/2022).

[Bak+22] Abu BAKAR et al., « Protean : An Energy-Efficient and Heterogeneous Platform for Adaptive and Hardware-Accelerated Battery-Free Computing », en, in : *Proceedings of the Twentieth ACM Conference on Embedded Networked Sensor Systems*, Boston Massachusetts : ACM, nov. 2022, p. 207-221, ISBN : 978-1-4503-9886-2, DOI : 10.1145/3560905.3568561, URL : https://dl.acm.org/doi/10.1145/3560905.3568561 (visité le 20/02/2023).

[Bal+15] Domenico BALSAMO et al., « Hibernus : Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems », in : *IEEE Embedded Systems Letters* 7.*1* (mars 2015), Conference Name : IEEE Embedded Systems Letters, p. 15-18, ISSN : 1943-0671, DOI : 10.1109/LES.2014.2371494.

[Bal+16] Domenico BALSAMO et al., « Hibernus++ : A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices », en, in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.*12* (2016), p. 1968-1980, ISSN : 0278-0070, 1937-4151, DOI : 10.1109/TCAD.

2016.2547919, URL : http://ieeexplore.ieee.org/document/7442814/ (visité le 03/11/2021).

[BD91] B. Berthomieu et M. Diaz, « Modeling and verification of time dependent systems using time Petri nets », in : *IEEE Transactions on Software Engineering* 17.*3* (mars 1991), Conference Name : IEEE Transactions on Software Engineering, p. 259-273, ISSN : 1939-3520, DOI : 10.1109/32.75415.

[Bec+06] Jean-Luc Bechennec et al., « Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification », in : *2006 IEEE Conference on Emerging Technologies and Factory Automation*, ISSN : 1946-0759, sept. 2006, p. 62-69, DOI : 10.1109/ETFA.2006.355432.

[Ber+19] Gautier Berthou et al., « Sytare : A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems », in : *IEEE Transactions on Computers* 68.*9* (sept. 2019), Conference Name : IEEE Transactions on Computers, p. 1390-1403, ISSN : 1557-9956, DOI : 10.1109/TC.2018.2889080.

[Ber+20a] Gautier Berthou et al., « Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power embedded systems », en, in : *2020 Global Internet of Things Summit (GIoTS)*, Dublin, Ireland : IEEE, juin 2020, p. 1-6, ISBN : 978-1-72816-728-2, DOI : 10.1109/GIOTS49054.2020.9119593, URL : https://ieeexplore.ieee.org/document/9119593/ (visité le 04/05/2022).

[Ber+20b] Gautier Berthou et al., « Intermittent Computing with Peripherals, Formally Verified », en, in : ACM, juin 2020, p. 85, DOI : 10.1145/3372799.3394365, URL : https://hal.inria.fr/hal-02556878 (visité le 29/10/2021).

[Ber+21] Antoine Bernabeu et al., « Synthèse de traces temporisées à coût optimal pour l'ordonnancement de systèmes embarqués intermittents », fr, in : nov. 2021, URL : https://hal.science/hal-03449539 (visité le 24/08/2023).

[Ber+22] Antoine Bernabeu et al., « Cost-optimal timed trace synthesis for scheduling of intermittent embedded systems », in : *Discrete Event Dynamic Systems* (déc. 2022), ISSN : 1573-7594, DOI : 10.1007/s10626-022-00372-6, URL : https://doi.org/10.1007/s10626-022-00372-6.

[BM83] Bernard Berthomieu et Miguel Menasche, « An Enumerative Approach For Analyzing Time Petri Nets », in : *Proceedings IFIP*, Elsevier Science Publishers, 1983, p. 41-46.

[BMF14] Adrianna Bruni, Daniel J. Mennill et Jennifer R. Foote, « Dawn chorus start time variation in a temperate bird community : relationships with seasonality, weather, and ambient light », en, in : *Journal of Ornithology* 155.*4* (oct. 2014), p. 877-890, issn : 2193-7206, doi : 10.1007/s10336-014-1071-7, url : https://doi.org/10.1007/s10336-014-1071-7 (visité le 20/02/2023).

[Bot+04] H. Bottner et al., « New thermoelectric components using microsystem technologies », in : *Journal of Microelectromechanical Systems* 13.*3* (2004), p. 414-420, doi : 10.1109/JMEMS.2004.828740.

[Bou+17] Hanifa Boucheneb et al., « Optimal Reachability in Cost Time Petri Nets », en, in : *Formal Modeling and Analysis of Timed Systems*, sous la dir. d'Alessandro Abate et Gilles Geeraerts, t. 10419, Series Title : Lecture Notes in Computer Science, Cham : Springer International Publishing, 2017, p. 58-73, isbn : 978-3-319-65764-6 978-3-319-65765-3, doi : 10.1007/978-3-319-65765-3_4, url : http://link.springer.com/10.1007/978-3-319-65765-3_4 (visité le 13/12/2021).

[Bou+18] Jalil Boukhobza et al., « Emerging NVM : A Survey on Architectural Integration and Research Challenges », en, in : *ACM Transactions on Design Automation of Electronic Systems* 23.*2* (mars 2018), p. 1-32, issn : 1084-4309, 1557-7309, doi : 10.1145/3131848, url : https://dl.acm.org/doi/10.1145/3131848 (visité le 13/01/2023).

[Bro69] William Brown, « Experiments Involving a Microwave Beam to Power and Position a Helicopter », in : *IEEE Transactions on Aerospace and Electronic Systems* AES-5.*5* (sept. 1969), p. 692-702, issn : 0018-9251, doi : 10.1109/TAES.1969.309867, url : http://ieeexplore.ieee.org/document/4103386/ (visité le 16/10/2023).

[Cal+05] B.H. Calhoun et al., « Design considerations for ultra-low energy wireless microsensor nodes », in : *IEEE Transactions on Computers* 54.*6* (juin 2005), p. 727-740, issn : 1557-9956, doi : 10.1109/TC.2005.98.

[CL16] Alexei Colin et Brandon Lucia, « Chain : tasks and channels for reliable intermittent programs », in : *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, New York, NY, USA : Association for Computing Machinery, oct. 2016, p. 514-530, ISBN : 978-1-4503-4444-9, DOI : 10.1145/2983990.2983995, URL : https://dl.acm.org/doi/10.1145/2983990.2983995 (visité le 25/06/2023).

[CRL18] Alexei Colin, Emily Ruppel et Brandon Lucia, « A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices », in : *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, New York, NY, USA : Association for Computing Machinery, mars 2018, p. 767-781, ISBN : 978-1-4503-4911-6, DOI : 10.1145/3173162.3173210, URL : https://dl.acm.org/doi/10.1145/3173162.3173210 (visité le 17/10/2023).

[DAL18] Behnam Dezfouli, Immanuel Amirtharaj et Chia-Chi Li, « EMPIOT : An Energy Measurement Platform for Wireless IoT Devices », en, in : *arXiv :1804.04794 [cs]* (déc. 2018), arXiv : 1804.04794, URL : http://arxiv.org/abs/1804.04794 (visité le 04/05/2022).

[DFB22] Léna De Framond et Henrik Brumm, « Long-term effects of noise pollution on the avian dawn chorus : a natural experiment facilitated by the closure of an international airport », en, in : *Proceedings of the Royal Society B : Biological Sciences* 289.*1982* (sept. 2022), p. 20220906, ISSN : 0962-8452, 1471-2954, DOI : 10.1098/rspb.2022.0906, URL : https://royalsocietypublishing.org/doi/10.1098/rspb.2022.0906 (visité le 11/09/2023).

[Dya+13] Boris Dyatkin et al., « Development of a Green Supercapacitor Composed Entirely of Environmentally Friendly Materials », en, in : *ChemSusChem* 6.*12* (2013), _eprint : https ://onlinelibrary.wiley.com/doi/pdf/10.1002/cssc.201300852, p. 2269-2280, ISSN : 1864-564X, DOI : 10.1002/cssc.201300852, URL : https://onlinelibrary.wiley.com/doi/abs/10.1002/cssc.201300852 (visité le 14/03/2023).

[Fox+04] Glen R. Fox et al., « The Current Status of FeRAM », en, in : *Ferroelectric Random Access Memories : Fundamentals and Applications*, sous la dir. d'Hiroshi Ishiwara, Masanori Okuyama et Yoshihiro Arimoto, Topics in

Applied Physics, Berlin, Heidelberg : Springer, 2004, p. 139-148, ISBN : 978-3-540-45163-1, DOI : 10.1007/978-3-540-45163-1_10, URL : https://doi.org/10.1007/978-3-540-45163-1_10 (visité le 03/04/2023).

[GLB19] Graham GOBIESKI, Brandon LUCIA et Nathan BECKMANN, « Intelligence Beyond the Edge : Inference on Intermittent Embedded Systems », en, in : *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence RI USA : ACM, avr. 2019, p. 199-213, ISBN : 978-1-4503-6240-5, DOI : 10.1145/3297858.3304011, URL : https://dl.acm.org/doi/10.1145/3297858.3304011 (visité le 01/12/2021).

[Hab+17] Mohamadhadi HABIBZADEH et al., « Hybrid Solar-Wind Energy Harvesting for Embedded Applications : Supercapacitor-Based System Architectures and Design Tradeoffs », in : *IEEE Circuits and Systems Magazine* 17.*4* (2017), Conference Name : IEEE Circuits and Systems Magazine, p. 29-63, ISSN : 1558-0830, DOI : 10.1109/MCAS.2017.2757081.

[HCL93] Jayant R. HARITSA, Michael J. CANREY et Miron LIVNY, « Value-based scheduling in real-time database systems », en, in : *The VLDB Journal* 2.*2* (avr. 1993), p. 117-152, ISSN : 0949-877X, DOI : 10.1007/BF01232184, URL : https://doi.org/10.1007/BF01232184 (visité le 21/08/2023).

[Hes+16] Josiah HESTER et al., « Persistent Clocks for Batteryless Sensing Devices », en, in : *ACM Transactions on Embedded Computing Systems* 15.*4* (sept. 2016), p. 1-28, ISSN : 1539-9087, 1558-3465, DOI : 10.1145/2903140, URL : https://dl.acm.org/doi/10.1145/2903140 (visité le 28/09/2023).

[Hil+18] Andrew P. HILL et al., « AudioMoth : Evaluation of a smart open acoustic device for monitoring biodiversity and the environment », en, in : *Methods in Ecology and Evolution* 9.*5* (mai 2018), sous la dir. de Nick ISAAC, p. 1199-1211, ISSN : 2041-210X, 2041-210X, DOI : 10.1111/2041-210X.12955, URL : https://onlinelibrary.wiley.com/doi/10.1111/2041-210X.12955 (visité le 24/05/2022).

[HNR68] Peter E. HART, Nils J. NILSSON et Bertram RAPHAEL, « A Formal Basis for the Heuristic Determination of Minimum Cost Paths », in : *IEEE Transactions on Systems Science and Cybernetics* 4.*2* (juill. 1968), Conference

Name : IEEE Transactions on Systems Science and Cybernetics, p. 100-107, ISSN : 2168-2887, DOI : 10.1109/TSSC.1968.300136.

[HS17] Josiah HESTER et Jacob SORBER, « Flicker : Rapid Prototyping for the Batteryless Internet-of-Things », in : *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA : Association for Computing Machinery, nov. 2017, p. 1-13, ISBN : 978-1-4503-5459-2, DOI : 10.1145/3131672.3131674, URL : https://doi.org/10.1145/3131672.3131674 (visité le 10/01/2023).

[HSS17] Josiah HESTER, Kevin STORER et Jacob SORBER, « Timely Execution on Intermittently Powered Batteryless Sensors », in : *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA : Association for Computing Machinery, nov. 2017, p. 1-13, ISBN : 978-1-4503-5459-2, DOI : 10.1145/3131672.3131673, URL : https://dl.acm.org/doi/10.1145/3131672.3131673 (visité le 25/06/2023).

[Ish12] Hiroshi ISHIWARA, « Ferroelectric Random Access Memories », in : *Journal of Nanoscience and Nanotechnology* 12.*10* (oct. 2012), p. 7619-7627, DOI : 10.1166/jnn.2012.6651.

[Jai+14] Navendu JAIN et al., « A Truthful Mechanism for Value-Based Scheduling in Cloud Computing », en, in : *Theory of Computing Systems* 54.*3* (avr. 2014), p. 388-406, ISSN : 1433-0490, DOI : 10.1007/s00224-013-9449-0, URL : https://doi.org/10.1007/s00224-013-9449-0 (visité le 21/08/2023).

[Jay+15] Hrishikesh JAYAKUMAR et al., « QuickRecall : A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers », in : *ACM Journal on Emerging Technologies in Computing Systems* 12.*1* (août 2015), 8 :1-8 :19, ISSN : 1550-4832, DOI : 10.1145/2700249, URL : https://dl.acm.org/doi/10.1145/2700249 (visité le 28/09/2023).

[Kim+14] Sangkil KIM et al., « Ambient RF energy-harvesting technologies for self-sustainable standalone wireless sensor platforms », in : *Proceedings of the IEEE* 102.*11* (nov. 2014), p. 1649-1666, ISSN : 1558-2256, DOI : 10.1109/JPROC.2014.2357031.

[KK15] Piotr KOCANDA et Andrzej KOS, « Static and dynamic energy losses vs. temperature in different CMOS technologies », in : *2015 22nd International Conference Mixed Design of Integrated Circuits & Systems (MIXDES)*, juin 2015, p. 446-449, DOI : 10.1109/MIXDES.2015.7208560.

[Kor+20] Vito KORTBEEK et al., « Time-sensitive Intermittent Computing Meets Legacy Software », in : *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, New York, NY, USA : Association for Computing Machinery, mars 2020, p. 85-99, ISBN : 978-1-4503-7102-5, DOI : 10.1145/3373376.3378476, URL : https://doi.org/10.1145/3373376.3378476 (visité le 02/12/2022).

[Kwo17] Roberta KWOK, « Field Instruments : Build it yourself », en, in : *Nature* 545.*7653* (mai 2017), Number : 7653 Publisher : Nature Publishing Group, p. 253-255, ISSN : 1476-4687, DOI : 10.1038/nj7653-253a, URL : https://www.nature.com/articles/nj7653-253a (visité le 28/02/2023).

[LA19] Xilong LIU et Nirwan ANSARI, « Toward Green IoT : Energy Solutions and Key Challenges », in : *IEEE Communications Magazine* 57.*3* (mars 2019), Conference Name : IEEE Communications Magazine, p. 104-110, ISSN : 1558-1896, DOI : 10.1109/MCOM.2019.1800175.

[Lib] « librosa/librosa : 0.10.1 », en, in : (), DOI : 10.5281/zenodo.8252662, URL : https://zenodo.org/records/8252662 (visité le 08/11/2023).

[Los+19] Vincent LOSTANLEN et al., « Per-Channel Energy Normalization : Why and How », en, in : *IEEE Signal Processing Letters* 26.*1* (jan. 2019), p. 39-43, ISSN : 1070-9908, 1558-2361, DOI : 10.1109/LSP.2018.2878620, URL : https://ieeexplore.ieee.org/document/8514023/ (visité le 12/11/2021).

[Los+21] Vincent LOSTANLEN et al., « Energy Efficiency is Not Enough :Towards a Batteryless Internet of Sounds », in : *Proceedings of the 16th International Audio Mostly Conference*, AM '21, New York, NY, USA : Association for Computing Machinery, oct. 2021, p. 147-155, ISBN : 978-1-4503-8569-5, DOI : 10.1145/3478384.3478408, URL : https://doi.org/10.1145/3478384.3478408 (visité le 16/04/2023).

[LR15] Brandon Lucia et Benjamin Ransford, « A simpler, safer programming and execution model for intermittent systems », en, in : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland OR USA : ACM, juin 2015, p. 575-585, isbn : 978-1-4503-3468-6, doi : 10.1145/2737924.2737978, url : https://dl.acm.org/doi/10.1145/2737924.2737978 (visité le 01/12/2021).

[LRS21] Didier Lime, Olivier H. Roux et Charlotte Seidner, *Cost Problems for Parametric Time Petri Nets*, en, arXiv :2109.03658 [cs], déc. 2021, url : http://arxiv.org/abs/2109.03658 (visité le 05/09/2023).

[MA07] C. Maxey et A. Andreas, *Oak Ridge National Laboratory (ORNL) ; Rotating Shadowband Radiometer (RSR) ; Oak Ridge, Tennessee (Data)*, en, 2007, doi : 10.5439/1052553, url : http://www.osti.gov/servlets/purl/1052553/ (visité le 02/10/2023).

[Mai+22] Pol Maistriaux et al., « Modeling the Carbon Footprint of Battery-Powered IoT Sensor Nodes for Environmental-Monitoring Applications », en, in : *Proceedings of the 12th International Conference on the Internet of Things*, Delft Netherlands : ACM, nov. 2022, p. 9-16, isbn : 978-1-4503-9665-3, doi : 10.1145/3567445.3567448, url : https://dl.acm.org/doi/10.1145/3567445.3567448 (visité le 14/04/2023).

[MCL17] Kiwan Maeng, Alexei Colin et Brandon Lucia, « Alpaca : intermittent execution without checkpoints », en, in : *Proceedings of the ACM on Programming Languages* 1.*OOPSLA* (oct. 2017), p. 1-30, issn : 2475-1421, doi : 10.1145/3133920, url : https://dl.acm.org/doi/10.1145/3133920 (visité le 01/12/2021).

[Mee+14] Jagan Singh Meena et al., « Overview of emerging nonvolatile memory technologies », en, in : *Nanoscale Research Letters* 9.*1* (sept. 2014), p. 526, issn : 1556-276X, doi : 10.1186/1556-276X-9-526, url : https://doi.org/10.1186/1556-276X-9-526 (visité le 13/01/2023).

[Mer74] Philip Meir Merlin, « A study of the recoverability of computing systems. », AAI7511026, phd, University of California, Irvine, 1974.

[Mil+16] V. A. Milichko et al., « Solar photovoltaics : current state and trends », en, in : *Physics-Uspekhi* 59.*8* (août 2016), Publisher : IOP Publishing, p. 727, ISSN : 1063-7869, DOI : 10.3367/UFNe.2016.02.037703, URL : https://iopscience.iop.org/article/10.3367/UFNe.2016.02.037703/meta (visité le 04/09/2023).

[ML18] Kiwan Maeng et Brandon Lucia, « Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing », en, in : 2018, p. 129-144, ISBN : 978-1-939133-08-3, URL : https://www.usenix.org/conference/osdi18/presentation/maeng (visité le 28/11/2022).

[ML20] Kiwan Maeng et Brandon Lucia, « Adaptive low-overhead scheduling for periodic and reactive intermittent execution », en, in : *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London UK : ACM, juin 2020, p. 1005-1021, ISBN : 978-1-4503-7613-6, DOI : 10.1145/3385412.3385998, URL : https://dl.acm.org/doi/10.1145/3385412.3385998 (visité le 01/12/2021).

[Msp] *MSP430FR5994 datasheet | TI.com*, URL : https://www.ti.com/document-viewer/msp430fr5994/datasheet (visité le 07/11/2023).

[Nar+19] Matteo Nardello et al., « Camaroptera : a Batteryless Long-Range Remote Visual Sensing System », en, in : *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems - ENSsys'19*, New York, NY, USA : ACM Press, 2019, p. 8-14, ISBN : 978-1-4503-7010-3, DOI : 10.1145/3362053.3363491, URL : http://dl.acm.org/citation.cfm?doid=3362053.3363491 (visité le 01/12/2021).

[PB00] D. Prasad et A. Burns, « A value-based scheduling approach for real-time autonomous vehicle control », en, in : *Robotica* 18.*3* (mai 2000), Publisher : Cambridge University Press, p. 273-279, ISSN : 1469-8668, 0263-5747, DOI : 10.1017/S0263574799002349, URL : https://www.cambridge.org/core/journals/robotica/article/valuebased-scheduling-approach-for-realtime-autonomous-vehicle-control/9574539C92423436D88AFCF538705278 (visité le 21/08/2023).

[Pra+18] Michal Prauzek et al., « Energy Harvesting Sources, Storage Devices and System Topologies for Environmental Wireless Sensor Networks : A Review », en, in : *Sensors* 18.*8* (août 2018), Number : 8 Publisher : Multi-

disciplinary Digital Publishing Institute, p. 2446, ISSN : 1424-8220, DOI : 10.3390/s18082446, URL : https://www.mdpi.com/1424-8220/18/8/2446 (visité le 07/08/2023).

[Raz+18] Waseem RAZA et al., « Recent advancements in supercapacitor technology », in : *Nano Energy* 52 (oct. 2018), p. 441-473, ISSN : 2211-2855, DOI : 10.1016/j.nanoen.2018.08.013, URL : https://www.sciencedirect.com/science/article/pii/S2211285518305755 (visité le 18/10/2023).

[RL14] Benjamin RANSFORD et Brandon LUCIA, « Nonvolatile memory is a broken time machine », en, in : *Proceedings of the workshop on Memory Systems Performance and Correctness*, Edinburgh United Kingdom : ACM, juin 2014, p. 1-3, ISBN : 978-1-4503-2917-0, DOI : 10.1145/2618128.2618136, URL : https://dl.acm.org/doi/10.1145/2618128.2618136 (visité le 31/10/2023).

[RLS06] J. I. RASMUSSEN, K. G. LARSEN et K. SUBRAMANI, « On using priced timed automata to achieve optimal scheduling », en, in : *Formal Methods in System Design* 29.*1* (juill. 2006), p. 97-114, ISSN : 1572-8102, DOI : 10.1007/s10703-006-0014-1, URL : https://doi.org/10.1007/s10703-006-0014-1 (visité le 05/09/2023).

[RSF11] Benjamin RANSFORD, Jacob SORBER et Kevin FU, « Mementos : system support for long-running computation on RFID-scale devices », in : *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, New York, NY, USA : Association for Computing Machinery, mars 2011, p. 159-170, ISBN : 978-1-4503-0266-1, DOI : 10.1145/1950365.1950386, URL : https://dl.acm.org/doi/10.1145/1950365.1950386 (visité le 27/09/2023).

[Sam+08] Alanson P. SAMPLE et al., « Design of an RFID-Based Battery-Free Programmable Sensing Platform », in : *IEEE Transactions on Instrumentation and Measurement* 57.*11* (nov. 2008), Conference Name : IEEE Transactions on Instrumentation and Measurement, p. 2608-2615, ISSN : 1557-9662, DOI : 10.1109/TIM.2008.925019.

[SK11] Sujesha SUDEVALAYAM et Purushottam KULKARNI, « Energy Harvesting Sensor Nodes : Survey and Implications », in : *IEEE Communications Surveys & Tutorials* 13.*3* (2011), Conference Name : IEEE Communications Surveys

& Tutorials, p. 443-461, ISSN : 1553-877X, DOI : 10.1109/SURV.2011.060710.00094.

[Sli+20a] Sivert T. SLIPER et al., « Energy-driven computing », en, in : *Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences* 378.*2164* (fév. 2020), p. 20190158, ISSN : 1364-503X, 1471-2962, DOI : 10.1098/rsta.2019.0158, URL : https://royalsocietypublishing.org/doi/10.1098/rsta.2019.0158 (visité le 18/11/2021).

[Sli+20b] Sivert T. SLIPER et al., « Fused : Closed-Loop Performance and Energy Simulation of Embedded Systems », in : *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, août 2020, p. 263-272, DOI : 10.1109/ISPASS48437.2020.00046.

[SLJ20] Milijana SURBATOVICH, Brandon LUCIA et Limin JIA, « Towards a formal foundation of intermittent computing », en, in : *Proceedings of the ACM on Programming Languages* 4.*OOPSLA* (nov. 2020), p. 1-31, ISSN : 2475-1421, DOI : 10.1145/3428231, URL : https://dl.acm.org/doi/10.1145/3428231 (visité le 01/12/2021).

[Sue+14] Jérôme SUEUR et al., « Acoustic Indices for Biodiversity Assessment and Landscape Investigation », en, in : *Acta Acustica united with Acustica* 100.*4* (juill. 2014), p. 772-781, ISSN : 16101928, DOI : 10.3813/AAA.918757, URL : http://openurl.ingenta.com/content/xref?genre=article&issn=1610-1928&volume=100&issue=4&spage=772 (visité le 28/02/2023).

[Sug+19] Larissa Sayuri Moreira SUGAI et al., « Terrestrial Passive Acoustic Monitoring : Review and Perspectives », en, in : *BioScience* 69.*1* (jan. 2019), p. 15-25, ISSN : 0006-3568, 1525-3244, DOI : 10.1093/biosci/biy147, URL : https://academic.oup.com/bioscience/article/69/1/15/5193506 (visité le 28/02/2023).

[TR21] Matthew TOENIES et Lindsey RICH, « Advancing bird survey efforts through novel recorder technology and automated species identification », en, in : *California Fish and Wildlife Journal* 107.*2* (août 2021), p. 56-70, ISSN : 2689-4203, 2689-419X, DOI : 10.51492/cfwj.107.5, URL : https://nrm.dfg.ca.gov/FileHandler.ashx?DocumentID=193712&inline (visité le 28/02/2023).

[Wäg+18] Peter Wägemann et al., « Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems », en, in : (2018), p. 25.

[Wan+16] Yuxuan Wang et al., « Trainable Frontend For Robust and Far-Field Keyword Spotting », en, in : *arXiv :1607.05666 [cs]* (juill. 2016), arXiv : 1607.05666, URL : http://arxiv.org/abs/1607.05666 (visité le 18/03/2022).

[WH16] Joel Van Der Woude et Matthew Hicks, « Intermittent Computation without Hardware Support or Programmer Intervention », en, in : 2016, p. 17-32, ISBN : 978-1-931971-33-1, URL : https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude (visité le 28/11/2022).

[Wil+08] Reinhard Wilhelm et al., « The worst-case execution-time problem—overview of methods and survey of tools », in : *ACM Transactions on Embedded Computing Systems* 7.*3* (mai 2008), 36 :1-36 :53, ISSN : 1539-9087, DOI : 10.1145/1347375.1347389, URL : https://dl.acm.org/doi/10.1145/1347375.1347389 (visité le 06/11/2023).

[Win+20] Jasper de Winkel et al., « Battery-Free Game Boy », en, in : *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4.*3* (sept. 2020), p. 1-34, ISSN : 2474-9567, DOI : 10.1145/3411839, URL : https://dl.acm.org/doi/10.1145/3411839 (visité le 01/12/2021).

[WJ17] Chongfeng Wei et Xingjian Jing, « A comprehensive review on vibration energy harvesting : Modelling and realization », in : *Renewable and Sustainable Energy Reviews* 74 (juill. 2017), p. 1-18, ISSN : 1364-0321, DOI : 10.1016/j.rser.2017.01.073, URL : https://www.sciencedirect.com/science/article/pii/S1364032117300837 (visité le 16/10/2023).

[YCY] Eren Yildiz, Lijun Chen et Kasım Sinan Yildirim, « Immortal Threads : Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers », en, in : (), p. 18.

[Yil09] Faruk Yildiz, « Potential Ambient Energy-Harvesting Sources and Techniques », in : *The Journal of Technology Studies* 35.*1* (2009), ISSN : 1541-9258.

[Yun+22] Jihoon Yun et al., « Infrastructure-free, Deep Learned Urban Noise Monitoring at $\sim$100mW », in : *arXiv :2203.06220 [cs, eess]* (mars 2022),

arXiv : 2203.06220, URL : http://arxiv.org/abs/2203.06220 (visité le 21/03/2022).

[Yı+18] Kasım Sinan YILDIRIM et al., « InK : Reactive Kernel for Tiny Batteryless Sensors », en, in : *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, Shenzhen China : ACM, nov. 2018, p. 41-53, ISBN : 978-1-4503-5952-8, DOI : 10.1145/3274783.3274837, URL : https://dl.acm.org/doi/10.1145/3274783.3274837 (visité le 06/10/2022).

PUBLICATIONS

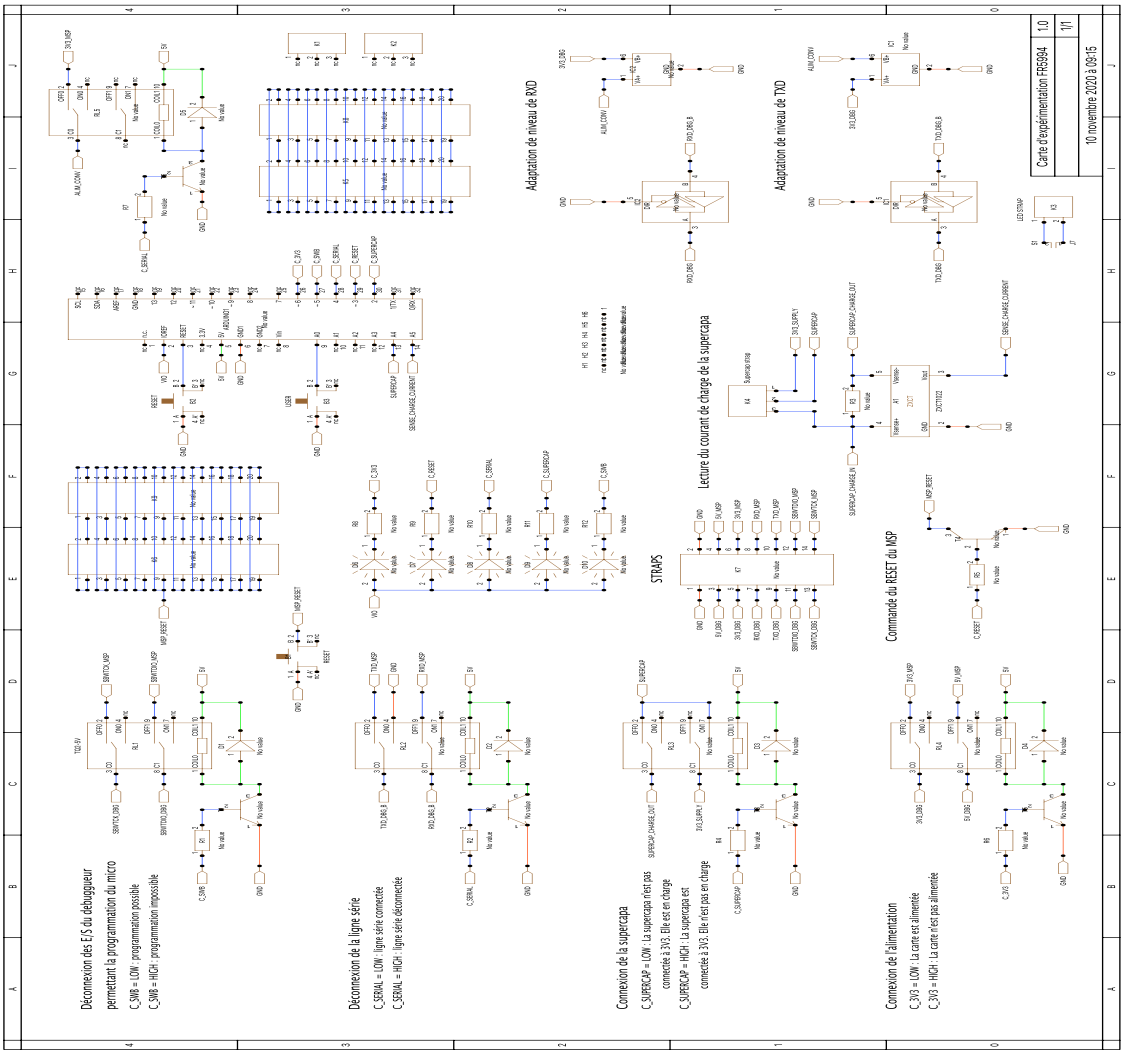
Les travaux présentés dans ce manuscrit ont fait l'objet des publications suivantes :

[**Ber+22**] Antoine BERNABEU et al., « Cost-optimal timed trace synthesis for scheduling of intermittent embedded systems », in : *Discrete Event Dynamic Systems* (déc. 2022), ISSN : 1573-7594, DOI : 10.1007/s10626-022-00372-6, URL : https://doi.org/10.1007/s10626-022-00372-6

[**Ber+21**] Antoine BERNABEU et al., « Synthèse de traces temporisées à coût optimal pour l'ordonnancement de systèmes embarqués intermittents », fr, in : nov. 2021, URL : https://hal.science/hal-03449539 (visité le 24/08/2023)

[**Los+21**] Vincent LOSTANLEN et al., « Energy Efficiency is Not Enough :Towards a Batteryless Internet of Sounds », in : *Proceedings of the 16th International Audio Mostly Conference*, AM '21, New York, NY, USA : Association for Computing Machinery, oct. 2021, p. 147-155, ISBN : 978-1-4503-8569-5, DOI : 10.1145/3478384.3478408, URL : https://doi.org/10.1145/3478384.3478408 (visité le 16/04/2023)

SCHÉMAS DU CIRCUIT AD-HOC



COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

CENTRALE NANTES

Titre : Support d'exécution pour les systèmes intermittents

Mot clés : Intermittent, sans-batteries, support d'exécution, récolte d'énergie, modélisation hors-ligne

Résumé : Les travaux de cette thèse s'intéressent à la modélisation et le développement de mécanismes permettant une exécution efficace sur des systèmes alimentés par une énergie intermittente tels que les capteurs sans fil. De tels systèmes sont alimentés par énergie renouvelable via un petit tampon d'énergie sous la forme d'un supercondensateur. La contribution principale est l'exploration d'une approche consciente de l'énergie pour les supports d'exécution intermittents. Tout d'abord, nous proposons d'utiliser les réseaux de Petri temporels à coût afin de modéliser de tels systèmes et nous proposons une extension de cette sémantique afin de résoudre un problème d'ordonnancement maximisant une variable sous contrainte de coût. En combinant le modèle proposé et une approche pire cas, nous générons un ordonnancement fiable pour les systèmes intermittents qui ne commence aucunes opérations sans la certitude de les finir vis-à-vis de l'énergie disponible. Nous avons ensuite utilisé cet ordonnancement dans un support d'exécution basé sur un système d'exploitation temps-réel pour gérer l'intermittence. Ce support d'exécution utilise un modèle de consommation de l'énergie du système intermittent afin de minimiser les opérations liées à l'exécution intermittente et d'assurer la continuité des opérations de l'application malgré les interruptions fréquentes d'alimentation. Ces travaux ont été jusqu'à la mise en œuvre pratique d'un tel support d'exécution sur une étude de cas concernant la détection de chant des oiseaux.

Title: Runtime support for intermittent computing

Keywords: Intermittent, batteryless, runtime, energy harvesting, offline modeling

Abstract: The work in this thesis focuses on the modeling and development of mechanisms to enable efficient execution on systems powered by intermittent energy such as wireless sensors. Such systems are powered by renewable energy via a small energy buffer in the form of a supercapacitor. The main contribution is the exploration of an energy-aware approach for intermittent runtimes. First, we propose to use cost time Petri nets to model such systems and we propose an extension of this semantics to solve a scheduling problem maximizing a variable under cost constraint. By combining the proposed model and a worst-case approach, we generate a reliable scheduling for intermittent systems that does not start any operation without the certainty of finishing it with respect to the available energy. We then used this scheduling in an execution support based on a real-time operating system to manage intermittency. This runtime support uses a model of the intermittent system's energy consumption to minimize the operations associated with intermittent execution and to ensure the continuity of the application's operations despite frequent power interruptions. This work has gone as far as the practical implementation of such execution support on a case study concerning the detection of birdsong.