Modèle d'exécution conscient de l'énergie pour les systèmes intermittents

Hugo Reymond

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE RENNES

Par
Hugo REYMOND

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Thèse présentée et soutenue à Rennes, le 8 novembre 2024
Unité de recherche : Centre Inria de l'université de Rennes

Modèle d'exécution conscient de l'énergie pour les systèmes intermittents

Rapporteurs avant soutenance :

Kevin MARTIN	Professeur des Universités, Université de Bretagne Sud
Tanguy RISSET	Professeur des Universités, INSA Lyon

Composition du Jury :

Président :	Steven DERRIEN	Professeur des Universités, Université de Bretage Occidentale
Examinateurs :	Thomas CARLE	Maître de conférences, Université de Toulouse 3
	Kevin MARTIN	Professeur des Universités, Université de Bretagne Sud
	Tanguy RISSET	Professeur des Universités, INSA Lyon
Dir. de thèse :	Isabelle PUAUT	Professeur des Universités, Université de Rennes
Co-dir. de thèse :	Erven ROHOU	Directeur de Recherche, INRIA Rennes
Co-enc. de thèse :	Sébastien FAUCOU	Maître de conférences, Nantes Université

Invités :

Co-enc. de thèse : Jean-Luc BÉCHENNEC	Chargé de Recherche, CNRS
Co-enc. de thèse : Mikaël BRIDAY	Maitre de conférences, Centrale Nantes

Remerciements

En préambule de ce manuscrit, je tiens à remercier toutes les personnes qui m'ont aidées tout au long de cette thèse.

Ce travail de thèse n'aurait pas pu voir le jour sans mes encadrants de thèse Isabelle Puaut, Erven Rohou, Sébastien Faucou, Jean-Luc Béchennec et Mikaël Briday. C'est grâce à leur expertise, leur soutien et leur bienveillance que j'ai pu m'épanouir durant ces trois passionnantes années. Un immense MERCI pour votre confiance et votre accompagnement dans cette merveilleuse aventure.

Je tiens également à exprimer ma gratitude aux membres du jury Kevin Martin, Tanguy Risset, Thomas Carle, Steven Derrien qui m'ont fait l'honneur d'évaluer mon travail.

Je souhaite aussi remercier chaleureusement Guillaume Salagnac, qui m'a fait découvrir le monde de la recherche et de l'enseignement, et m'a encouragé à poursuivre une thèse.

Un grand merci aux membres de l'équipe PACAP : André, Antoine, Aurore, Camille, Caroline, Damien, Erven, Hector, Imane, Matthieu, Nassim, Nicolas Ba., Nicolas Be., Pierre B., Pierre M., Sara, Thomas, Virginie, Xabi, pour les moments de partage, que ce soit autour d'un café ou d'une bière, et leur bonne humeur quotidienne.

Merci aux personnes formidables que j'ai rencontrées à Rennes : les amis de l'équipe PACAP, Jade, Rémi, Tom, Emily, Amélie, Opale. Toutes nos soirées piscine, Sumup, jeux de société et sorties vélo, qui ne sont finalement qu'une excuse parfaite pour se retrouver autour d'un bon repas, resteront gravées dans ma mémoire.

Je souhaite remercier mes amis de toujours, qui m'ont soutenu tout au long de ma thèse malgré la distance : Laurine, Théo, Loïc, Alban, Guillaume, Matthieu, Valentine, Xoxo et tous ceux qui m'ont accompagné durant ces années.

Enfin, je souhaite exprimer toute ma reconnaissance à ma famille pour leurs relectures de cette thèse, leur soutien inconditionnel et leur amour.

Pour conclure, un merci tout particulier à ma sœur Claire, qui me rend tous les jours plus fier d'elle, pour tous les merveilleux moments que nous avons passés, et pour ceux à venir.

Sommaire

Glossaire

Dans ce manuscrit, nous avons fait le choix de traduire les termes techniques en français, mais de garder leurs acronymes en anglais pour des raisons de clarté. Cette liste récapitule les acronymes utilisés dans le manuscrit, ainsi que leur définition.

Acronyme		
ADC	Analog to digital converter	Convertisseur analogique numérique
CFG	Control flow graph	Graphe de flot de contrôle
ILP	Integer linear programming	Programmation linéaire en nombres entiers
IPET	Implicit path enumeration technique	Énumération implicite des chemins
IR	Intermediate representation	Représentation intermédiaire
ML	Machine learning	Apprentissage automatique
NVM	Non-volatile memory	Mémoire non volatile
RCG	Reachable checkpoint graph	Graphe des points de sauvegarde atteignables
TBPF	Time between power failures	Intervalle entre les pertes d'alimentation
VM	Volatile memory	Mémoire volatile
WAR	Write After Read	Écriture après lecture
WCEC	Worst case energy consumption	Consommation d'énergie pire cas
WCET	Worst case execution time	Temps d'exécution pire cas

Tableau 1 – Liste des acronymes utilisés

Introduction

Contexte du travail : capteurs sans batterie

Depuis un certain nombre d'années, les capteurs sans fil sont de plus en plus présents dans notre paysage : thermostats connectés, capteurs de pression de pneus, capteurs de taux de CO_2. D'après une étude prospective menée par Arm, la quantité d'objets connectés déployés augmentera continuellement dans les prochaines décennies, avec plus d'un trilliard en circulation à l'horizon 2050 [Spa17]. Face à cette augmentation conséquente se pose la question de l'autonomie énergétique : Comment alimenter en énergie des capteurs de plus en plus nombreux et de plus en plus complexes ?

Historiquement, les capteurs sans fil étaient alimentés à l'aide de piles ou de batteries. Cependant, ces technologies présentent des limites en termes de durée de vie et nécessitent des interventions de maintenance fréquentes pour remplacer ces piles ou batteries. Lorsque ces capteurs sont déployés dans des environnements isolés ou difficiles d'accès comme des forêts, des rivières ou des montagnes, ces remplacements ont un impact écologique considérable et engendrent des coûts de maintenance élevés. De plus, dans certains cas, ce remplacement est impossible, par exemple lorsque le capteur est intégré dans des fondations pour surveiller leur vieillissement, ou encore quand le capteur est envoyé dans l'espace.

Face à ces problématiques, les capteurs *sans batterie* offrent une alternative aux systèmes classiques et s'affranchissent des batteries en tirant parti de l'énergie disponible dans l'environnement. Seulement, en l'absence d'une batterie pour agir comme réserve d'énergie, ces capteurs sont contraints par la quantité limitée d'énergie disponible dans l'environnement. Ainsi, même si ces capteurs sont équipés de condensateurs qui agissent comme des petites réserves d'énergie, ils subissent des pertes d'alimentations fréquentes. Ces interruptions changent le modèle d'exécution de ces capteurs : on ne parle alors plus d'exécution *continue*, mais *intermittente*, où l'exécution d'un programme est découpée en plusieurs phases, séparées par des pertes d'alimentation.

Ce nouveau paradigme comporte plusieurs défis, le plus notable étant celui de la *progression* du programme : comment s'assurer de la progression d'un programme alors

que des pertes d'alimentation régulières effacent le contenu de la mémoire volatile et des registres du processeur ?

Plusieurs solutions ont été proposées dans la littérature, aussi bien matérielles que logicielles. L'une d'elle consiste à instrumenter le programme avec des *points de sauvegarde* (*checkpoint* en anglais) placés dans le binaire du programme. Lorsque le programme rencontre un point de sauvegarde, l'état volatil du programme (la mémoire volatile et les registres du CPU) est copié dans une mémoire non volatile (*Non-Volatile Memory* en anglais, abrégé NVM). Ainsi, si le capteur subit une perte d'alimentation, il attendra simplement d'être suffisamment alimenté avant de reprendre l'exécution du programme depuis la dernière sauvegarde.

Problématiques de la thèse

Dans cette thèse, nous nous donnons l'objectif d'améliorer les performances de programmes exécutés de manière intermittente, en fournissant des mécanismes de sauvegarde de la progression sûrs et efficaces. Pour cela, nous soutenons qu'une approche *consciente de l'énergie* est essentielle : sans la connaissance de la consommation énergétique d'un programme et de la taille de la réserve d'énergie, il est impossible de garantir formellement la progression de ce dernier.

Dans ce cadre, nous nous sommes d'abord interrogés sur les aspects suivants :

— Où doit-on placer des points de sauvegarde dans un programme pour garantir sa progression sans compromettre ses performances ?

— Dans quelle mémoire (volatile ou non volatile) doit-on allouer les variables d'un programme ?

Placement mémoire et sauvegarde de l'état du programme

Placer les points de sauvegarde manuellement dans le programme est une tâche complexe qui nécessite une bonne connaissance du système dans son ensemble. En effet, un placement pessimiste des points de sauvegarde, c'est-à-dire des points de sauvegarde très rapprochés, mène à un surcoût de sauvegarde considérable. À contrario, un placement optimiste peut mener à une situation dans laquelle l'énergie disponible ne permet pas d'atteindre le prochain point de sauvegarde. Dans ce cas le capteur, incapable de sauvegarder son progrès, réexécute indéfiniment une même portion de code.

Pour diminuer le coût de ces sauvegardes, il est possible d'allouer certaines variables d'un programme en mémoire non volatile. En effet, il n'est alors plus nécessaire d'inclure ces variables dans les sauvegardes. Cependant, bien que les dernières technologies de mémoire non volatile soient efficaces, elles n'atteignent pas encore les latences d'accès et l'efficacité énergétique des mémoires volatiles. De ce fait, seules les variables rarement accédées ont intérêt à résider en NVM pour éviter les coûts de sauvegarde. Le bénéfice du choix d'une allocation mémoire dépend donc de différents paramètres, et doit être considéré à la granularité de la variable.

Les problèmes du placement de points de sauvegarde et d'allocation mémoire sont complexes par nature, mais aussi interdépendants. En effet, d'une part le choix d'une allocation mémoire influence l'énergie consommée par le programme, et donc les placements de points de sauvegarde possibles, et d'autre part ce placement défini le surcoût de sauvegarde, dictant le bénéfice d'une allocation mémoire. Gérer cette complexité manuellement est source d'erreurs et de mauvaises performances, d'où le besoin de méthodes automatisées.

Réveil après une perte d'alimentation

Les mécanismes de sauvegarde permettent de reprendre l'exécution du programme à partir du dernier point de sauvegarde après une perte d'alimentation. Naturellement, il est nécessaire de récolter de l'énergie avant de reprendre l'exécution dudit programme. Pour cela, les utilisateurs de capteurs sans batterie définissent un seuil, appelé *tension de réveil*. Ce seuil représente la tension aux bornes du condensateur qui doit être atteinte pour que le capteur se réveille après une perte d'alimentation. Usuellement, cette tension est choisie pour que le condensateur soit rempli lorsque l'exécution reprend.

Dans le cadre de cette thèse nous nous sommes posés les questions suivantes : *Est-il nécessaire d'attendre que le condensateur soit rempli avant de reprendre l'exécution du programme ? Est-ce bénéfique ?*

Pour répondre à ces questions, nous avons étudié l'impact du choix d'une tension de réveil sur l'exécution d'un programme intermittent. Nos expérimentations démontrent qu'une faible tension de réveil est bénéfique, car l'énergie consommée par le microcontrôleur augmente avec sa tension d'exécution. Seulement, l'exécution du code entre deux points de sauvegarde nécessite une quantité d'énergie minimale et une tension de réveil trop faible peut compromettre la capacité à assurer la progression du programme. La tension de réveil idéale pour un capteur serait alors la plus petite tension qui permet

d'assurer la progression du programme.

Cependant, nous soutenons que définir une tension de réveil unique à l'échelle du capteur n'est pas optimal et nous pensons qu'il est nécessaire d'adapter cette tension à chaque point de redémarrage potentiel. En effet, la tension de réveil requise pour reprendre l'exécution depuis un point donné du programme dépend de la charge de calcul à exécuter avant le prochain point de sauvegarde et cette charge peut grandement varier selon l'emplacement d'où l'exécution reprend. En adaptant la tension de réveil à chaque point de sauvegarde, il est possible de diminuer la tension d'exécution moyenne et donc de limiter la consommation d'énergie des capteurs sans batterie.

Pour cela, il est nécessaire de disposer d'outils capables de déterminer la tension de réveil optimale pour un point de sauvegarde donné, et de mécanismes à l'exécution pour faire appliquer ces choix de tension de réveil.

Contributions

Pour répondre à ces problématiques, nous avons proposé trois contributions :

— Schematic [RBB+24b], une technique de compilation pour le placement de points de sauvegarde et le choix d'une allocation mémoire. Schematic place des points de sauvegarde automatiquement de manière sûre, en adaptant ce placement à la taille de la réserve d'énergie. Schematic effectue aussi l'allocation mémoire des variables du programme pour réduire grandement la consommation d'énergie du système, tout en tenant compte de la taille limitée de la mémoire volatile.

— EarlyBird [RBB+24a], une technique pour sélectionner automatiquement une tension de réveil adaptée à la charge à exécuter. Plutôt que de choisir une tension de réveil à l'échelle du système, EarlyBird l'adapte à chaque endroit d'où l'exécution peut reprendre. À l'aide d'une analyse du binaire, EarlyBird s'assure que le placement de points de sauvegarde sélectionné puisse garantir la progression du programme et détermine la tension requise pour cela. À l'exécution, le système redémarre dès que la tension cible est atteinte, diminuant ainsi la tension d'exécution moyenne, et donc la quantité d'énergie dépensée.

— WORTEX [RAP24], un travail conjoint avec Abderaouf Nassim Amalou sur l'élaboration d'un modèle d'estimation de pire consommation d'énergie pour un microcontrôleur. Ce travail, non présenté dans ce manuscrit, utilise des techniques d'apprentissage automatique pour entrainer des modèles de prédiction plus ou moins complexes à prédire la pire consommation d'énergie d'un bloc de base.

Organisation du document

Dans le premier chapitre de ce manuscrit, nous revenons sur les raisons de l'apparition des systèmes sans batterie et les défis associés à ces derniers. Nous évoquons aussi les briques logicielles et matérielles nécessaires à la mise en place de tels systèmes. Le chapitre 2 présente les différentes solutions de l'état de l'art pour permettre l'exécution intermittente de programmes et positionne les travaux de la thèse.

Le chapitre 3 se concentre sur la description de SCHEMATIC tandis que le chapitre 4 présente EARLYBIRD.

Pour finir, nous établissons un bilan des solutions présentées dans ce manuscrit et identifions quelques perspectives ouvertes et qui pourront faire l'objet de travaux futurs.

Liste des contributions

[RAP24] Hugo REYMOND, Abderaouf Nassim AMALOU et Isabelle PUAUT. WORTEX : Worst-Case Execution Time and Energy Estimation in Low-Power Microprocessors Using Explainable ML, in *International Workshop on Worst-Case Execution Time Analysis*, 2024, DOI : 10.4230/OASIcs.WCET.2024.1.

[RBB+24a] Hugo REYMOND, Jean-Luc BÉCHENNEC, Mikaël BRIDAY, Sébastien FAUCOU, Isabelle PUAUT et Erven ROHOU. EarlyBird : Energy Belongs to Those Who Wake Up Early, in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'24)*, 2024.

[RBB+24b] Hugo REYMOND, Jean-Luc BÉCHENNEC, Mikaël BRIDAY, Sébastien FAUCOU, Isabelle PUAUT et Erven ROHOU. SCHEMATIC : Compile-Time Checkpoint Placement and Memory Allocation for Intermittent Systems, in *International Symposium on Code Generation and Optimization (CGO)*, 2024, DOI : 10.1109/CGO57630.2024.10444789.

Chapitre 1

Contexte du travail : Capteurs sans batterie

Dans ce chapitre, nous présentons tout d'abord comment le développement des technologies de récolte d'énergie a permis d'évoluer du capteur à piles au capteur sans batterie. Cette avancée permet d'améliorer radicalement l'autonomie de ces systèmes, mais implique un changement de paradigme où l'exécution du programme est rythmée par la disponibilité en énergie, rare et aléatoire. Nous évoquons ensuite les défis que posent ce nouveau paradigme ainsi que les briques matérielles et logicielles nécessaires pour y répondre.

1.1 Autonomie des capteurs sans fil : Comment alimenter les capteurs de demain ?

Les capteurs autonomes font partie intégrante de nos vies depuis un certain nombre d'années : thermostats connectés dans nos maisons, détecteurs de fumée, et plus récemment, capteurs de taux de CO_2 dans les lieux publics. Le déploiement de ces capteurs se multiplie à travers le monde, et on prévoit qu'un trilliard d'objets connectés soient en service d'ici à 2035 [Spa17]. Même si cette trajectoire, formulée en 2017, reste critiquable, notamment du point de vue de l'impact écologique d'un tel déploiement et de la pression sur les ressources rares, il n'en reste pas moins que nous observons aujourd'hui une augmentation importante du nombre de capteurs connectés.

Devant cette augmentation, la question de l'alimentation énergétique de ces capteurs se pose. Comment alimenter en énergie, de manière soutenable, les trilliards de capteurs qui sont voués à apparaitre dans les prochaines décennies ?

1.1.1 Capteur sans fil "classique"

Un capteur sans fil, ou *nœud capteur*, est un dispositif permettant de mesurer une grandeur physique, et de la communiquer via un réseau sans fil (Bluetooth, LoRa, RF...). Les capteurs sans fil sont des systèmes embarqués composés de différentes parties :

- Une unité de captage des données, qui transforme une grandeur physique, comme la température, en signal numérique.
- Une unité de traitement des données, qui fait l'acquisition du signal numérique et traite les données ainsi récoltées. Le microcontrôleur, « cerveau » du capteur sans fil, contrôle les unités de captage et de transmission.
- Une unité de transmission des données, qui envoie les données collectées à un nœud-puits (*sink* en anglais) connecté à internet, qui permettra la remontée d'information à l'utilisateur.

Les capteurs sans fil sont utilisés dans différents domaines, comme la surveillance de feu de forêt [YWM05 ; KCN09], la détection de fuites d'eau dans les réseaux de canalisations urbains [PHL19] ou encore la surveillance des écosystèmes [JOW+02 ; BDH+20]. Ces capteurs récoltant des données in-situ, et souvent dans des lieux isolés, la connexion au réseau électrique est impossible. Les capteurs sans fil sont donc par définition *autonomes*, capables de fonctionner sans être reliés à un réseau filaire, ni pour l'alimentation, ni pour la communication.

Pour ces capteurs autonomes, il est primordial de réduire au maximum la quantité de données transmises, la communication étant le poste de dépense principal des capteurs sans fil [Ber23]. Pour cela, ces capteurs dits « *intelligents* », appelés *smart sensors* en anglais, effectuent de plus en plus de traitements directement sur le microcontrôleur, une tendance que l'on nomme l'« informatique en périphérie » (*edge computing* en anglais). Ainsi, au lieu de transmettre les données brutes au centre de données pour qu'elles y soient traitées, ces données sont traitées en amont pour en extraire l'information et n'envoyer que l'information pertinente. Cela permet ainsi de réduire considérablement les coûts de transmission, mais requiert d'effectuer des calculs complexes (FFT, réseau de neurones) sur le capteur.

Aujourd'hui cependant, encore beaucoup de capteurs déployés restent relativement peu autonomes. Par exemple, l'*AudioMoth*[1], utilisé dans le cadre du suivi de la migration de la faune aviaire, enregistre les cris d'oiseaux pour permettre aux biologistes d'étudier le déplacement des nuées. Ce capteur est déployé en ville ou dans les zones naturelles, et ne

1. `https://www.openacousticdevices.info/audiomoth`

peut être relié au réseau électrique. Pour pallier cela, il est équipé de piles AAA. Cependant, malgré son apparente autonomie, ce capteur stocke les cris d'oiseaux sur une carte SD, et requiert une intervention humaine régulière pour remplacer cette carte. En effet, transmettre les cris enregistrés viderait sa batterie prématurément. Une opportunité pour rendre ces capteurs plus autonomes serait de pouvoir classifier l'espèce à qui appartient un cri sur le capteur pour transmettre l'information pertinente plutôt que de se reposer sur l'intervention humaine pour récupérer le signal brut. Le développement de tels capteurs « *intelligents* » permettrait de réduire l'intervention humaine dans ces milieux, souvent difficiles d'accès ou protégés.

Les prochaines sections reviennent sur l'évolution des capteurs sans fil pour devenir toujours plus autonomes, en se passant de batteries et en tirant parti de l'énergie disponible dans l'environnement. Dans cette thèse, nous visons à permettre à ces capteurs de continuer à réaliser des opérations complexes, malgré une alimentation électrique de plus en plus faible et instable.

Aparté : Autonomie de stockage

Dans cette thèse, nous nous concentrons principalement sur des problématiques d'autonomie énergétique, mais la gestion des grandes quantités de données enregistrées par les capteurs est un véritable problème. Par exemple, on trouve dans un manuel de déploiement de capteurs bio-acoustiques qu'il est nécessaire d'équiper ces capteurs d'un téraoctet de mémoire (deux cartes SD de 512Go) à remplacer tous les 6 à 12 mois [The19].
Pour pallier cela, certains travaux adoptent des approches « conscientes de l'information » (*information-aware* en anglais). On retrouve par exemple des travaux qui diminuent la fréquence d'envoi des données si les mesures effectuées correspondent à un modèle, pour fournir autant d'informations avec une quantité moindre d'énergie [JSW23 ; KRJ09].

De manière générale, les capteurs sont équipés de piles à usage unique, notamment de piles bouton (ou *coin cell* en anglais), ces petites piles plates qui équipent nos clés de voiture. Ces piles permettent aux capteurs de fonctionner pendant des mois, voire des années, en toute autonomie. Seulement, elles possèdent une quantité d'énergie finie, ce qui nécessite de les remplacer pour assurer le bon fonctionnement du capteur. Par exemple, dans une étude menée par Nikolic *et al.*, [NNS+17], les auteurs évaluent à 125 jours la durée de vie d'un capteur alimenté par pile bouton (CR2032), dans un scénario d'envoi de

données toutes les 5 minutes. Si l'on reprend l'exemple de l'AudioMoth, la page internet de ce dernier indique une autonomie bien plus basse, de 35 jours lorsqu'il est alimenté par trois piles AAA au lithium [2].

Ce remplacement de batterie nécessite l'intervention d'une personne et présente à la fois un impact écologique et économique : déplacement jusqu'au lieu de mesure, fabrication de la nouvelle pile, recyclage ou enfouissement de l'ancienne. Au vu des problématiques écologiques actuelles, il est essentiel de réduire cet impact.

De plus, même si dans certains cas les capteurs à pile semblent un bon compromis entre le coût (économique et écologique) et la simplicité d'utilisation, ils ne sont pas adaptés à toutes les situations. Prenons l'exemple d'un déploiement de capteurs pour la détection de feu de forêt au Liban [3]. Ce réseau de capteurs pilote compte déjà 91 capteurs répartis sur 78 hectares. Remplacer les batteries de chacun des capteurs tous les 4 mois, voire tout les 35 jours, n'est pas envisageable.

1.1.2 Vers une plus grande autonomie des capteurs : La récolte d'énergie

Face aux limites des piles classiques, des efforts de recherches ont été dirigés vers l'alimentation de ces capteurs avec de l'énergie ambiante, telle que l'énergie solaire. Ces capteurs, équipés de batteries, peuvent ainsi être autonomes pendant plusieurs années, sans besoin de maintenance.

Le tableau 1.1 présente les principales sources d'énergie utilisées à ce jour pour alimenter des capteurs sans fil, et la puissance récoltable. La récolte d'énergie solaire est communément utilisée car elle offre un bon rendement.

Cependant, même si la récolte d'énergie permet d'étendre la durée de vie des capteurs sans fil, cela vient néanmoins avec ses limites. En effet, les batteries utilisées pour ce genre d'applications sont très sensibles à leurs conditions d'opération. Elles supportent assez mal les températures extrêmes, et une sous-alimentation de la batterie endommage prématurément cette dernière. Ainsi, pour protéger les batteries, un système de gestion de batterie (*battery management system* en anglais, ou BMS) est nécessaire. Malheureusement, ce système ajoute un coût énergétique non négligeable, avec des rendements énergétiques qui atteignent 80 % pour des BMS spécialisés [EC16 ; Max18]. De plus, mal-

2. `https://www.openacousticdevices.info/getting-started` → *How do I calculate the battery lifespan ?*

3. `https://www.dryad.net/post/dryad-silvanet-detects-unauthorized-wildfire-in-lebanon`

Tableau 1.1 – Puissance disponible avec de la récolte d'énergie ambiante. Source : Mouser Electronics [Don]

Source	Puissance Source	Puissance Récoltée
Solaire		
Intérieur	0.1 mW/cm²	10 µW/cm²
Extérieur	100 mW/cm²	10 mW/cm²
Vibration/Mouvement		
Humain	0.5m at 1 Hz	
	1m/s² at 50 Hz	4 µW/cm²
Machine	1m at 5 Hz	
	10m/s² at 1 kHz	100 µW/cm²
Thermique		
Humain	20 mW/cm²	30 µW/cm²
Machine	100 mW/cm²	1-10 mW/cm²
RF		
GSM BSS	0.3 µW/cm²	0.1 µW/cm²

gré ces précautions, du fait de la nature aléatoire des sources d'énergie ambiante, il reste impossible de garantir que la batterie ne passera pas sous un seuil critique de tension.

Plus important encore, les batteries ont une durée de vie assez limitée, exprimée en cycles de charge/décharge (autour des 10 000 cycles [MTD10]). De ce fait, même si la récolte d'énergie permet d'augmenter l'autonomie des capteurs sans fil, la durée de vie de ces derniers est toujours limitée par la durée de vie de leur batterie. Cela devient un problème d'autant plus important que le système est isolé. On peut penser par exemple aux capteurs utilisés pour le suivi de la santé des structures d'ingénierie civile (ponts, bâtiments) qui sont coulés dans le béton [CKK+19], ou aux capteurs déployés dans l'espace [RAA+14], dont l'accès et donc la maintenance est impossible.

À l'heure où la récolte d'énergie permet d'augmenter considérablement l'autonomie des capteurs sans fil, les batteries apparaissent comme leur « maillon faible » en raison de leur durée de vie limitée et de leur impact environnemental. Cela soulève la question suivante : Pourrait-on se passer de ces batteries ?

1.1.3 Capteurs sans batterie

Depuis un certain nombre d'années maintenant, la recherche et l'industrie cherchent à se séparer des batteries. Les exemples les plus connus de systèmes sans batterie sont

probablement les systèmes de paiement sans contact ou les tags RFID utilisés comme antivols dans les magasins.

L'alimentation énergétique de ces systèmes précurseurs se fait avec de la récolte d'énergie, qui doit être suffisante pour alimenter ces systèmes en continu. On ne pourrait malheureusement pas appliquer les mêmes principes pour des capteurs autonomes, car la quantité d'énergie disponible dans l'environnement ne permettrait pas de répondre aux besoins de puissance instantanée nécessaire pour la communication longue distance, hormis avec des systèmes de récolte d'énergie disproportionnés. Pour pallier ce problème, des condensateurs ou supercondensateurs sont utilisés comme stockage d'énergie. Contrairement aux batteries, les condensateurs possèdent une durée de vie très longue (autour des 500 000 cycles [MTD10]) et ne nécessitent pas de circuit de charge complexe.

Avec un condensateur, même si la quantité d'énergie disponible dans l'environnement est faible, un capteur sans batterie peut fonctionner. Il passe une partie de son temps éteint, à récolter l'énergie environnante pour remplir le condensateur et ne devient actif que lorsque ce dernier est plein. Ce fonctionnement en phases actives/éteintes change alors le modèle d'exécution sous-jacent de ces capteurs, passant d'un modèle d'exécution *continu* à un modèle d'exécution *intermittent*.

L'exécution *intermittente* est définie comme l'alternance de phases actives et de phases d'arrêt ou de sommeil profond pendant lequel le système récolte de l'énergie, comme illustré sur la figure 1.1. Selon le type de capteur et la taille du condensateur utilisé, la durée d'une phase active peut varier de quelques millisecondes à plusieurs minutes. Ce nouveau paradigme force à séparer l'exécution de programmes en plusieurs phases de calcul séparées par des pertes d'alimentation, aussi appelées « cycles intermittents » ce qui soulève de nombreux défis, présentés ci-dessous.

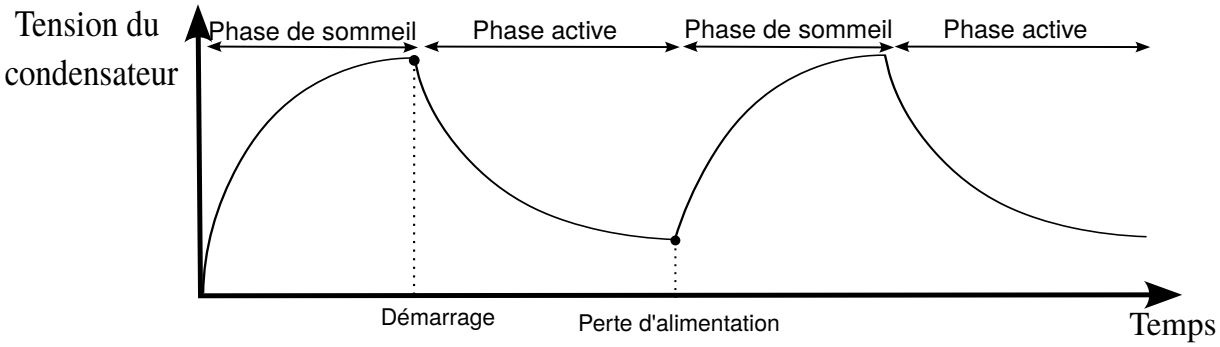


FIGURE 1.1 – Exécution intermittente

Défi 1 : Progression du programme Le principal défi des capteurs sans batterie est d'assurer la progression du programme malgré les pertes d'alimentation fréquentes. En effet, lorsque le système s'éteint, tout le contenu de la mémoire volatile du processeur est perdue, ainsi que le contenu des registres du processeur. De ce fait, toute la progression effectuée durant une phase active est perdue lorsque le système subit une perte d'alimentation. Il est donc nécessaire de mettre en place des mécanismes pour sauvegarder cette progression.

Défi 2 : Exécution atomique L'exécution *intermittente* d'un programme classique répartit l'exécution des différentes opérations du programme en plusieurs phases d'exécution. Ces phases d'exécution sont séparées par des pertes d'alimentation, qui peuvent être plus ou moins longues. S'il est souvent possible d'interrompre et de reprendre l'exécution d'un calcul à n'importe quel endroit, ce n'est pas le cas de toutes les opérations. Certaines opérations sont par nature insécables, comme l'envoi d'un message radio ou la mesure d'une grandeur physique. On parle alors d'opération *atomique*. Cette opération devra être effectuée dans un seul cycle d'exécution, et, dans le cas où une perte d'alimentation survenait pendant cette opération, il serait nécessaire de reprendre l'opération depuis le début.

1.2 Construire un capteur sans batterie, problématique et mode d'emploi

Pour répondre à ces défis, les capteurs sans batterie reposent sur différentes technologies, que ce soit au niveau matériel ou au niveau logiciel. Cette section présente les composants matériels nécessaires au développement de capteurs sans batterie (section 1.2.1) et les briques logicielles nécessaires à leur fonctionnement (section 1.2.2).

1.2.1 Composants matériels

Comme tout système embarqué, un capteur sans batterie est un système complexe. Dans cette section, nous ne présenterons pas en détail l'architecture d'un tel capteur, mais plutôt les éléments qui le différencient d'un capteur traditionnel.

Le condensateur, ou supercondensateur, est un composant essentiel des capteurs sans batterie. Ce condensateur, d'une capacité d'une centaine de microfarads à plus d'une

dizaine de farads, sert de stockage d'énergie (*energy buffer* en anglais). Il a deux objectifs principaux. Son premier objectif est de permettre au capteur de fonctionner de manière intermittente, lorsque la quantité d'énergie récoltée ne suffit pas à alimenter le capteur de manière continue. Le second objectif de ce condensateur est de pouvoir fournir une grande quantité d'énergie instantanée pour l'exécution de périphériques avec une forte puissance. Alimenter des périphériques tels qu'une radio directement avec de l'énergie solaire demanderait une grande surface de panneau solaire, surdimensionnée pour la majorité du temps, pendant lequel la radio n'émet pas. L'appellation supercondensateur désigne les condensateurs dont la densité énergétique est importante. Par abus de langage, nous parlerons dans cette thèse de condensateur pour évoquer aussi bien les condensateurs que les supercondensateurs.

Le microcontrôleur est au centre d'un capteur sans batterie. Pour être compatible avec l'exécution intermittente, le microcontrôleur doit pouvoir fonctionner sur une large plage de tension. En effet, contrairement à la tension des batteries qui reste relativement stable pendant leur décharge, la tension aux bornes d'un condensateur évolue grandement en fonction de l'énergie qu'il contient [4]. Si ce dernier n'accepte pas une grande plage de tension d'entrée, il est alors nécessaire d'équiper le capteur d'un convertisseur DC/DC (Buck/Boost), qui convertit la tension d'entrée (aux bornes du condensateur) en une tension de fonctionnement acceptable par le microcontrôleur.

La mémoire non volatile (*Non-Volatile Memory* en anglais, abrégé NVM) est essentielle au développement des systèmes sans batterie, car les pertes d'alimentation fréquentes forcent les capteurs sans batterie à régulièrement sauvegarder l'état du programme dans une mémoire non volatile. L'arrivée de nouvelles technologies de mémoire non volatile telles que la FRAM (*Ferroelectric RAM*) offre des performances énergétiques sans précédent comparé aux technologies FLASH et permettent ainsi de rendre viables les capteurs sans batterie, en réduisant considérablement le coût d'une sauvegarde. Aujourd'hui, beaucoup de mémoires non volatiles externes sont disponibles sur le marché et certains constructeurs intègrent même, comme TI avec sa série des MSP430FR, de la mémoire FRAM interne dans leurs microcontrôleurs.

Le convertisseur analogique-numérique (CAN), ou *analog to digital converter (ADC)* en anglais, permet de convertir une tension analogique en une valeur numérique. Il est utilisé dans les capteurs sans batterie pour mesurer la quantité d'énergie contenue dans le condensateur. On parle alors de capteur conscient de l'énergie (*energy-aware sensor* en

4. $E = \frac{1}{2}CV^2$, avec C la capacité du condensateur, et V la tension à ses bornes.

anglais).

Variations. Les composants décrits ci-dessus sont utilisés dans l'architecture typique de capteurs sans batterie, mais d'autres modèles d'architecture existent dans la littérature.

Certaines architectures reposent sur un processeur dit « non volatile », c'est-à-dire dont les registres sont dupliqués, avec une version volatile et une version non volatile [STS+17]. Le registre volatil est utilisé par le processeur et sa version non volatile permet une sauvegarde quasiment instantanée, déclenchée par un signal logique. Sans rentrer à l'intérieur du processeur, des ajouts matériels, souvent placés entre le processeur et la mémoire, permettent aussi une sauvegarde efficace de l'état du programme [Hic17 ; CKL+20 ; PMS21]

Certaines architectures utilisent un stockage d'énergie « fédéré » [HSS15 ; YY20], où chaque périphérique dispose de son propre condensateur, voire « reconfigurable » [CRL18], c'est-à-dire composé de plusieurs condensateurs découplés du circuit de recharge au besoin, pour varier la capacité du stockage d'énergie à l'exécution.

Enfin, pour des raisons d'efficacité énergétique, les capteurs sans batterie font appel à des accélérateurs matériels, pour le traitement du signal [LBB+21] et plus récemment l'inférence de réseaux de neurones [BGd+22].

Architecture considérée. Dans le cadre de cette thèse, nous considérons l'architecture schématisée sur la figure 1.2 avec un unique condensateur et un processeur volatile TI MSP430 [Ins12] équipé de mémoire ferromagnétique (FRAM).

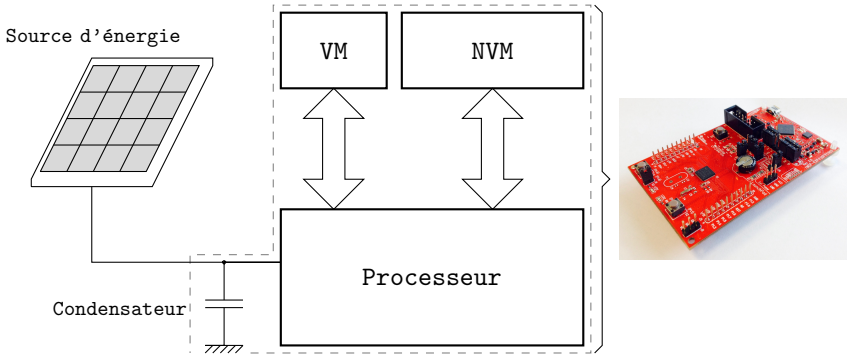


FIGURE 1.2 – Architecture matérielle considérée dans le cadre de cette thèse

1.2.2 Composants logiciels

Pour fonctionner, un capteur sans batterie ne dépend pas uniquement d'une architecture matérielle adaptée, mais aussi de briques logicielles spécifiques.

Dans cette section, nous présentons tout d'abord les techniques de sauvegarde de l'état du programme (*checkpointing* en anglais). Ces techniques sont la pierre angulaire d'un capteur sans batterie, car elles permettent de répondre au défi de la progression du programme. Ensuite, nous présentons rapidement comment le choix d'exécuter sur machine nue, ou de passer par l'intermédiaire d'un support d'exécution impacte le fonctionnement d'un programme intermittent.

Le *checkpointing*, ou l'art de sauvegarder l'état d'un programme

Pour assurer la progression du programme, deux fonctions sont nécessaires, que l'on nomme usuellement `checkpoint()` et `restore()`. La première fonction, `checkpoint()`, sauvegarde l'état courant du programme dans la mémoire non volatile. La seconde fonction, `restore()`, appelée après le redémarrage du capteur, restaure l'état du programme à partir de la sauvegarde et reprend l'exécution.

Sauvegarder les données volatiles Pour sauvegarder les données volatiles du programme, il est nécessaire de copier la mémoire volatile et les registres du processeur dans la mémoire non volatile. Cela peut être effectué de manière logicielle via le biais du processeur lui-même, ou accéléré matériellement en utilisant un DMA (abréviation de *Direct Memory Access* en anglais, ou *Accès Direct à la Mémoire* en français).

Pour assurer le cas où une perte d'alimentation arrive pendant la sauvegarde, une technique consiste à réserver deux emplacements mémoire pour la sauvegarde, qui sont utilisés chacun leur tour (*Double buffering*). Dans le cas où une sauvegarde échoue, l'autre emplacement contient ainsi toujours une sauvegarde antérieure complète.

Lors d'un redémarrage après une perte d'alimentation, la fonction `restore()` copie les données depuis la NVM vers la VM et repeuple les registres du processeur. Le dernier registre à être restauré est le compteur de programme. Lorsque ce dernier est restauré, l'exécution de programme reprend.

Sauvegarder les données non volatiles Le développement de mémoires non volatiles efficaces permet l'utilisation de ces dernières comme mémoires de travail. Les mémoires non volatiles, bien que moins efficaces énergétiquement parlant, présentent l'avantage de garder leur contenu lorsque le capteur connait une perte d'alimentation.

À première vue, il ne semble alors pas nécessaire de sauvegarder l'état des données non volatiles. Seulement, si la mémoire non volatile est modifiée après que les données volatiles

aient été sauvegardées, l'état de la sauvegarde devient incohérent avec les données en NVM [MMA+21 ; RL14]. De ce fait, si une perte d'alimentation survient, il est impossible de restaurer le programme dans un état cohérent.

Pour illustrer ce phénomène, prenons l'exemple de la fonction `f` visible sur la figure 1.3a. Dans cet exemple, on suppose que la variable `i` est stockée en mémoire non volatile (NVM). On observe sur la figure 1.3b, que si l'on exécute le programme de manière intermittente, une anomalie mémoire apparait lorsque le système subit une perte d'alimentation avant l'instruction ligne 5. En effet, en ré-exécutant la portion de code ligne 4 la variable i est incrémentée deux fois.

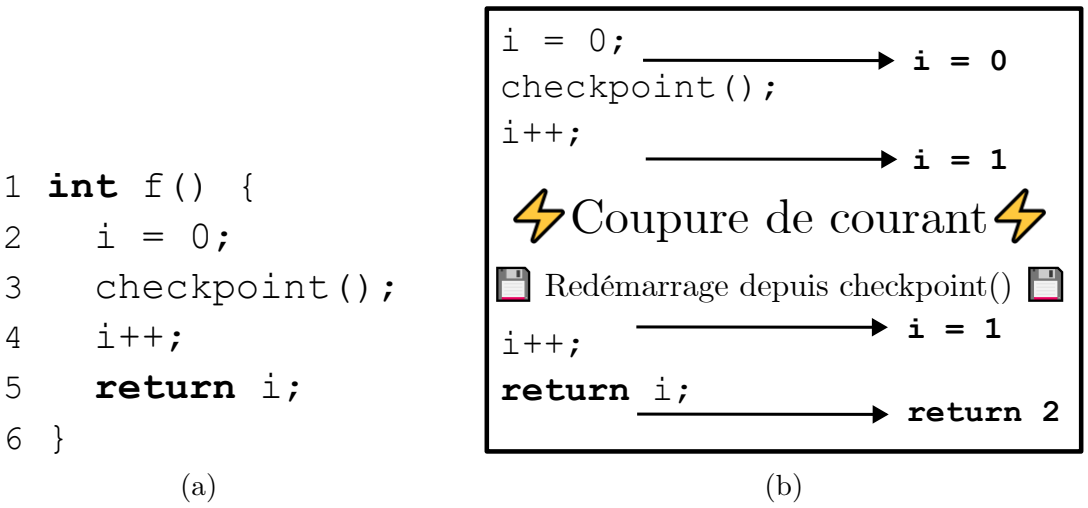


FIGURE 1.3 – Exemple d'anomalie mémoire. La fonction `f()`, visible sur la figure (a), renvoie toujours 1 en exécution continue. Cependant, l'exécution intermittente de `f` figure (b) donne lieu à une anomalie mémoire. Comme la variable `i` est stockée en NVM, la variable est incrémentée deux fois à la suite d'une perte d'alimentation. Cet état mémoire est incohérent, il n'arrive pas lors d'une exécution continue. On parle ici d'anomalie mémoire.

On parle dans ce cas d'une portion de code non *idempotente*, c'est-à-dire que son résultat change si on la réexécute. Pour pallier ce problème, il est nécessaire de sauvegarder l'état de la NVM au checkpoint ligne 3. Ainsi, lors de l'exécution de `restore()`, on remet la NVM dans son état sauvegardé pour "défaire" les modifications effectuées depuis le dernier checkpoint. On parle alors en anglais de *undo-logging*.

Nous examinerons en détail les techniques de sauvegarde de la progression du programme et l'utilisation de la mémoire NVM comme mémoire de travail tout au long de ce manuscrit.

Aparté : Sauvegarder l'état des périphériques

Dans cette section, nous avons évoqué la sauvegarde de l'état du processeur, mais il est aussi nécessaire de sauvegarder l'état des périphériques, ce qui représente un défi à part entière. Pour les périphériques les plus simples, comme des entrées/sorties logiques, il est possible de sauvegarder leur état en copiant le contenu de leur registre de configuration en mémoire non volatile lors d'une sauvegarde. Cependant, les périphériques plus complexes (radio, ADC, capteur...) possèdent une machine à état interne, qu'il est impossible de sauvegarder. Pour pallier cela, différentes techniques ont été proposées [BDM+19 ; RBM+18].

Support d'exécution pour les systèmes intermittents

Les capteurs sans fil sont classiquement équipés d'un système d'exploitation (*Operating System*, OS, en anglais), comme ContikiOS [ODE+22]. Ce dernier fournit un ensemble de fonctionnalités aux applications, comme la connectivité réseau et l'ordonnancement des tâches. Bien que dimensionnés pour les systèmes embarqués et donc très peu gourmands en ressources mémoire comme processeur, ces OS sont encore trop gros pour l'exécution intermittente [5].

Ainsi une grande partie de l'état de l'art développe des applications sur machine nue (*Bare-Metal* en anglais), c'est-à-dire sans OS. Cependant, certains travaux proposent des supports d'exécution dédiés aux applications intermittentes.

Ces supports très légers ne gardent que l'essence des OS traditionnellement utilisés : La gestion des ressources partagées. Ils permettent en effet l'exécution concurrente [YMP+18 ; YCY22], voire parallèle [AY22], le partage de l'énergie [ML20] ou des périphériques [BDM+19]. De plus, ces derniers fournissent des mécanismes de sauvegarde pour supporter l'exécution intermittente.

Dans cette thèse, nous ne considérons pas de support d'exécution. En effet, ces derniers peuvent avoir un surcoût à l'exécution et imposent d'utiliser le modèle de programmation de « micro-tâches », comme nous le verrons dans le prochain chapitre.

Capteurs sans batterie : Défis et solutions

Ce chapitre a retracé comment le besoin d'une autonomie toujours plus importante

5. « Une configuration typique de Contiki (le noyau et le chargeur de programmes) consomme 2 kilooctets de RAM et 40 kilooctets de ROM », extrait de la page Wikipédia de Contiki `https://fr.wikipedia.org/wiki/Contiki`

des capteurs sans fil a mené à l'apparition et au développement des capteurs sans batterie. Ces derniers reposent sur l'existence de composants matériels et de mécanismes logiciels pour faire face aux pertes d'alimentation fréquentes. Dans le prochain chapitre, nous présenterons plus en détail comment les techniques de l'état de l'art relèvent le défi de la progression du programme dans ce contexte d'exécution intermittente.

Chapitre 2

État de l'art : Techniques de checkpointing pour les systèmes intermittents

Dans ce chapitre, nous présentons les différentes méthodes pour assurer la progression d'un programme dans un contexte intermittent. Dans un premier temps, nous présentons les différentes classes de techniques existantes : le *checkpointing dynamique*, le *checkpointing basé sur les tâches* et le *checkpointing statique*. Ensuite, nous nous concentrerons sur la technique au centre de cette thèse : le *checkpointing statique*.

2.1 Classes de techniques pour assurer la progression du programme

2.1.1 Checkpointing dynamique, ou « Juste à temps »

Le checkpointing dynamique, ou juste à temps, est une technique de sauvegarde qui consiste, comme son nom l'indique, à attendre le dernier moment pour sauvegarder. Pour cela, cette technique doit évaluer en continu l'énergie disponible pour alimenter le système, typiquement en mesurant la tension aux bornes du condensateur, et déclencher une sauvegarde dès que cette tension passe sous un seuil défini (voir figure 2.1).

Cette technique présente l'avantage de sauvegarder uniquement lorsque l'énergie vient à manquer et donc de ne pas faire de sauvegarde inutile. Ainsi, le capteur s'adapte à l'énergie disponible dans l'environnement : s'il récolte plus d'énergie qu'il n'en consomme, alors il ne sauvegardera pas. Dans le cas contraire, la fréquence de sauvegarde augmentera.

De plus, cette technique est la plus simple à mettre en place pour le développeur, car il n'est pas nécessaire de modifier les programmes existants pour les exécuter de manière

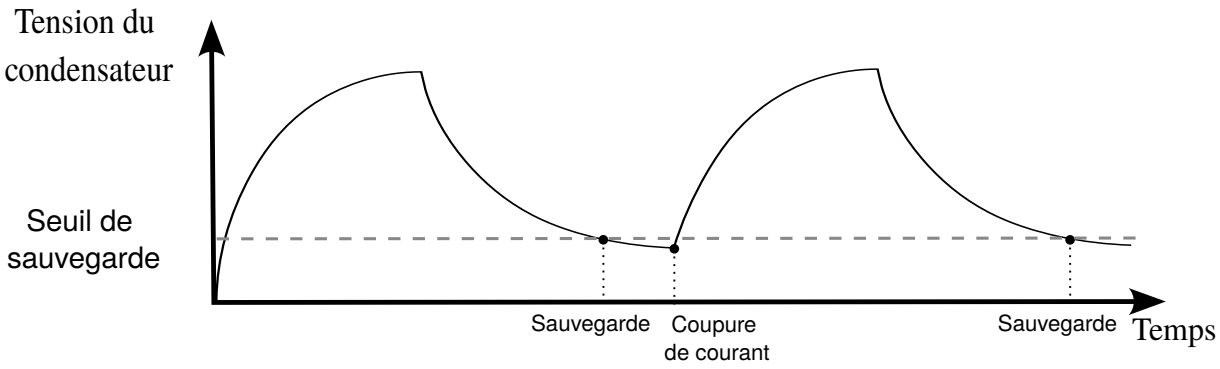


FIGURE 2.1 – Checkpointing Dynamique

intermittente. A priori, cette technique serait la meilleure candidate pour l'adaptation de codes dits « legacy ».

Malheureusement, cette technique possède des limites importantes. En effet, comme la sauvegarde peut être effectuée à tout moment lors de l'exécution du programme, il est impossible de savoir à l'avance à quel endroit du code la sauvegarde aura lieu. Cela pose plusieurs problèmes. Premièrement, le lieu de sauvegarde n'est pas forcément le plus optimal, notamment vis-à-vis de la quantité de données à sauvegarder. Il est donc nécessaire de surestimer le seuil de tension de sauvegarde pour prendre en compte la situation pire cas : la sauvegarde de l'état du programme à l'endroit où l'empreinte mémoire est maximale. Enfin, plus important encore, il est impossible de définir des sections atomiques avec cette technique de sauvegarde. L'utilisation de périphériques est donc naturellement complexe, même si certains supports d'exécution permettent de gérer ce problème [BDM+19]. De plus, la mesure en continu de la tension aux bornes du condensateur entraine un surcoût énergétique non négligeable.

Un exemple de code pour du checkpointing dynamique est visible Extrait 1. On observe qu'aucune modification n'est nécessaire pour la fonction `main()`.

```c
// Computes and send the sum of `array`
void main() {
    int sum = 0;
    for(int i = 0; i < 10; i++) {
            sum += array[i];
    }
    send(sum);
}
```

```c
// Interruption called when voltage
// reaches a low threshold
void low_voltage_interrupt() {
    checkpoint();
}
```

Extrait 1 – Exemple de programme instrumenté avec du checkpointing dynamique

2.1.2 Checkpointing basé sur des tâches

Le checkpointing basé sur les tâches demande au développeur d'organiser son programme en différentes sections atomiques, appelées tâches ou micro-tâches. Ces dernières requièrent un support d'exécution pour gérer l'enchainement des tâches et la sauvegarde de l'état du programme. De manière générale, cette sauvegarde est effectuée pendant les transitions entre les différentes tâches. L'exécution des tâches est transactionnelle, de manière à garder la sémantique atomique de la tâche. Si une perte d'alimentation intervient pendant l'exécution d'une tâche, alors elle est réexécutée entièrement.

Cette technique offre l'avantage d'organiser le code en sections atomiques explicites. De plus, de par la séparation en tâches, il est souvent nécessaire d'expliciter le partage de données entre les tâches, que ce soit par le biais d'un canal de communication [CL16 ; HSS17], de files (FIFO) [BBF+23] ou en explicitant quelles variables sont partagées ou locales [MCL17]. De cette manière, les données à sauvegarder sont clairement identifiées, réduisant grandement le coût d'une sauvegarde. Les données locales à une tâche (pile et registres) n'ont pas besoin d'être sauvegardées.

Aussi, la structuration du programme en micro-tâches facilite l'exécution concurrente, voire parallèle de programmes intermittents.

Cependant, cette technique vient au prix d'une programmation complexe. Il est nécessaire de découper le code en micro-tâches d'une taille adaptée à la réserve d'énergie. Le processus de découpage en micro-tâches est long, complexe et peut nuire à la lisibilité du code. Le découpage du programme en micro-tâches peut impliquer des transformations non triviales. Ainsi, dans le cas où le nombre d'itérations d'une boucle est trop grand pour garantir une exécution même lorsque la réserve d'énergie est pleine, il est nécessaire de séparer cette boucle en plusieurs micro-tâches, rendant le flot de contrôle confus et complexifiant la lisibilité du code. Un exemple de code découpé en micro-tâches est visible Extrait 2. Ce même code dans le cas où la boucle `for` ne peut pas s'exécuter avec un condensateur plein est visible Extrait 3. Ce découpage en micro-tâches doit être effectué à nouveau dès que la taille du condensateur considéré varie, complexifiant d'autant plus le processus de prototypage.

```
// Computes the sum of `array`
TASK compute_sum() {
    INPUT(array);
    int sum = 0;
    for(int i = 0; i < 10; i++) {
        sum += array[i];
    }
    OUTPUT(sum);
    NEXT_TASK(send);
}
```

```
// Send the value of the sum
TASK send() {
    INPUT(sum);
    send(sum);
}
```

Extrait 2 – Exemple de programme instrumenté avec du checkpointing basé sur les micro-tâches. On considère que la boucle `for` peut s'exécuter avec un condensateur plein

```
// Launch computation of `array` sum
TASK compute_sum() {
    INPUT(array);
    int sum,i = 0;
    OUTPUT(array, sum, i);
    NEXT_TASK(sum_one_iteration);
}

// Send the value of the sum
TASK send() {
    INPUT(sum)
    send(sum)
}
```

```
// Computes one iteration of the sum
TASK sum_one_iteration() {
    INPUT(array, sum, i)
    sum += array[i];
    i++;
    if(i < 10) {
        OUTPUT(array, sum, i);
        NEXT_TASK(sum_one_iteration);
    } else {
        OUTPUT(sum);
        NEXT_TASK(send);
    }
}
```

Extrait 3 – Exemple de programme instrumenté avec du checkpointing basé sur les micro-tâches. On considère que la boucle `for` ne peut pas s'exécuter avec un condensateur plein

2.1.3 Checkpointing statique

Le checkpointing statique consiste à insérer des points de sauvegarde dans un programme classique, pour lui permettre de survivre aux pertes d'alimentation. Contrairement au checkpointing basé sur des tâches, les points de sauvegarde peuvent être placés à n'importe quel endroit du programme. Contrairement au checkpointing dynamique, les positions des points de sauvegarde ne sont pas déterminées à l'exécution, mais à la compilation.

Le checkpointing statique se présente comme un compromis entre l'efficacité de sauvegarde du checkpointing basé sur les tâches et la granularité du checkpointing dynamique. En effet, les points de sauvegarde étant placés statiquement, il est possible de bénéficier d'optimisations à la compilation (notamment sur la quantité de données sauvegardées)

tout en gardant la possibilité de définir des sections atomiques (sections entre deux points de sauvegarde) pour un moindre coût de développement. Un exemple de code instrumenté avec du checkpointing statique est visible Extrait 4.

```
// Computes and send the sum of `array`
void main() {
    int sum = 0;
    for(int i = 0; i < 10; i++) {
        checkpoint();
        sum += array[i];
    }
    checkpoint();
    send(sum);
    checkpoint();
}
```

Extrait 4 – Exemple de programme instrumenté avec du checkpointing statique

Du fait de la granularité du placement des points de sauvegarde statique (à l'instruction près), il est plus facile de déployer des techniques de placement automatique et ces placements sont facilement adaptables en cas de changement de taille de condensateur. Un des désavantages du checkpointing statique est que les sections atomiques sont plus complexes à visualiser qu'avec une organisation en tâche et les variables partagées entre deux sections atomiques sont plus difficilement identifiables.

Le Tableau 2.1 récapitule les avantages et inconvénients des trois familles de techniques pour assurer la progression du programme. Ces familles restent des catégorisations générales, et beaucoup de techniques représentent des hybrides entre plusieurs familles [ML18 ; ML19].

Tableau 2.1 – Avantages et inconvénients des techniques pour assurer la progression du programme

Critère	Dynamique	Tâches	Statique
Support des sections atomiques	✗	✓	✓
Facilité d'utilisation	+++	-	+
Surcoût de sauvegarde	Haut	Très bas	Bas
Granularité	Instruction	Fonction	Instruction
Exemples	[JRR14 ; BWM+15 ; BDM+19 ; NCA+23]	[BGW11 ; YMP+18 ; SSD+22 ; ML20]	[RSF11 ; ZFL+17 ; YR20 ; KGH+22]

Dans cette thèse, nous nous concentrons sur le checkpointing statique, car il offre

un modèle de programmation peu contraint (pas de séparation en tâches), peut bénéficier d'optimisations à la compilation et permet la définition de sections atomiques. La prochaine section explore plus en détail les travaux apparentés au checkpointing statique.

2.2 Défis du checkpointing statique

Les efforts de recherche dans le domaine du checkpointing statique sont principalements orientés autour de trois axes :

— Le placement des points de sauvegarde : *Où doit-on sauvegarder*

— La réduction de la taille de la sauvegarde : *Que doit-on sauvegarder ?*

— L'adaptation à l'exécution : *Quand doit-on sauvegarder ?*

Dans cette section, nous revenons en détails sur ces différents axes, et les techniques proposées dans l'état de l'art.

2.2.1 Placement de points de sauvegarde

Le checkpointing statique possède l'avantage de permettre un placement précis des points de sauvegarde, à la granularité de l'instruction. Seulement, le placement de ces points de sauvegarde peut se révéler complexe, c'est pourquoi des techniques de placement automatique des points de sauvegarde ont été développées, pour réduire la charge du développeur. On retrouve ainsi quatre approches pour le placement automatique de points de sauvegarde : *selon la structure du code*, *selon l'empreinte mémoire*, *selon les schémas d'accès mémoire* et *selon la taille de la réserve d'énergie*

Placement de points de sauvegarde conditionné par la structure du code

La première approche, sûrement la plus naturelle, est de placer les points de sauvegarde (manuellement ou non), selon la structure du code. On pourra par exemple, comme proposé dans Mementos [RSF11], placer des points de sauvegarde avant chaque retour de fonction, ou à chaque itération d'une boucle.

Ces techniques présentent l'avantage d'être très simples à mettre en œuvre. Elles induisent généralement une fréquence de sauvegarde élevée, permettant ainsi d'assurer le progrès du programme même dans des scénarios d'intermittence extrêmes. Malheureusement, ces sauvegardes très fréquentes conduisent à une exécution des programmes très peu efficace sur le plan énergétique.

Placement de points de sauvegarde conditionné par la mémoire

Dans un but de diminuer le coût des sauvegardes de l'état du programme, plusieurs équipes se sont attelées à placer des points de sauvegarde aux endroits où la quantité

de données à sauvegarder est moindre. Ainsi, Shoemaker *et al.*, [SPS20] motivent le fait de placer des points de sauvegarde aux emplacements où l'empreinte mémoire du tas est minimale. Dans le domaine du checkpointing dynamique, mais applicable au checkpointing statique, Zhao *et al.*, [ZFL+17] proposent de déterminer l'ensemble des données vivantes à chaque emplacement du programme, pour déterminer les endroits favorables à la sauvegarde.

Seulement, ces travaux ne se concentrent pas sur la réduction de la quantité de données à sauvegarder dans les boucles. Or, dans beaucoup d'applications, les boucles représentent le cœur des calculs effectués. Fort de ce constat, Li *et al.*, proposent une technique pour réduire la quantité de données vivantes, et donc à sauvegarder, dans les boucles imbriquées. Pour cela, le code de l'application est modifié en appliquant automatiquement du tuilage de boucle (*loop-tiling* en anglais) de manière à limiter les dépendances de données entre les tuiles. Des sauvegardes sont effectuées entre chaque tuile et uniquement les données liées à des dépendances entre tuiles sont sauvegardées, réduisant ainsi la quantité de données à sauvegarder. Les paramètres de ce tuilage sont déterminés automatiquement de manière à respecter la capacité de la réserve d'énergie, et à minimiser la quantité de données à sauvegarder entre chaque tuile. Cette technique est illustrée par la figure 2.2.

Ces travaux permettent ainsi de réduire considérablement la quantité de données volatiles à sauvegarder. Pour réduire d'autant plus la quantité de données volatiles, une autre solution est de tirer parti de la mémoire non volatile comme mémoire de travail.

Placement de points de sauvegarde pour supprimer les dépendances WAR

Les premières techniques à avoir proposé d'utiliser de la mémoire non volatile comme mémoire de travail pour le calcul intermittent reposent sur de la sauvegarde dynamique. Les calculs sont effectués avec les données en mémoire non volatile et les registres sont sauvegardés au dernier moment (avant une perte d'alimentation). Cependant, en checkpointing statique, utiliser la mémoire non volatile (NVM) comme mémoire de travail génère des anomalies mémoires dès lors que des dépendances écriture-après-lecture (*write-after-read* en anglais, ou WAR) sont présentes. En effet, comme précisé dans la section 1.2.2, une différence de version entre les données volatiles sauvegardées et les données non volatiles peut amener à des situations impossibles en exécution continue.

Pour lutter contre ces anomalies, deux solutions sont possibles :

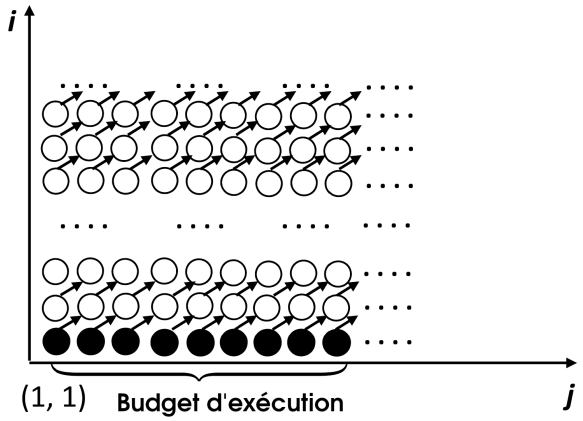
— Effectuer une sauvegarde des données non volatiles lorsqu'un point de sauvegarde est atteint, ce qui supprime l'intérêt d'utiliser une mémoire non volatile comme

```
int i, j, k=2;
for(i=1;i<=30;i++)
   for(j=1;j<=30;j++)
     a[i][j] += a[i-1][j-1]/k;
```

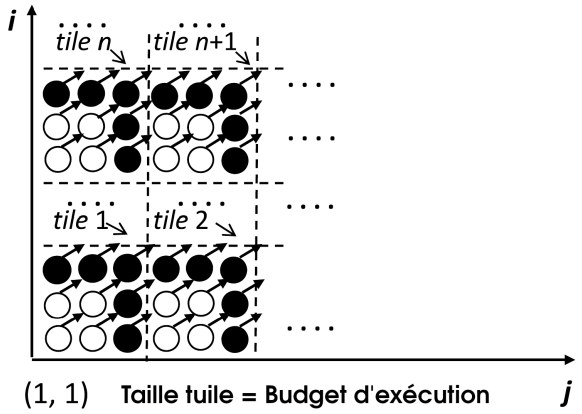
(a) Code de la boucle étudiée. Le papier original considère que les résultats intermédiaires du calcul ne seront pas accédés après l'exécution de cette boucle, seulement l'élément `a[30][30]` du tableau sera accédé.

```
int i, j, x, y, k=2;
for(y=1;y<=30;y+=3)
   for(x=1;x<=30;x+=3)
      for(i=y;i<y+3;i++)
         for(j=x;j<x+3;j++)
            a[i][j] += a[i-1][j-1]/k;
```

(b) Code de la boucle après tuilage de cette dernière.



(c) Sans tuilage de boucle : Au vu des dépendances de données (représentées par les flèches), l'état du programme à sauvegarder est de 9 nœuds.



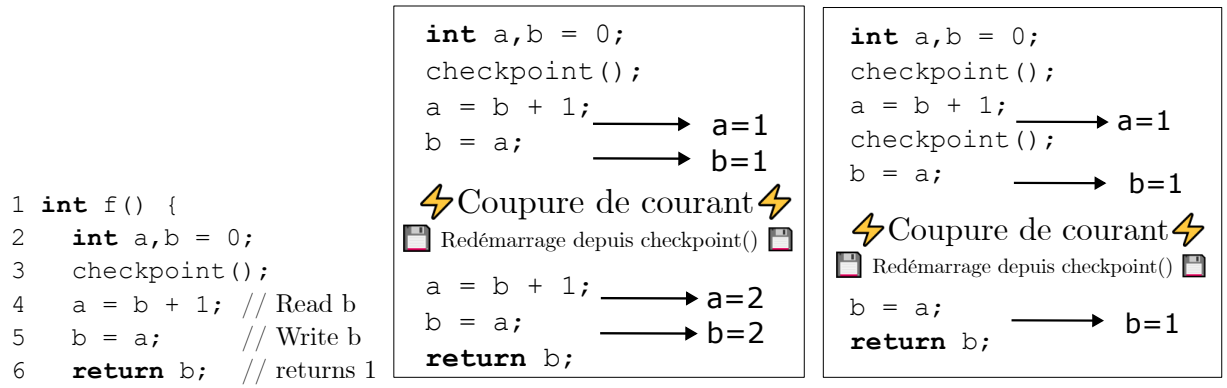
(d) Avec tuilage de boucle : Au vu des dépendances de données, l'état du programme à sauvegarder est de 5 nœuds uniquement. De cette manière, le coût de sauvegarde est diminué de 44% par rapport à l'exécution sans tuilage.

FIGURE 2.2 – Exemple d'application de tuilage de boucle pour réduire la quantité de données à sauvegarder. Ici, la capacité de la réserve d'énergie est de 9 calculs, chaque calcul étant représenté par un nœud. Les calculs dont le résultat doit être sauvegardé sont représentés en noir. Source : Papier de Li *et al.*, [LQZ+19]

mémoire principale

— Découper les dépendances WAR en insérant un point de sauvegarde entre chaque lecture et écriture à la même variable.

Cette dernière solution a été choisie par les auteurs de RATCHET [WH16]. RATCHET instrumente un programme à la compilation avec des points de sauvegarde, placés entre chaque dépendance WAR. De cette manière, RATCHET s'assure qu'aucune anomalie mémoire puisse apparaitre, comme illustré sur la figure 2.3



(a) Fonction `f`

(b) Exécution intermittente avant découpage

(c) Exécution intermittente après découpage

FIGURE 2.3 – Placement de points de sauvegarde pour découper les dépendances WAR. La fonction `f`, visible sur la figure (a), retourne toujours 1 en exécution continue. En exécution intermittente, une anomalie mémoire apparait si une perte d'alimentation survient avant l'exécution de l'instruction `return` ligne 6, comme illustré sur la figure (b). En ajoutant un point de sauvegarde entre la dépendance WAR (l.4 et l.5), il n'y a plus d'anomalie mémoire.

Cependant, séparer toutes les dépendances WAR par des points de sauvegarde peut mener à placer énormément de points de sauvegarde. Pour éviter cela, WARIO [KGH+22] réutilise le même principe que Ratchet mais réordonne les instructions pour permettre de couper plusieurs dépendances WAR avec un même point de sauvegarde. Pour cela, une technique consiste par exemple à placer toutes les lectures en début de fonction, pour finir avec les écritures. De cette manière WARIO diminue considérablement la quantité de points de sauvegarde dans un programme, et donc leur surcoût.

Placement de points de sauvegarde conditionné par le « budget d'exécution »

Toutes les solutions évoquées jusque-là ne considèrent pas la taille de la réserve d'énergie, un élément pourtant essentiel pour le placement des points de sauvegarde. En effet, ce

dernier définit le « budget d'exécution » disponible pour un cycle d'exécution, en fonction de la quantité d'énergie qu'il contient une fois rempli. Ainsi, ne pas considérer le budget d'exécution peut facilement mener à un placement de points de sauvegarde pessimiste, où la majorité du temps est passé en sauvegarde, voire optimiste, où la progression du programme n'est pas assurée. Pour pallier cela, différentes techniques se sont intéressées à ce budget d'exécution.

Une des premières techniques documentée dans la littérature scientifique sur le sujet est HarvOS [BM17], qui considère un budget d'exécution temporel, c'est-à-dire un nombre de cycles d'exécution avant une perte d'alimentation. Pour garantir un placement de points de sauvegarde sûr, HarvOS fait en sorte de créer des régions de code qui peuvent s'exécuter avec la moitié du budget d'exécution. Entre ces régions de code, l'état du programme est sauvegardé, ce qui permet de garantir la progression du programme. Similairement, Baghsorkhi et Margiolas [BM18] ont proposé de séparer le programme en différentes régions simple-entrée, multiples sorties, dont la taille ne dépasse pas le budget d'exécution disponible.

Ces deux techniques se concentrent sur un budget temporel, mais d'autres techniques préfèrent un budget énergétique, comme Yarahmadi et Rohou [YR20], qui proposent la séparation du programme en régions simple entrée, simple sortie (*SESE* en anglais) dont la taille ne dépasse pas le budget énergétique du capteur. Là aussi, des points de sauvegarde sont placés entre les différentes régions SESE.

Enfin, Choi *et al.*, [CKL+22] ont récemment proposé d'effectuer un placement de points de sauvegarde pire cas avec un budget temporel, et de garantir qu'aucune perte d'alimentation ne puisse advenir entre deux points de sauvegarde, en imposant le remplissage complet du condensateur à chaque point de sauvegarde. De cette manière, PFI/Rockclimb permet de garantir la progression du programme, tout en empêchant les anomalies mémoires qui pourraient advenir à la suite de la réexécution d'une portion de code.

Ces différentes heuristiques de placement de points de sauvegarde permettent aux programmes intermittents de pouvoir s'exécuter avec très peu d'énergie récoltée, mais imposent naturellement un surcoût. Dans une optique de diminuer ce surcoût, certaines recherches ont cherché à réduire la taille des sauvegardes

2.2.2 Réduction de la taille des sauvegardes

Un des grands axes de recherches dans le checkpointing statique concerne la réduction de la taille des sauvegardes. En effet, la taille des sauvegardes a un impact sur le surcoût

de la sauvegarde, mais aussi sur celui de la restauration de l'état du programme.

Pour réduire la quantité de données volatiles sauvegardées, il existe deux approches : identifier et ne sauvegarder que les données utiles dans la suite du programme, c'est-à-dire vivantes au moment d'un point de sauvegarde ou réduire la quantité de données en mémoire volatile, en allouant certaines variables en mémoire non volatile.

Efficacité des sauvegardes

La technique la plus basique pour sauvegarder l'état des données volatiles consiste à copier tout le contenu des registres et de la mémoire volatile dans la mémoire non volatile. Cela reste très peu efficace, mais pas invraisemblable, car les petits microcontrôleurs sont équipés de peu de mémoire (2Ko pour le msp430fr5969). Seulement, il est apparu très rapidement plus efficace de sauvegarder uniquement les portions de la mémoire volatile allouées : la pile (`.stack`), les données en écriture (`.data`) et le tas, si présent. Malgré cette amélioration, la sauvegarde n'est pas aussi optimisée que possible, car une grande partie des données peut ne pas avoir été modifiée, et n'a donc pas besoin d'être sauvegardée.

Pour permettre une sauvegarde plus optimisée, Bhatti et Mottola [BM16] ont proposé deux stratégies, nommées Heap-Tracker et Copy If Change. La première propose une sauvegarde du tas à la granularité des blocs mémoire alloués. De cette manière, uniquement les portions du tas utilisées sont sauvegardées. La seconde technique, Copy If Change, ne sauvegarde que les données qui ont changé, en comparant le contenu de la mémoire non volatile et celui de la mémoire volatile. Cette technique nécessite donc plus de lectures mémoire et est à destination des microcontrôleurs équipés de mémoire FLASH, la lecture en FLASH étant bien moins consommatrice que l'écriture.

Avec l'essor des mémoires non volatiles efficaces, la différence de coût énergétique entre une lecture et une écriture s'est amoindrie, c'est pourquoi Ahmed *et al.*, [ABA+19] ont proposé une amélioration de la technique Copy If Change, nommée DICE. Cette technique repose sur le compilateur pour déterminer, pour chaque point de sauvegarde, quelles sont les données volatiles vivantes, et donc à sauvegarder. Ensuite, DICE insère des instructions après les accès aux variables en VM pour suivre, à l'exécution, si la valeur d'une variable a été modifiée et doit être sauvegardée.

Toujours du côté de la compilation, Li *et al.*, [LZH+15], tirent parti de l'analyse de durée de vie des variables pour identifier des espaces inutilisés de la pile, car contenant des variables mortes. Ils utilisent ensuite ces espaces pour allouer les nouvelles variables de la pile, réduisant ainsi la taille de cette dernière. En réduisant la taille de la pile, la

taille de la sauvegarde décroît.

Enfin, les tableaux représentent de grandes quantités de données à sauvegarder, et appliquer les techniques présentées précédemment n'est pas toujours possible, car les analyses de durée de vie s'effectuent à la granularité de la variable, les tableaux et structures étant considérés comme des variables. Cependant, Kim *et al.*, [LZH+15] analysent les dépendances de données entre les différents éléments du tableau pour identifier les cellules vivantes et mortes d'un tableau. De cette manière, ils parviennent à réduire grandement la quantité de données sauvegardées.

Cependant, si le schéma d'accès au tableau est complexe, il devient impossible de prédire la durée de vie des éléments qui composent ce dernier. Dans ce cas, une solution consiste à allouer le tableau en mémoire non volatile. La prochaine section pose la question de l'allocation mémoire des données du programme : VM ou NVM ?

Mémoire de travail mixte VM/NVM

La différence de consommation énergétique entre les mémoires volatiles et non volatiles pose la question de l'allocation mémoire des variables. Faut-il allouer les variables en VM, dont l'accès est rapide et peu coûteux, ou en NVM pour éviter les surcoûts de sauvegarde ?

Jayamukar *et al.*, [JRR14] se sont intéressés à la question, à la granularité des fonctions. Les auteurs implémentent un mécanisme de sauvegarde des données volatiles à chaque entrée/sortie de fonction. Ensuite, pour une fonction donnée, ils évaluent la pertinence d'allouer les sections du programme (`.stack`, `.data`...) en mémoire volatile ou non volatile. Pour cela, ils mesurent l'énergie consommée par l'exécution de la fonction en fonction de chaque combinaison d'allocation mémoire. Une fois que la combinaison la plus efficace en énergie est identifiée (par exemple `.stack` en VM et `.data` en NVM), cette allocation mémoire est appliquée à chaque prochaine exécution de la fonction. De cette manière, les auteurs tirent parti de la mémoire non volatile lorsque c'est bénéfique. Les auteurs s'assurent de l'absence d'anomalies due à l'utilisation de la NVM en s'assurant que suffisamment d'énergie a été récoltée avant de reprendre l'exécution depuis un point de sauvegarde. En effet, de cette manière, aucune réexécution du code n'est nécessaire, supprimant les effets des codes non idempotents.

Même si elle permet des économies d'énergie, cette technique ne considère qu'une allocation mémoire gros grain (à l'échelle des sections). Or, à l'intérieur d'une même section, plusieurs variables peuvent avoir des schémas d'accès radicalement différents, il est donc intéressant d'étudier cette allocation mémoire à l'échelle des variables directement.

Pour cela, les auteurs de Alfred [MM21] proposent un mécanisme de sauvegarde anticipée et de restauration décalée. En temps normal, Alfred utilise la mémoire non volatile comme mémoire de travail. Seulement, il ne sauvegarde pas les données en VM vers la NVM lors d'un point de sauvegarde, mais plutôt lors du dernier accès en écriture à la variable. Usuellement, lors d'un accès mémoire, le contenu de la variable est stocké dans un registre, puis copié en mémoire volatile (`Registre→ VM`). Plus tard, lors d'une sauvegarde, la donnée en mémoire volatile est copiée en mémoire non volatile (`VM→NVM`). Pour diminuer la quantité d'opérations mémoire nécessaire, Alfred vient modifier un programme à la compilation, pour rediriger le dernier accès en écriture à une variable pour qu'il cible la NVM au lieu de la VM (`Registre→NVM`), supprimant ainsi la nécessité de sauvegarder cette variable lors d'un point de sauvegarde. Similairement, il modifie le premier accès en lecture à une variable après un point de sauvegarde pour qu'il cible la NVM, à la place de la VM. Ainsi, il économise le coût d'une copie de la NVM vers la VM.

Après le dernier accès en écriture à une variable, le contenu de cette variable réside en NVM. Cependant, les accès NVM étant coûteux, Alfred propose d'effectuer une copie de la variable en VM si l'économie d'énergie réalisée due aux accès à la VM surpasse le coût de copie. De ce point de vue là, Alfred propose une forme d'utilisation d'une mémoire de travail mixte VM/NVM.

2.2.3 Adaptation à l'exécution

Les améliorations présentées jusqu'ici s'appliquent à la compilation, mais ne tiennent pas compte des conditions de récolte d'énergie. Or, cette adaptabilité permettrait une amélioration notable pour de tels systèmes.

Une première solution pour s'adapter à l'énergie disponible consiste à vérifier, à chaque point de sauvegarde, l'énergie contenue dans le condensateur [RSF11]. Si cette énergie est supérieure à un seuil prédéfini, alors on considère qu'il est possible de sauter le point de sauvegarde et de continuer l'exécution du programme. De cette manière, les sauvegardes sont effectuées uniquement lorsque l'énergie vient à manquer, similairement à du checkpointing dynamique. Seulement, contrairement au checkpointing dynamique, il n'est pas nécessaire d'effectuer un suivi constant de la tension aux bornes du condensateur, limitant le surcoût d'un tel suivi.

Pour les capteurs qui ne seraient pas équipés d'un ADC pour mesurer la tension aux bornes du condensateur, les auteurs de Chinchilla proposent une solution similaire, basée sur un compte à rebours. Chinchilla [ML18] place des points de sauvegarde entre

chaque bloc de base à la compilation, et lance un compte à rebours à chaque redémarrage du capteur. Tant que ce compte à rebours n'est pas écoulé, les points de sauvegarde rencontrés sont sautés. Lorsque le compte à rebours est atteint, le prochain point de sauvegarde donnera lieu à une sauvegarde de l'état du programme. Pour déterminer la valeur du compte à rebours, Chinchilla cherche expérimentalement la valeur appropriée : si une perte d'alimentation intervient avant la fin du compte à rebours, alors ce dernier est trop long et doit être diminué (de moitié dans la contribution). Dans le cas inverse, Chinchilla augmente petit à petit ce compte à rebours. De cette manière, Chinchilla diminue le surcoût de la sauvegarde et s'adapte aux conditions de récolte d'énergie.

Idéalement, l'adaptation de l'exécution d'un programme intermittent doit correspondre aux besoins applicatifs auxquels réponds le programme. Sur ce sujet, Kortbeek *et al.*, [KYB+20] ont fait le constat que les pertes d'alimentation fréquentes, et potentiellement longues, peuvent ajouter une latence entre la récolte d'une donnée, son traitement, et son envoi, rendant la donnée obsolète. Pour éviter de dépenser inutilement de l'énergie pour le traitement de ces données obsolètes, ils proposent un outil nommé TICS qui permet de définir une « durée de vie » pour les données récoltées. Les données sélectionnées sont automatiquement horodatées et leur traitement est conditionné par leur « date de péremption », définie manuellement lors de la conception du système. De cette manière, l'exécution du programme s'adapte aux conditions de récolte, en ne traitant pas les données obsolètes.

Objectifs de la thèse. Dans cette thèse, nous nous proposons de répondre à la question du placement des points de sauvegarde (*Où sauvegarder ?*) et à celle de la réduction de la taille des données sauvegardées (*Quoi sauvegarder ?*) avec une première contribution nommée Schematic, présentée Chapitre 3. Plus précisément, Schematic propose d'allier le placement automatique de points de sauvegarde basé sur le budget énergétique à une allocation mémoire à la granularité de la variable pour tirer parti de la mémoire mixte VM/NVM.

Ensuite, nous présenterons EarlyBird, une technique orthogonale aux axes présentés précédemment et qui vise à améliorer les performances des systèmes basés sur du checkpointing statique en modifiant leur tension de réveil, c'est-à-dire la tension au-delà de laquelle l'exécution du programme reprend. Cette technique est présentée dans le chapitre 4.

Chapitre 3

Schematic : Placement de points de sauvegarde et allocation mémoire simultanés

Où sauvegarder, et quoi sauvegarder ?

Dans ce chapitre, nous présentons Schematic, un outil de compilation pour le placement de points de sauvegarde et le choix d'une allocation mémoire pour les variables d'un programme intermittent. À notre connaissance, Schematic est la première technique à résoudre le problème du placement de point de sauvegarde et de l'allocation mémoire de manière simultanée.

En premier lieu, nous motivons dans la section 3.1 les raisons pour lesquelles, selon nous, résoudre ces deux problèmes de manière simultanée est essentiel. Ensuite, nous décrivons le principe de fonctionnement de Schematic sur un programme simple section 3.2, puis nous décrivons comment Schematic gère les boucles et les fonctions section 3.3. Enfin, nous procédons à une évaluation de Schematic section 3.5 et concluons section 3.6.

3.1 Motivation pour un placement de checkpoint et une allocation mémoire simultanée

Pour mieux comprendre les défis liés au placement des points de sauvegarde et à l'allocation mémoire des variables, étudions l'exemple simple, mais néanmoins réaliste du code extrait 5. Ce programme calcule la somme des éléments d'un tableau (variable `array`) et la stocke dans la variable scalaire `sum`. La variable `sum` est ensuite utilisée comme paramètre dans un appel à la fonction `f`. Pour les besoins de la démonstration, nous supposons que le compilateur n'optimise pas les variables en les promouvant en

registre, et donc que les accès aux variables ciblent la mémoire. Considérons un système de checkpointing statique qui sauvegarde toutes les données volatiles (variables et registres) dans la mémoire non volatile à des points de sauvegarde prédéterminés (par le biais d'un appel à la fonction `checkpoint`).

```
1  int sum = 0;
2  checkpoint();
3  for(int i=0; i<SIZE; i++)
4      sum += array[i];
5  /* ..... */
6  checkpoint();
7  class = f(sum);
8  checkpoint();
```

Extrait 5 – Exemple de code source instrumenté pour l'exécution intermittente

3.1.1 Motivation pour une allocation mémoire économe en énergie

Comme présenté précédemment, les microcontrôleurs utilisés dans le calcul intermittent disposent de deux types de mémoire : une mémoire volatile (*VM*) rapide, petite et économe en énergie, et une mémoire non volatile (*NVM*) plus importante.

L'efficacité énergétique de la VM suggèrerait de stocker toutes les données dans la mémoire volatile pour économiser de l'énergie. En effet, même avec les technologies de mémoire non volatile émergentes efficaces telles que la mémoire RAM ferromagnétique (FRAM), les accès en NVM consomment encore jusqu'à 2,47 fois plus que les accès en mémoire volatile [Ins12].

Cependant, les données dans la mémoire volatile sont perdues en cas de perte d'alimentation et doivent donc être sauvegardées dans la NVM, ce qui entraîne un surcoût. En outre, la taille de la VM est limitée et généralement petite (quelques ko sur la plupart des plateformes). Stocker les données en NVM permet alors d'éviter le surcoût de sauvegarde, tout en respectant les contraintes de taille mémoire.

Prenons notre exemple extrait 5. Au cours de la première phase du programme (lignes 3–4), la variable `sum` est fréquemment utilisée. De ce fait, le gain énergétique de stocker `sum` dans la VM compense le coût de son chargement depuis la NVM au démarrage et de sa sauvegarde dans la NVM. L'allocation de `sum` en VM est la meilleure option dans ce cas.

Au cours de la deuxième phase du programme (ligne 7), supposée se dérouler beaucoup plus tard, il est cependant plus approprié de conserver `sum` dans la NVM, car son chargement dans la VM consommerait plus d'énergie que l'énergie gagnée pour l'unique accès à `sum`.

Alors même que le choix d'une allocation mémoire est complexe et fastidieux, ce simple exemple nous montre que, même en étant expert, un développeur ne pourra pas exploiter pleinement les opportunités offertes par les architectures hybrides VM/NVM. En effet, un développeur sélectionnera une allocation mémoire pour chaque variable pour l'ensemble du programme. Or, cet exemple nous montre que pour une même variable, le bénéfice du choix d'une allocation mémoire dépend de la phase de programme. Une allocation mémoire efficace devra donc évoluer au fur et à mesure de l'exécution du programme. Pour ce faire, le support du compilateur est indispensable.

3.1.2 Motivation pour un placement de points de sauvegarde et un choix d'une allocation mémoire simultanés

Le processus de prise de décision concernant l'emplacement des points de sauvegarde est intrinsèquement lié à l'allocation mémoire. Par exemple, il peut arriver dans l'extrait 5 qu'il n'y ait pas assez d'énergie pour exécuter le code entre les lignes 2 et 6 si la variable `sum` est allouée en NVM, alors que cela devient possible lorsque `sum` est stockée dans la VM. Une allocation mémoire à la granularité de la variable permet ainsi d'offrir de nouvelles opportunités de placement de checkpoints qu'une allocation gros grain (tout en VM ou tout en NVM) ne permet pas.

Inversement, la sélection d'une allocation mémoire dépend de l'emplacement des points de sauvegarde. Par exemple, si un point de sauvegarde est placé dans le corps de la boucle (ligne 4), il n'y a plus d'avantage à allouer `sum` en VM car `sum` devrait être sauvegardée à chaque itération. En outre, les points de sauvegarde représentent des endroits naturels pour modifier l'allocation mémoire des variables, puisque la sauvegarde copie les variables de la VM vers la mémoire NVM et masque donc les coûts énergétiques de migration des variables.

Il apparait clairement que les problèmes d'allocation mémoire et de placement de points de sauvegarde sont interdépendants et doivent être résolus conjointement. Seulement, laisser le choix de l'emplacement des points de sauvegarde et l'allocation mémoire des variables sous la responsabilité exclusive des développeurs est source d'erreurs et peut

donner lieu à des performances sous-optimales. La complexité de la résolution de ces deux problèmes de manière conjointe appelle donc à l'utilisation d'une solution automatisée. Bien que des recherches se soient concentrées individuellement sur le placement des points de sauvegarde ou sur l'allocation mémoire, à notre connaissance, aucune étude n'a à ce jour tenté de traiter ces deux problèmes simultanément.

Dans ce chapitre, nous proposons Schematic pour répondre à ce besoin.

3.1.3 Notre contribution : Schematic

Schematic [1] automatise le placement de points de sauvegarde et l'allocation mémoire des variables d'un programme dans les architectures hybrides VM/NVM. L'objectif de Schematic est de minimiser la consommation d'énergie du programme tout en garantissant la progression de ce dernier. À la compilation, Schematic insère des points de sauvegarde et décide de l'allocation mémoire des variables en temps polynomial. Au moment de l'exécution, lorsqu'un point de sauvegarde est atteint, les données volatiles sont sauvegardées dans la NVM et le système attend que le condensateur soit entièrement rempli avant de reprendre l'exécution.

Schematic s'appuie sur une approche consciente de l'énergie : on suppose que la consommation d'énergie dans le pire cas (en anglais *worst case energy consumption*, abrégé *WCEC*) de toute activité dans le système est connue et on tire parti de cette connaissance pour sélectionner les points de sauvegarde, de sorte que le code entre deux points de sauvegarde puisse toujours être exécuté avec un condensateur plein. Par conséquent, Schematic garantit que tout programme finira par se terminer. De plus, en faisant en sorte d'attendre le remplissage du condensateur à chaque point de sauvegarde, Schematic garantit l'absence de perte d'alimentation entre deux points de sauvegarde. Avec cette approche, Schematic garantit l'absence de réexécutions, évitant de dépenser de l'énergie inutilement.

Énoncé du problème

Plus formellement, le problème de recherche abordé par Schematic est le suivant.

Hypothèses et données d'entrée :

— Schematic suppose une architecture hybride avec NVM et VM, comme le montre

1. pour *Simultaneous Checkpoint Placement and Memory Allocation Tailored for Intermittent Computing*

la figure 3.1. La NVM est supposée être suffisamment grande pour stocker l'ensemble du code et des données de l'application. La VM est supposée être plus petite, avec une taille S_{VM}.

— La plateforme est équipée d'un condensateur avec un stockage d'énergie limité E_B.

— Un modèle de consommation d'énergie pire cas, sûr et précis, est fourni en entrée à SCHEMATIC.

Garanties :

— *Progression.* Le placement des points de sauvegarde dans SCHEMATIC garantit que tout calcul finit par se terminer : l'exécution ne restera pas bloquée dans des réexécutions sans fin en raison de pertes d'alimentation répétitives.

— *Absence d'anomalies mémoire.* SCHEMATIC évite les réexécutions de code en s'assurant que l'énergie requise pour exécuter le programme entre deux points de sauvegarde successifs est toujours inférieure à E_B, et que le condensateur est rempli à chaque point de sauvegarde. Par conséquent, les incohérences entre VM et NVM (anomalies mémoire) sont évitées par conception.

— *Minimisation de l'énergie consommée.* Le placement des points de sauvegarde et l'allocation mémoire sont conçus pour minimiser la consommation d'énergie sur les chemins les plus fréquemment exécutés.

— *Utilisation efficace de la VM.* SCHEMATIC alloue des variables dans la VM pour réduire la consommation d'énergie et garantit de manière transparente qu'à tout moment, le volume de données alloué en VM est inférieur ou égal à S_{VM}.

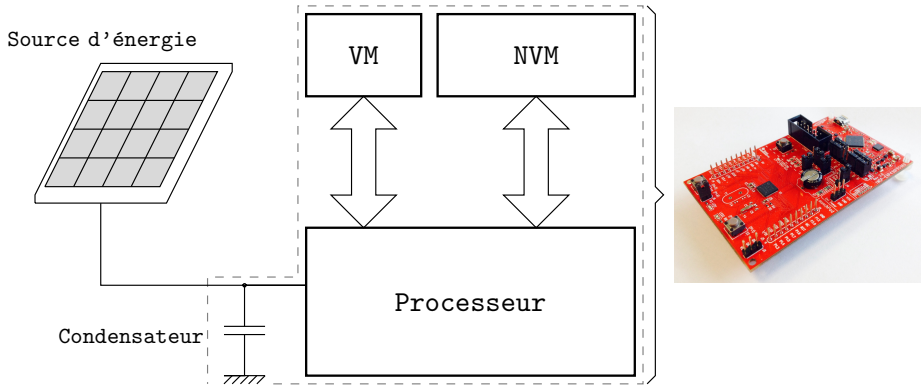


FIGURE 3.1 – Architecture considérée

La suite de ce chapitre est consacrée à la description de SCHEMATIC (section 3.2 à 3.4), ainsi qu'à son évaluation expérimentale (section 3.5).

3.2 Vue d'ensemble de Schematic

Schematic est une technique de compilation qui détermine à la fois (i) les emplacements où la sauvegarde est effectuée (points de sauvegarde) et (ii) l'allocation des variables (VM ou NVM) entre les points de sauvegarde.

Lors de l'exécution, lorsqu'un point de sauvegarde est atteint, les données volatiles (registres du processeur, données allouées dans la VM) sont sauvegardées dans la mémoire non volatile (voir la figure 3.2, légende ▨). Le système reste ensuite en veille jusqu'à ce que le condensateur soit complètement chargé (□). Pendant cette période de veille, le système est en sommeil profond et se réveille régulièrement pour mesurer la tension aux bornes du condensateur. Si une perte d'alimentation survient pendant la période de veille, le système se remet en veille au redémarrage. Lorsque les mesures de tension indiquent que le condensateur est complètement chargé, l'état du programme est restauré (▩), et l'exécution reprend (▧). La courbe supérieure de la figure 3.2 représente l'évolution de l'état de charge du condensateur au cours des différentes phases (exécution du programme, veille...)

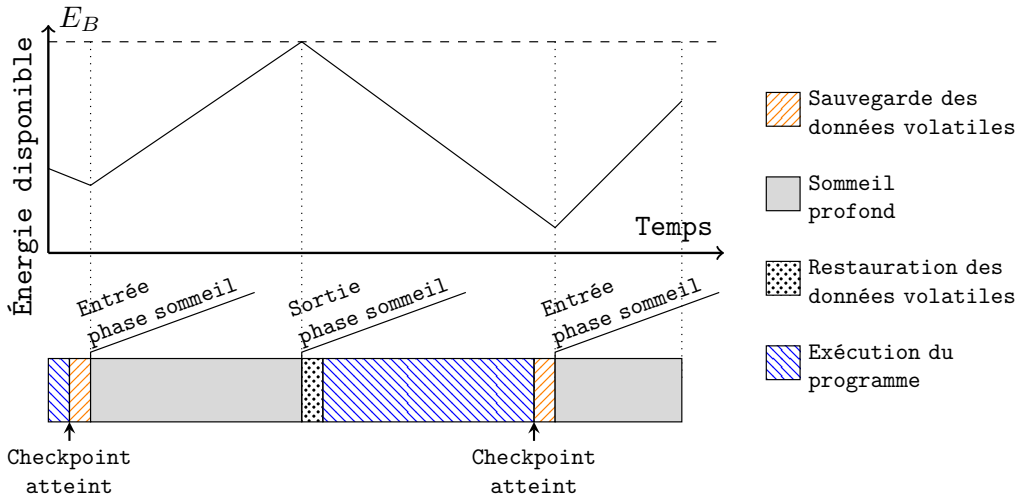


Figure 3.2 – Stratégie de checkpointing de Schematic

Schematic détermine l'allocation mémoire des variables (VM ou NVM) à la compilation. L'allocation d'une variable peut changer à l'exécution, mais les points où l'allocation peut changer (emplacements des points de sauvegarde) sont définis à la compilation. L'allocation mémoire est effectuée à la granularité des variables dans le code source (scalaires, structures, tableaux considérés dans leur ensemble).

SCHEMATIC analyse et modifie un programme en utilisant sa représentation en graphe de flot de contrôle (*control flow graph* en anglais, abrégé CFG). L'exécution d'un programme peut donner lieu à un certain nombre de chemins d'exécution au moment de l'exécution, un chemin étant défini comme une séquence ordonnée de blocs de base, commençant au point d'entrée du CFG et se terminant à l'un de ses points de sortie.

SCHEMATIC positionne les points de sauvegarde et sélectionne l'allocation des variables avec l'objectif de minimiser l'énergie requise pour exécuter les chemins les plus fréquemment exécutés. Le déroulement de l'algorithme est le suivant :

1. Sélectionner un chemin à analyser.

2. Décider de l'emplacement des points de sauvegarde et de l'allocation mémoire le long de ce chemin, de sorte que les calculs entre les points de sauvegarde, compte tenu de l'allocation des variables, puissent être exécutés avec le budget énergétique E_B.

3. Modifier le code du programme pour insérer des instructions de sauvegarde/restauration aux points de sauvegarde sélectionnés et changer les cibles des accès à la mémoire (VM ou NVM) en fonction de l'allocation mémoire sélectionnée.

4. Répéter le processus jusqu'à ce que tous les blocs de base du CFG aient été analysés. Les décisions prises par SCHEMATIC le long d'un chemin sont définitives. Lors de l'examen d'un nouveau chemin, des contraintes relatives aux allocations mémoire et au placement de points de sauvegarde sont hérités des chemins déjà analysés.

Les emplacements considérés par SCHEMATIC pour le placement des points de sauvegarde sont les arcs du CFG. Au cours de l'analyse du programme, certains des points de sauvegarde potentiels deviennent *activés*, ce qui signifie que des instructions sont insérées dans le code pour sauvegarder et restaurer les données volatiles. Les autres points de sauvegarde potentiels sont *désactivés*. Ils n'entraînent pas d'instrumentation du code, et n'ont donc aucun surcoût à l'exécution.

La section 3.2.1 décrit d'abord le placement des points de sauvegarde et l'allocation mémoire dans un programme simple. La section 3.3 se concentre ensuite sur la façon dont SCHEMATIC gère les appels de fonction et les boucles. Enfin, la section 3.4 analyse la complexité de l'algorithme de SCHEMATIC.

3.2.1 Analyse d'un chemin à l'aide du graphe des points de sauvegarde atteignables

Dans cette section, nous décrivons comment Schematic analyse un chemin sur un exemple de CFG représenté dans la figure 3.3.a. Les emplacements considérés par Schematic pour le placement des points de sauvegarde sont les arcs AB, AC, BD et CD, ce qui donne quatre points de sauvegarde potentiels (c_{AB}, c_{AC}, c_{BD} et c_{CD})[2]. Considérons que le chemin (A, B, D) est analysé en premier. La figure 3.3.b affiche des informations sur ce chemin avant l'analyse. Le but de l'analyse est de définir l'allocation mémoire de `sum` sur ce chemin, ainsi que l'état des points de sauvegarde c_{AB} et c_{BD} : *activé* (un point de sauvegarde sera inséré à cet emplacement) ou *désactivé*.

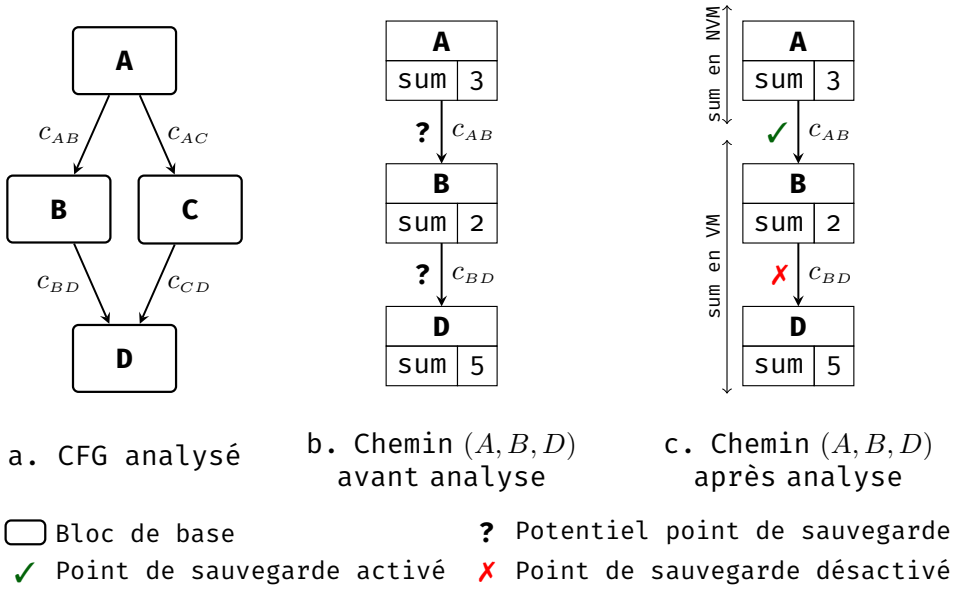


Figure 3.3 – Analyse du chemin (A, B, D) pour le placement des points de sauvegarde et la sélection de l'allocation mémoire. La partie a. décrit le CFG analysé. La partie b. décrit l'état du chemin (A,B,D) avant l'analyse : les emplacements potentiels des points de sauvegarde (c_{AB}, c_{BD}) sont marqués par « ? » parce qu'il n'est pas encore décidé si un point de sauvegarde sera inséré à ces emplacements ; pour chaque bloc de base, la figure décrit les variables accédées (ici seulement une variable nommée `sum`) et le nombre d'accès (accès en lecture et en écriture agrégés pour des raisons de brièveté). La partie c. illustre le résultat de l'analyse : un point de sauvegarde est inséré sur l'arc AB, la variable *sum* est allouée dans la NVM avant ce point de sauvegarde et dans la VM après.

2. Sans perte de généralité, nous supposons que les blocs de base ont besoin de moins de E_B d'énergie pour s'exécuter (ceux qui ont besoin de plus de E_B sont découpés pour entrer dans le budget énergétique).

Pour effectuer cette analyse, nous introduisons le concept de *Graphe des Points de Sauvegarde Atteignables* (*Reachable Checkpoint Graph* en anglais, abrégé RCG). Un RCG est construit à partir d'un chemin du CFG donné et capture l'énergie consommée entre les points de sauvegarde potentiels. Un nœud du RCG représente un point de sauvegarde potentiel c_x. Un arc (c_1, c_2) dans le RCG relie deux points de sauvegarde potentiels c_1 et c_2. Un arc (c_1, c_2) existe s'il est possible d'atteindre c_2 depuis c_1 avec le budget énergétique E_B. Chaque arc est associé à un coût énergétique et à une allocation mémoire. Le coût correspond à l'énergie nécessaire pour atteindre c_2 à partir de c_1 en considérant l'allocation mémoire de chaque variable (VM ou NVM) qui minimise la consommation d'énergie. Ce coût énergétique comprend :

— L'énergie requise pour restaurer les données volatiles à l'emplacement c_1.
— L'énergie nécessaire pour exécuter les blocs de base le long du chemin analysé avec l'allocation mémoire sélectionnée. La sélection d'une allocation mémoire est détaillée dans la section suivante.
— L'énergie nécessaire pour sauvegarder les données volatiles dans la NVM à l'emplacement c_2.

L'absence d'un arc entre c_1 et c_2 signifie qu'il n'y a pas d'allocation mémoire permettant d'atteindre c_2 à partir de c_1 avec un budget énergétique inférieur à E_B.

Deux nœuds virtuels *start* et *end* représentent le début et la fin du chemin analysé. Chaque chemin de *start* à *end* correspond à un placement valide de points de sauvegarde pour le chemin analysé, avec les allocations mémoire correspondantes. Le chemin le plus court entre *start* et *end* représente donc le placement de points de sauvegarde et l'allocation mémoire qui minimisent l'énergie consommée le long du chemin analysé. Tous les points de sauvegarde situés le long du chemin le plus court sont activés, tandis que les autres points de sauvegarde du RCG sont désactivés. Les allocations mémoire sélectionnées sont celles qui apparaissent sur les arcs du chemin le plus court.

La figure 3.4 représente le RCG pour le chemin (A, B, D) du CFG. Les points de sauvegarde potentiels pour ce chemin sont c_{AB} et c_{BD}. Nous considérons un budget énergétique de la plate-forme E_B de 20 *unités d'énergie*. Le RCG contient deux chemins possibles entre *start* et *end*. Nous pouvons soit choisir c_{AB} et c_{BD} comme points de sauvegarde – chemin (*start*, c_{AB}, c_{BD}, *end*) –, ou ne choisir que c_{AB} – chemin (*start*, c_{AB}, *end*).

Le chemin le plus court dans ce cas est (*start*, c_{AB}, *end*). Nous n'insérerons donc que des instructions pour sauvegarder/restaurer des données volatiles à l'emplacement du point de sauvegarde c_{AB} (entre les blocs de base A et B). Le coût total de l'exécution de ce

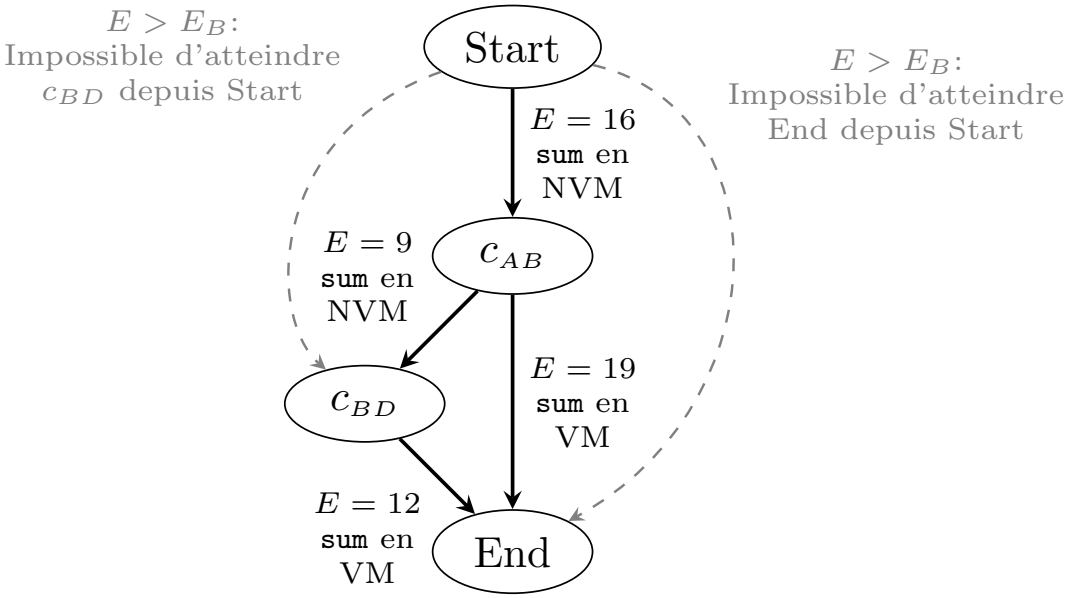


Figure 3.4 – Graphe des points de sauvegarde atteignables (RCG) pour le chemin (A, B, D) pour un budget énergétique E_B de 20 *unités d'énergie*. Chaque arc du RCG indique l'énergie consommée (étiquetée E) pour aller d'un point de sauvegarde potentiel à un autre, pour une allocation donnée des variables (ici la variable *sum*). Les lignes pointillées indiquent les arcs absents du RCG car E dépasse le budget E_B.

chemin est de 35 unités d'énergie. La figure 3.3.c illustre le résultat de l'analyse.

3.2.2 Sélection d'une allocation mémoire

Afin de construire le RCG pour un chemin donné du CFG, il est nécessaire de connaitre l'énergie nécessaire pour aller d'un point de sauvegarde potentiel c_1 à un autre point de sauvegarde potentiel c_2. Ce coût dépend de l'allocation des variables du programme dans l'intervalle $[c_1, c_2]$ sur ce chemin.

La sélection de l'allocation mémoire de chaque variable est basée sur une fonction de coût, qui définit le gain obtenu en plaçant une variable v dans la VM par rapport à la NVM.

Il y a un gain d'énergie en effectuant un accès à la VM plutôt qu'à la NVM : ΔE_W pour un accès en écriture, ΔE_R pour un accès en lecture. En revanche, il y a un coût énergétique de $E_{\text{save/restore}}$ lors de l'allocation d'une variable v dans la VM, en raison du coût de restauration de la variable au point de sauvegarde c_1 et de sauvegarde dans la NVM au point de sauvegarde c_2.

Le gain d'énergie lié à l'allocation d'une variable v en VM est donc de :

$$gain_v = \Delta E_W \times \mathrm{n}_W + \Delta E_R \times \mathrm{n}_R - E_{\text{save/restore}} \tag{3.1}$$

Avec n_W et n_R les nombres d'accès en écriture et en lecture à v dans l'intervalle. Il est ainsi nécessaire d'effectuer plusieurs accès à la variable pour compenser le coût de sauvegarde de cette dernière.

Le calcul de ce gain pour toutes les variables donne non seulement des informations sur le bénéfice qu'il y a à placer une variable dans la VM, mais nous permet également de donner la priorité aux variables à stocker dans la VM en cas d'espace mémoire limité.

Afin de maximiser le gain total pour une taille mémoire limitée, SCHEMATIC trie les variables en fonction du rapport $\frac{\text{gain}}{\text{taille}}$, de manière décroissante. Ainsi, si plusieurs variables partagent le même gain, les plus petites sont allouées en priorité, ce qui permet de placer plus de variables dans la VM. Les variables sont allouées dans la VM de manière contiguë, jusqu'à ce que la liste des variables ayant des gains positifs soit épuisée ou que la VM soit pleine.

Il est à noter que chaque variable v a une adresse unique dans la NVM, attribuée par le compilateur. Lorsque SCHEMATIC alloue une variable dans la VM dans un intervalle $[c_1, c_2]$, une adresse en VM est attribuée à la variable, qui ne change pas dans l'intervalle. Néanmoins, cette variable peut se voir attribuer une adresse différente en VM sur un autre intervalle.

Prise en compte de la durée de vie des variables pour une sauvegarde optimisée. SCHEMATIC exploite les informations sur la durée de vie des variables pour réduire le coût de la sauvegarde. Si une variable en VM v n'est pas utilisée après un point de sauvegarde $c2$, il n'est pas nécessaire de la sauvegarder. De même, si le premier accès à v après un point de sauvegarde c_1 est un accès en écriture, nous ne restaurons pas la variable, car sa valeur sera écrasée.

Le coût $E_{\text{save/restore}}$ (présenté dans l'équation 3.1) lié à la sauvegarde et la restauration d'une variable v dépend de sa taille, mais aussi de sa durée de vie, et se calcule comme suit :

$$E_{\text{save/restore}} = E_{\text{restore}} \times \text{live\_}c_1 + E_{\text{save}} \times \text{live\_}c_2 \tag{3.2}$$

Avec E_{restore} le coût de restauration de v et E_{save} le coût de sauvegarde de v. live_X est égal à 1 si la variable v est vivante à l'emplacement du point de sauvegarde X, et est égal à 0 dans le cas contraire.

3.2.3 Exploration des chemins du CFG et couverture de code

Afin de réduire l'énergie consommée en priorité pour les chemins les plus courants, les différents chemins du CFG sont analysés de manière itérative, par fréquence décroissante. Les décisions prises par Schematic le long d'un chemin sont définitives. La priorisation des chemins est réalisée par une instrumentation extensive du code afin de recueillir des traces d'exécution (séquences de blocs de base exécutés). Les traces sont triées par fonction. Il peut arriver qu'en dépit d'une instrumentation poussée, certaines parties du code ne soient jamais exécutées. Des chemins, formés à partir de ces portions de code, sont extraits du CFG et sont analysés à la fin de l'algorithme pour assurer une couverture complète du code.

Lors de l'analyse d'un nouveau chemin p, seuls les segments de p dont les blocs de base n'ont pas déjà été analysés sont explorés. L'exploration de ces segments hérite des résultats obtenus pour les chemins déjà analysés, de la manière suivante. Des informations sont attachées aux blocs de base des chemins déjà analysés : l'allocation mémoire pour ce bloc de base, l'énergie restante après son exécution E_{left}, et l'énergie à laisser au bloc de base pour pouvoir atteindre les points de sauvegarde suivants E_{to_leave}.

Lors de la construction du graphe des points de sauvegarde atteignables pour un segment du chemin p contenant des blocs de base déjà analysés, les critères permettant de déterminer si un arc (c_i, c_j) est présent dans le RCG changent légèrement. Plus précisément, les modifications suivantes sont effectuées :

— Lors de l'évaluation des arcs provenant du nœud *start*, le critère pris en compte est E_{left}, plutôt que le budget énergétique E_B.

— De même, pour les arcs dirigés vers le nœud *end*, le critère E_B n'est plus utilisé. À la place, la différence entre le budget énergétique et l'énergie à laisser au bloc de base ($E_B - E_{to\_leave}$) est utilisée.

L'énergie restante et l'énergie à laisser sont recalculées et propagées après chaque chemin analysé. Tout au long de l'analyse, l'énergie restante ne peut que diminuer tandis que l'énergie à laisser ne peut qu'augmenter. Ce faisant, l'analyse des nouveaux chemins s'adapte aux contraintes imposées par les chemins précédemment analysés, garantissant un placement sûr des points de sauvegarde.

Dans notre exemple, le bloc de base A contient l'information que l'énergie restante après son exécution est de 4 ($E_B = 20$, le coût d'exécution de A est de 16). L'énergie à laisser à A pour atteindre le point de sauvegarde c_{AB} est de 16. Cette énergie peut augmenter si aucun point de sauvegarde n'est placé à c_{AC}.

Nous avons décrit dans cette section comment SCHEMATIC effectue le placement de points de sauvegarde ainsi que l'allocation mémoire des variables sur un programme simple. Cependant, nous n'avons pas encore évoqué comment SCHEMATIC traite les appels de fonctions, ni comment SCHEMATIC gère les boucles.

3.3 Gestion des appels de fonction et boucles

Dans cette Section, nous décrivons dans un premier temps comment SCHEMATIC gère les appels de fonctions (Section 3.3.1), puis nous nous penchons sur le sujet des boucles (Section 3.3.2).

3.3.1 Gestion des appels de fonction

Le principal défi posé par les fonctions est qu'elles peuvent être appelées depuis différents endroits du code. Comme le placement des points de sauvegarde et l'allocation mémoire d'une fonction sont décidés à la compilation et sont indépendants du contexte, les décisions prises pour une fonction doivent être valables pour tous les contextes d'appel.

Les fonctions sont analysées en parcourant le graphe des appels de fonctions, dans l'ordre topologique inverse, de sorte que chaque fonction est toujours analysée avant ses fonctions appelantes. Actuellement, SCHEMATIC ne gère que les fonctions non récursives [3].

Une décision (placement de point de sauvegarde, allocation mémoire) prise pour une fonction impose des contraintes à tous ces prédécesseurs dans le graphe d'appel.

L'analyse des fonctions est gloutonne : une fois qu'une décision est prise pour une fonction donnée, elle n'est jamais reconsidérée par la suite.

L'analyse des fonctions feuilles (fonctions sans successeur dans le graphe d'appel) se déroule comme décrit précédemment dans la Section 3.2. A contrario, pour une fonction f_{caller} qui comprend un appel à une fonction f_{callee}, il faut tenir compte des contraintes établies par l'analyse de f_{callee}.

Si f_{callee} n'inclut aucun point de sauvegarde, alors tous ses blocs de base partagent la même allocation mémoire, puisque les changements d'allocation mémoire sont effectués sur les points de sauvegarde. Dans ce cas, nous pouvons traiter l'appel de f_{callee} comme un seul bloc de base dans l'analyse de f_{caller}. En revanche, si f_{callee} comporte un ou plusieurs points de sauvegarde, il peut y avoir des allocations de variables différentes à l'entrée et

3. Nous ne considérons pas cela comme une restriction significative, car la récursion est généralement considérée comme une mauvaise pratique dans le développement de systèmes embarqués.

à la sortie de la fonction. Par conséquent, lors de l'analyse de f_{caller}, nous devons prendre en compte l'allocation mémoire et l'énergie nécessaire à l'exécution de f_{callee} jusqu'au(x) premier(s) point(s) de sauvegarde de f_{callee}, ainsi que l'allocation mémoire et l'énergie restante après l'exécution de f_{callee}. Pour simplifier l'analyse, nous imposons que tous les points de sortie d'une fonction partagent une même allocation mémoire.

3.3.2 Gestion des boucles

Nous traitons les boucles de la même manière que les fonctions. Schematic gère les *boucles naturelles* (*natural loops* en anglais). Les boucles naturelles sont des composantes fortement connexes du CFG avec un seul point d'entrée, appelé *en-tête de boucle* (*loop header* en anglais).

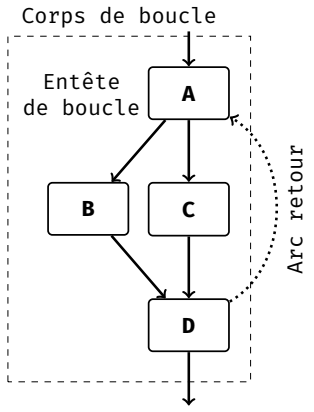


Figure 3.5 – Boucle naturelle

Sans perte de généralité, notre description de la gestion des boucles dans Schematic suppose un seul *arc retour* (*back-edge* en anglais) par boucle. Un *arc retour* est un arc du CFG allant de n'importe quel bloc de base du corps de la boucle vers l'en-tête de la boucle.

Les boucles sont analysées en parcourant l'arbre d'imbrication des boucles des feuilles vers la racine (dans le cas de boucles imbriquées, les boucles internes sont analysées avant les boucles externes).

Une approche simple pour traiter les boucles consisterait à placer un point de sauvegarde sur l'arc retour de la boucle et à s'assurer qu'une itération de la boucle peut être exécutée dans les limites du budget énergétique. Cela garantirait la progression de la boucle, mais entraînerait un surcoût de sauvegarde inutile.

Au lieu de cela, nous estimons le nombre maximum d'itérations de la boucle qui peuvent être exécutées avant qu'il ne soit nécessaire d'effectuer un point de sauvegarde, puis nous mettons en œuvre un mécanisme de sauvegarde conditionnel à l'exécution, basé sur le nombre d'itérations exécutées.

L'analyse des boucles s'effectue en deux étapes, détaillées ci-dessous et illustrées dans l'algorithme 1 :

Étape 1. Analyse d'une itération (ligne 1). Cette première étape traite une itération de la boucle en utilisant l'algorithme présenté Section 3.2. L'algorithme est appliqué au corps de la boucle dont l'arc retour a été supprimé. Le résultat de l'analyse est l'allocation

mémoire des variables accédées dans le corps de la boucle et le placement des points de sauvegarde dans le corps de la boucle. L'algorithme choisira de placer un point de sauvegarde seulement s'il n'y a pas assez d'énergie pour exécuter tout le corps de la boucle.

Étape 2. Analyse de la boucle entière (lignes 2–10). L'objectif de cette étape est de décider si un point de sauvegarde doit être placé sur l'arc retour de la boucle, et combien d'itérations peuvent être effectuées sans sauvegarder les données volatiles. Si l'allocation mémoire de l'en-tête de boucle et celle du bloc de base précédent l'arc retour ne sont pas les mêmes, il est nécessaire de placer un point de sauvegarde afin de modifier les allocations mémoire entre ces deux blocs de base (ligne 2). Dans le cas contraire, il n'est pas nécessaire de modifier l'allocation mémoire à chaque itération. Pour réduire le nombre de sauvegardes, un mécanisme de *sauvegarde conditionnelle* est mis en place : les opérations de sauvegarde/restauration ont lieu toutes les num_{it} itérations de boucle, avec num_{it} le nombre d'itérations de boucle qui peuvent être exécutées avec le budget énergétique de la plate-forme E_B (lignes 5–10). Si num_{it} est supérieur au nombre maximum d'itérations de la boucle, aucun code de point de sauvegarde conditionnel n'est inséré. Le nombre maximal d'itérations des boucles est fourni à l'aide d'annotations dans le code.

Algorithm 1 Analyse de boucles dans SCHEMATIC

▷ **1. Analyse du corps de boucle, sans l'arc retour**
1: AnalyzeCFG(LoopBodyNoBackedge)
▷ **2. Est-il nécessaire de sauvegarder lorsque l'arc retour est pris ?**
2: **if** $header.mem\_alloc \neq latch.mem\_alloc$ **then**
3: $backedge\_chkpt \leftarrow$ yes
4: **return**
5: $E_{loop} \leftarrow header.E_{left} - latch.E_{left}$
6: $\text{num}_{\text{it}} \leftarrow \left\lfloor \frac{E_B}{E_{loop}} \right\rfloor$
7: **if** $\text{num}_{\text{it}} > \text{max}_{\text{it}}$ **then**
8: $backedge\_chkpt \leftarrow$ no
9: **else**
10: $backedge\_chkpt \leftarrow$ every num_{it} iterations

3.4 Considérations de complexité

Dans cette section, nous dérivons la complexité de l'analyse de Schematic, basée sur le nombre d'arcs (E) et de nœuds (V) dans un CFG.

Pour ce faire, nous nous concentrons d'abord sur la complexité de l'analyse d'un chemin, qui implique trois étapes principales, à savoir la construction du RCG, l'identification du chemin le plus court dans le RCG et l'insertion d'instructions de point de sauvegarde dans le code.

La construction du RCG implique l'examen de toutes les combinaisons possibles de points de sauvegarde potentiels (c_i, c_j) où $i < j$. Ce processus peut être accompli en temps polynomial, en particulier $O(\mathrm{E}^2)$.

L'étape suivante, qui consiste à trouver le chemin le plus court dans le RCG, est réalisée à l'aide de l'algorithme du chemin le plus court de Dijkstra. Le nombre de nœuds dans le RCG est limité par $\mathrm{V'} \leq \mathrm{V} + 2$, car nous ajoutons deux nœuds virtuels *start* et *end*. Pour le nombre d'arcs, nous savons que le RCG est un graphe acyclique dirigé, il y a donc au plus $\mathrm{E'} \leq \frac{\mathrm{V'}^2 - \mathrm{V'}}{2}$ arcs dans le RCG. L'algorithme de Dijkstra a une complexité temporelle de $O(\mathrm{E'} + \mathrm{V'} \times log(\mathrm{V'}))$, donc trouver le chemin le plus court dans le RCG a une complexité de $O(\mathrm{V}^2 + Vlog(V)) = O(\mathrm{V}^2)$.

Enfin, l'insertion d'instructions de point de sauvegarde dans le code implique l'examen de chaque arc du chemin analysé et peut être effectuée en temps linéaire, plus précisément $O(\mathrm{E})$. Finalement, la complexité temporelle de l'analyse d'un chemin est de l'ordre de $O(\mathrm{V}^2 + \mathrm{E}^2)$.

Puisque, dans Schematic, un chemin n'est analysé que si l'un de ses blocs de base n'a pas déjà été analysé, il y a au plus un chemin analysé par bloc de base. Par conséquent, il y aura un maximum de V analyses effectuées, conduisant à une complexité polynomiale globale de $O(V \times (\mathrm{V}^2 + \mathrm{E}^2))$.

Empiriquement, le temps d'exécution de l'analyse de Schematic est d'environ 1 min (71 s en moyenne) lorsque appliqué sur les benchmarks considérés dans la Section 3.5.

3.5 Évaluation expérimentale de Schematic

À la différence de nombreux systèmes intermittents, l'allocation mémoire de Schematic permet l'exécution d'applications même lorsque leurs données dépassent la taille de la VM. Cette propriété est évaluée dans la section 3.5.2. Schematic garantit également

la progression du programme, malgré de fréquentes pertes d'alimentation, à l'aide d'un placement automatique des points de sauvegarde. Cette fonctionnalité est évaluée dans la section 3.5.3. La section 3.5.4 compare la consommation d'énergie de SCHEMATIC à celle des techniques apparentées. L'allocation de la mémoire effectuée par SCHEMATIC est évaluée dans la section 3.5.5. Enfin, l'impact de la taille du condensateur sur l'énergie consommée est étudié dans la section 3.5.6. Le dispositif expérimental est d'abord présenté à la section 3.5.1.

3.5.1 Dispositif expérimental

Microcontrôleur considéré et modèle énergétique

Nos expériences ciblent le msp430fr5969 [Ins12], un microcontrôleur basse consommation équipé d'une NVM de 64 Ko de RAM ferromagnétique (FRAM) et d'une VM de 2 Ko de SRAM. Par défaut, une fréquence de fonctionnement de 16 MHz est supposée.

Le modèle de consommation d'énergie pire cas utilisé dans les expériences se concentre sur la consommation d'énergie du processeur. Nous utilisons le même modèle énergétique (tiré de [MM21]) pour toutes les techniques, pour une comparaison équitable. L'énergie dépensée par instruction est calculée à partir du temps d'exécution de l'instruction et du type d'accès à la mémoire (VM ou NVM) (voir la publication originale de Maoli et Mottola [MM21] pour plus de détails). Nous ne prenons pas actuellement en compte la consommation d'énergie des périphériques dans notre modèle, car les benchmarks testés n'utilisent pas de périphériques.

Références

Nous avons comparé SCHEMATIC avec quatre techniques de références : RATCHET, MEMENTOS, ROCKCLIMB et ALFRED. Ces techniques ont été sélectionnées parce qu'elles reposent toutes sur une sélection statique des emplacements des points de sauvegarde, comme le fait SCHEMATIC.

— RATCHET [WH16] est conçu pour les systèmes uniquement équipés de NVM. Pour éviter les incohérences mémoire résultant des réexécutions, RATCHET place des points de sauvegarde de manière à rompre les dépendances écriture-après-lecture (comme l'incrémentation d'une variable). Comme RATCHET n'utilise pas de VM, les registres du processeur sont les seules données volatiles à faire l'objet d'une sauvegarde. Nous utilisons RATCHET comme technique de référence tout-NVM.

— Mementos [RSF11] utilise uniquement la VM comme mémoire de travail. Lorsqu'à l'exécution Mementos atteint un point de sauvegarde, ce dernier décide si une sauvegarde est nécessaire, compte tenu de l'énergie restante dans le condensateur. Pour estimer l'énergie disponible, il mesure la tension du condensateur. Mementos est utilisé comme technique de référence tout-VM, car il utilise la NVM uniquement pour les sauvegardes.

— RockClimb [CKL+22] est un algorithme de placement de points de sauvegarde utilisant uniquement la NVM comme mémoire de travail. La première passe du compilateur de RockClimb place des points de sauvegarde au niveau des en-têtes de boucle et avant les appels de fonction. La deuxième passe est chargée d'insérer des points de sauvegarde supplémentaires, si nécessaire, pour assurer la progression du programme : elle parcourt le CFG du programme et ajoute des points de sauvegarde sur les chemins pour lesquels la consommation d'énergie entre les points de sauvegarde successifs est supérieure à E_B. De plus, comme Schematic, RockClimb attend à chaque point de sauvegarde que le condensateur soit rempli avant de reprendre l'exécution du programme. Nous avons réimplémenté RockClimb et son optimisation *loop-unrolling* (*déroulement de boucle* en français). Cette optimisation déroule les boucles pour éviter de sauvegarder à chaque itération de boucle (nous limitons néanmoins le facteur de déroulement à 10 pour limiter la taille du code).

— Alfred [MM21] est la seule technique de compilation qui, comme Schematic, utilise à la fois la VM et la NVM comme mémoires de travail. Alfred réduit le surcoût du point de sauvegarde, en effectuant une restauration différée des variables (lors de leur première lecture) et une sauvegarde anticipée des variables (lors de leur dernière écriture). Alfred n'impose pas de stratégie de placement des points de sauvegarde et s'appuie plutôt sur les travaux existants pour la sélection statique des points de sauvegarde. Lorsqu'un point de sauvegarde est atteint, seuls les registres du processeur sont sauvegardés dans la NVM, puisque toutes les autres données volatiles ont été sauvegardées précédemment. La VM dans Alfred est utilisée autant que possible, afin de réduire la consommation d'énergie.

Pour Mementos et Alfred, nous avons placé des points de sauvegarde à l'intérieur des boucles, juste avant l'arc retour de ces dernières, comme décrit dans la publication de Mementos [RSF11].

Outils et mise en œuvre

De la même manière que ALFRED, SCHEMATIC opère sur la représentation intermédiaire (*Intermediate Representation*, ou *IR* en anglais) de l'infrastructure de compilation LLVM [LA04] (*version 8*), plus précisément, sur la représentation textuelle de l'IR générée par le compilateur *clang*. Pour permettre une comparaison équitable entre les techniques, toutes ont été réimplémentées dans SCEPTIC [MMA+21 ; Mai21], l'émulateur développé par les auteurs de ALFRED.

SCEPTIC permet l'analyse, la modification et l'émulation de programmes au niveau IR. Il permet un suivi précis de la consommation d'énergie, permettant de comprendre comment l'énergie disponible est utilisée (calcul, accès à la mémoire ...)

SCHEMATIC est implémenté en plusieurs passes de compilation. La première (déjà implémentée dans SCEPTIC) rassemble des informations sur les cibles des instructions *load* et *store*. Ensuite, le CFG du programme, enrichi avec les données recueillies par la première passe, est introduit dans la passe centrale de SCHEMATIC, qui sélectionne le placement des points de sauvegarde et l'allocation de la mémoire comme décrit dans la section 3.2. La dernière passe modifie le programme en définissant la mémoire ciblée par les opérations *load*/*store* en fonction de l'allocation mémoire calculée ainsi qu'en insérant des opérations de sauvegarde/restauration.

Dans l'implémentation actuelle de SCHEMATIC, les variables accédées par des pointeurs sont systématiquement allouées dans la NVM. Ce faisant, nous garantissons que leurs adresses ne changent pas au cours de l'exécution du programme. Nous pensons que cela n'entraîne pas une grande perte de performance, car ces variables sont généralement de grands tableaux que SCHEMATIC allouerait probablement dans la NVM.

L'évaluation du comportement d'exécution de SCHEMATIC, ALFRED, ROCKCLIMB, RATCHET et MEMENTOS repose sur l'émulateur SCEPTIC, qui exécute des programmes au niveau IR, sous une alimentation électrique intermittente. Pour déterminer quand les pertes d'alimentation se produisent de manière simple et reproductible, l'émulateur s'appuie sur des pertes d'alimentation périodiques, grâce à la notion d'*intervalle entre les pertes d'alimentation* (*Time between power failure* ou TBPF en anglais), qui définit l'intervalle de temps en cycles entre les pertes d'alimentation [WH16 ; CKL+22 ; YR21]. L'émulateur SCEPTIC surveille plusieurs métriques du programme au niveau IR et fournit des moyens pour lier ces métriques à des métriques spécifiques à la machine, en particulier la consommation d'énergie du msp430fr5969.

Les traces d'exécution des blocs de base utilisés par SCHEMATIC pour le placement des

points de sauvegarde et l'allocation de la mémoire ont été générées à l'aide de l'émulateur ScEpTIC en exécutant chaque benchmark 1000 fois avec des entrées générées de manière aléatoire pour chaque exécution.

Benchmarks

Schematic est évaluée sur la suite de référence MiBench2 [Hic16]. Nous restreignons notre analyse aux benchmarks dont la consommation mémoire globale est inférieure à 64 Ko, la taille de la NVM du msp430fr5969. Nous excluons également les benchmarks *stringsearch* et *rsa* qui ne s'exécutent pas correctement sur la version actuelle de ScEpTIC. Les benchmarks évalués sont *aes, basicmath, bitcount, crc, dijkstra, fft, randmath* et *rc4*.

3.5.2 Capacité à prendre en charge un espace VM limité

La taille de la mémoire volatile peut rapidement devenir un facteur restrictif sur les plateformes embarquées à faible consommation, même avec de très petits programmes. À notre connaissance, aucune méthode d'allocation mémoire mixte VM/NVM ne prend en compte cette contrainte. Pour mettre en évidence ce problème, nous avons évalué, pour chaque technique étudiée (Ratchet, Mementos, RockClimb, Alfred et Schematic) si elle pouvait exécuter les benchmarks sur une carte msp430fr5969 (64 ko NVM, 2 ko VM). Les résultats sont présentés dans le tableau 3.1.

Tableau 3.1 – Prise en compte de l'espace limité de la VM. La série de huit symboles représente les benchmarks : *aes*, *basicmath*, *bitcount*, *crc*, *dijkstra*, *fft*, *randmath*, *rc4*.

Ratchet	Mementos	RockClimb	Alfred	Schematic
✓✓✓✓✓✓✓✓	✓✓✓✓✗✗✓✗	✓✓✓✓✓✓✓✓	✓✓✓✓✗✗✓✗	✓✓✓✓✓✓✓✓

✓ : le benchmark peut être exécuté avec la taille de la VM limitée
✗ : La taille de la VM n'est pas suffisante

Ratchet et RockClimb utilisent la NVM comme mémoire de travail et n'ont donc pas besoin de VM. Ces techniques peuvent donc, par conception, exécuter tous les benchmarks. Cependant, en tant que techniques NVM uniquement, elles ne bénéficient pas des réductions d'énergie qui seraient possibles avec la VM.

Mementos utilise la VM comme mémoire de travail pour toutes les variables. Il ne peut donc pas exécuter de benchmarks dont la taille cumulée des variables est supérieure

à la taille de la VM, ce qui est le cas pour *dijkstra* (qui a besoin de 30 Ko de VM), *fft* (16.7 Ko) et *rc4* (6.5 Ko).

ALFRED relocalise les accès à la mémoire pour n'utiliser la VM que lorsque cela est rentable du point de vue de la consommation d'énergie. Cependant, comme il utilise le même décalage d'adresse pour accéder à la fois aux données dans la VM et aux données dans la NVM, une grande taille de VM (identique à la taille de la NVM) est nécessaire, même si seuls quelques octets sont effectivement accédés. Cela explique pourquoi ALFRED, comme les techniques basées uniquement sur la VM, est incapable d'exécuter les benchmarks *dijkstra*, *fft* et *rc4*.

SCHEMATIC profite de la réduction d'énergie apportée par la VM tout en tenant compte de sa capacité limitée. Il est donc en mesure d'exécuter tous les benchmarks.

Dans l'ensemble, seuls RATCHET, ROCKCLIMB et SCHEMATIC sont capables d'exécuter tous les benchmarks avec la taille de la VM de la plateforme msp430fr5969. Cependant, par rapport à RATCHET et ROCKCLIMB, SCHEMATIC bénéficie de réduction de consommation d'énergie résultant de l'utilisation de la VM.

3.5.3 Capacité à assurer la progression du programme

La capacité des techniques étudiées à assurer la progression est évaluée en les testant sur le même scénario d'intermittence simple, consistant en des pertes d'alimentation périodiques. Les techniques ont été testées pour différentes valeurs de TBPF, qui ont été choisies en fonction de la durée des benchmarks. Nous avons mesuré le temps d'exécution (en cycles d'horloge, avec toutes les données en VM) des benchmarks pour sélectionner une gamme de TBPF qui nous permet de couvrir différentes situations, de l'absence de panne pendant l'exécution du programme à des pannes fréquentes (voir le tableau 3.2). Les TBPF résultants sont 1k, 10k, et 100k cycles d'horloge. Pour chaque valeur de TBPF, nous fixons E_B à la quantité moyenne d'énergie consommée par la plateforme dans l'intervalle.

Le tableau 3.3 indique si les benchmarks terminent leur exécution ou non, pour toutes les techniques et toutes les valeurs de TBPF.

ALFRED, MEMENTOS et RATCHET ne s'adaptent pas aux caractéristiques de la plateforme et ne peuvent donc pas garantir la progression du programme pour de petites valeurs de TBPF. ROCKCLIMB et SCHEMATIC, en revanche, adaptent leur placement des points de sauvegarde pour toujours garantir la progression du programme. Nous considérons qu'un TBPF de 10k est un bon compromis entre l'intermittence extrême et l'absence

Tableau 3.2 – Temps d'exécution et nombre minimal de pertes d'alimentation pour chaque *intervalle entre les pertes d'alimentation* (TBPF) par benchmark (en cycles d'horloge)

Benchmark	Temps d'exécution (en cycles d'horloge)	Nombre minimal de pertes d'alimentation pour TBPF=	1k	10k	100k
aes	1 079 363		1 080	108	11
basicmath	169 599		170	17	2
bitcount	819 411		820	82	9
crc	41 133		42	5	0
dijkstra	1 381 746		1 382	139	14
fft	377 578		378	38	4
randmath	15 062		16	2	0
rc4	437 335		438	44	5

Tableau 3.3 – Capacité à assurer la progression du programme. La série de huit symboles représente les benchmarks : *aes*, *basicmath*, *bitcount*, *crc*, *dijkstra*, *fft*, *randmath*, *rc4*.

Technique	TBPF (cycles) 1k	10k	100k
Ratchet	✗✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓
Mementos	✗✓✗✗✗✗✗✗	✗✓✓✓✗✗✗✗	✓✓✓✓✗✓✓✓
RockClimb	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓
Alfred	✗✗✓✓✓✓✗✓	✓✓✓✓✓✓✗✓	✓✓✓✓✓✓✓✓
Schematic	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓✓✓✓✓

✓ : le benchmark a terminé son exécution. ✗ : le benchmark n'a pas pu se terminer.

d'intermittence pour les benchmarks considérés. Nous avons donc utilisé cette valeur pour les expériences présentées dans les sections 3.5.4 et 3.5.5.

3.5.4 Consommation énergétique

Réduire la consommation d'énergie des programmes intermittents est crucial, car cela permet aux capteurs sans batterie d'effectuer plus de calculs avec la même quantité d'énergie récoltée, améliorant ainsi le débit de ces derniers. Cette section évalue la consommation d'énergie des huit benchmarks considérés (gestion de l'intermittence incluse) pour des pertes d'alimentation périodiques.

La figure 3.6 affiche la consommation d'énergie des benchmarks considérés pour un

TBPF de 10k cycles pour toutes les techniques analysées. Chaque barre de la figure divise la consommation d'énergie en quatre catégories :

— Calcul (▨) : énergie pour l'exécution du programme, y compris le coût énergétique des accès à la mémoire, et en excluant les coûts énergétiques des réexécutions après une perte d'alimentation pour les techniques qui nécessitent des réexécutions.

— Sauvegarde (▨) : énergie dépensée pour sauvegarder les données volatiles dans la NVM. Ce coût énergétique dépend du nombre de points de sauvegarde et de la taille des données.

— Restauration (▨) : énergie dépensée pour restaurer les données volatiles lors du redémarrage de la plateforme après une perte d'alimentation.

— Réexécution (■) : énergie dépensée pour ré-exécuter le code pour les techniques qui reprennent l'exécution du point de sauvegarde précédent après une perte d'alimentation.

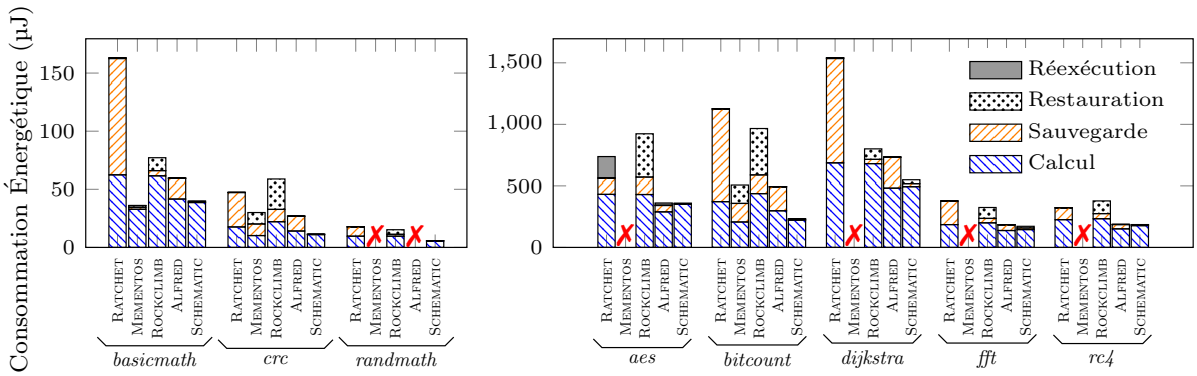


FIGURE 3.6 – Consommation énergétique des huit benchmarks pour les différentes techniques et un TBPF de 10k cycles. Les benchmarks qui n'ont pas pu finir sont marqués d'une croix rouge (✗).

Des constats généraux peuvent être faits, quel que soit le benchmark analysé.

Tout d'abord, SCHEMATIC permet en général une réduction de la consommation d'énergie globale par rapport aux autres solutions. En moyenne, SCHEMATIC consomme 51 % d'énergie en moins que les techniques de référence (la comparaison a été effectuée sur les benchmarks qui ont terminé uniquement).

Deuxièmement, SCHEMATIC et ROCKCLIMB ne dépensent pas d'énergie en réexécution car ils placent des points de sauvegarde de manière sûre dans le programme et attendent que le condensateur soit complètement réapprovisionné.

Troisièmement, la consommation d'énergie consacrée aux calculs diffère d'une tech-

nique à l'autre. Cela est dû à leurs allocations mémoire différentes. Comme prévu, Mementos consomme le moins d'énergie lors de l'exécution du programme puisque toutes les données sont allouées dans la VM, contrairement à Ratchet et RockClimb, qui stockent toutes les données dans la NVM. Alfred et Schematic, d'autre part, optent pour une allocation mémoire mixte, ce qui leur permet de consommer moins que les techniques tout-NVM.

Quatrièmement, par rapport aux autres techniques, Schematic sauvegarde les données volatiles du programme beaucoup moins souvent et pour un sous-ensemble réduit de variables. Ainsi, le coût énergétique lié à la sauvegarde dans Schematic est inférieur à celui des autres techniques.

Enfin, Schematic et RockClimb ont tendance à dépenser plus d'énergie dans la restauration des variables que les autres techniques. Cela est dû au fait que les deux techniques sauvegardent et restaurent les données volatiles à tous les points de sauvegarde, en supposant de manière conservatrice que la plateforme entre en sommeil profond et que la VM est donc perdue. Une gestion plus efficace des modes de veille (qui éviterait de sauvegarder des données lorsqu'il y a suffisamment d'énergie pour le faire) est laissée en suspens pour des travaux futurs. Il convient de noter que, même dans ce cas, il existe toujours un coût incompressible pour la sauvegarde/restauration des variables dont le placement en mémoire est modifié au point de sauvegarde.

Pour conclure, le placement de points de sauvegarde et l'allocation mémoire de Schematic entraînent une réduction significative de la consommation d'énergie. En ce qui concerne le temps d'exécution, des expériences similaires ont été menées et montrent des résultats analogues avec une réduction globale du temps d'exécution de 54 % en moyenne par rapport aux techniques apparentées.

3.5.5 Qualité de l'allocation mémoire de Schematic

Schematic vise à réduire l'énergie consommée par un programme en tirant profit des architectures hybrides VM/NVM. Afin de mesurer comment Schematic bénéficie de l'allocation de données dans la VM, nous avons comparé l'algorithme Schematic (placement conjoint de points de sauvegarde et allocation mémoire) à une version modifiée de Schematic appelée Tout NVM, où aucune allocation mémoire dans la VM n'est effectuée (toutes les données sont stockées dans la NVM).

Pour mieux comprendre les apports de l'allocation mémoire de Schematic, nous avons divisé l'énergie dépensée pour les calculs en trois catégories : les calculs sans accès

à la mémoire, les accès à la VM et les accès à la NVM. Les résultats sont présentés dans la figure 3.7.

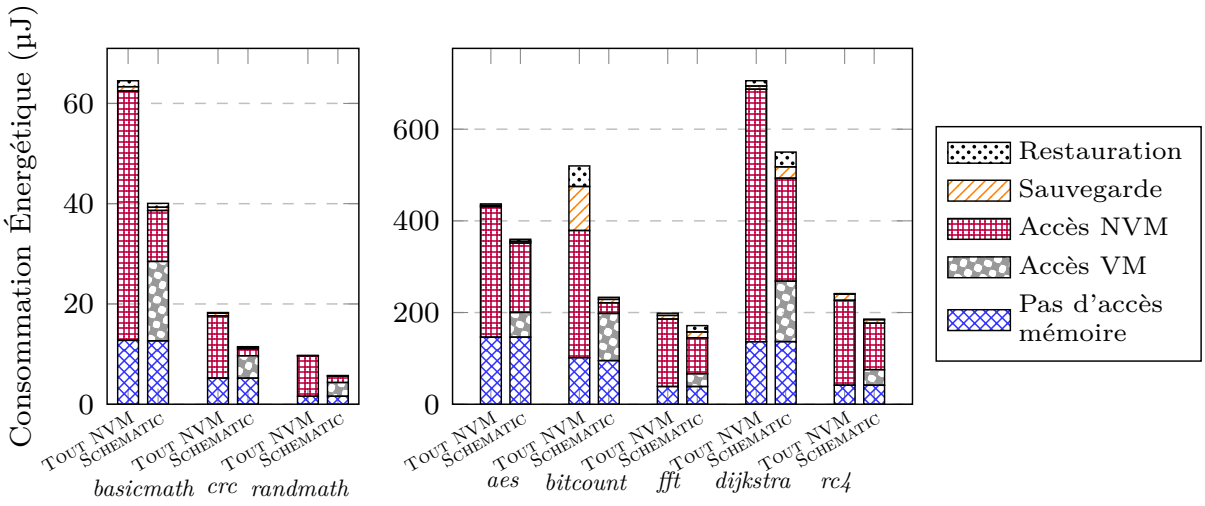


FIGURE 3.7 – Consommation énergétique de SCHEMATIC et *Tout NVM* pour les benchmarks étudiés (TBPF=10k cycles d'horloge)

Pour tous les benchmarks, l'allocation des variables en VM permet de réduire considérablement la consommation énergétique : en moyenne, SCHEMATIC nécessite 25 % d'énergie en moins par rapport à Tout NVM. L'allocation mémoire de SCHEMATIC permet ainsi d'exploiter efficacement l'architecture hybride VM/NVM, avec en moyenne 69 % des accès mémoire effectués ciblant la VM, ce qui représente 33 % de l'énergie dépensée pour les calculs.

3.5.6 Impact de la taille du condensateur

La taille du condensateur, et donc le budget énergétique du système E_B, joue un rôle important dans la consommation d'énergie de toutes les techniques, car un petit E_B entraîne des pertes d'alimentation plus fréquentes. L'impact du budget énergétique sur la consommation d'énergie est illustré dans la figure 3.8 pour les cinq techniques sur le benchmark *crc* (qui a été sélectionné parce qu'il illustre le mieux l'influence de E_B sur la consommation d'énergie).

Comme on pouvait s'y attendre, le coût global de la gestion de l'intermittence (somme de ▨, ⊡ et ■) diminue lorsque E_B augmente. Cependant, la diminution est plus faible pour ALFRED et RATCHET que pour SCHEMATIC, parce que leur placement des points

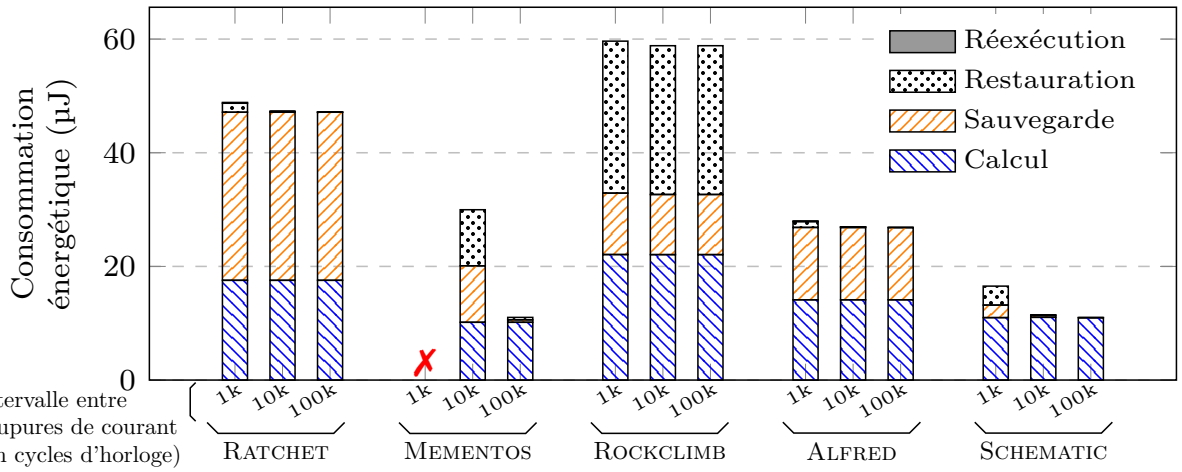


Figure 3.8 – Impact de la taille du condensateur, pour le benchmark *crc*. Pour simplifier l'implémentation sur le simulateur ScEpTIC, nous ne faisons pas varier directement E_B mais nous faisons varier le TBPF résultant, car un petit E_B entraîne un petit TBPF.

de sauvegarde (et donc l'énergie pour la sauvegarde des variables dans la NVM) ne dépend pas des caractéristiques de la plateforme, et est donc constant. Mementos effectue un point de sauvegarde conditionnel en fonction de l'énergie disponible au moment de l'exécution, il s'adapte donc au budget énergétique de la variation. RockClimb adapte le placement des points de sauvegarde aux caractéristiques de la plateforme. Cependant, comme il place les points de sauvegarde sur tous les en-têtes de boucle et avant tous les appels de fonction, sa consommation énergétique liée à la sauvegarde ainsi qu'à la restauration reste élevée. En revanche, pour Schematic, les coûts de sauvegarde et de restauration diminuent lorsque la valeur de E_B augmente, car moins de points de sauvegarde sont ajoutés au programme.

3.6 Conclusion

Ce chapitre a présenté Schematic, une technique de compilation qui automatise le placement des points de sauvegarde et l'allocation mémoire des variables d'un programme intermittent, afin de minimiser sa consommation énergétique. Schematic garantit la progression du programme, et adapte le placement des points de sauvegarde et l'allocation mémoire aux paramètres architecturaux (taille du condensateur, taille de la mémoire volatile). Une évaluation expérimentale de Schematic démontre sa capacité à exécuter des benchmarks dans des situations où d'autres solutions ne le pourraient pas (pertes d'ali-

mentation fréquentes, petites mémoires volatiles). Les expériences montrent également que SCHEMATIC permet des réductions de consommation d'énergie significatives par rapport aux techniques existantes.

Dans les travaux futurs, une direction serait de permettre aux variables pointées de changer d'allocation mémoire au cours de l'exécution, en utilisant une analyse statique « point-to ». Une autre direction consisterait à tirer parti des différents modes de veille du MSP430 pour éviter de sauvegarder l'ensemble des données volatiles (variables et registres) à chaque point de sauvegarde. En outre, nous avons constaté que le placement de point de sauvegarde de SCHEMATIC était contraint par la structure du code (fonctions, boucles) et que, par conséquent, la distance entre deux points de sauvegarde pouvait être plus courte que prévu. Cela suggère qu'il serait possible de se réveiller plus tôt, avec un condensateur partiellement chargé, sans compromettre la progression du programme pour certaines portions de code nécessitant moins d'énergie. Cette observation nous a conduit à réfléchir aux impacts du choix du moment de réveil, un sujet que nous abordons dans le prochain chapitre.

Chapitre 4

EarlyBird : L'énergie appartient à ceux qui se lèvent tôt

Quand se réveiller après une perte d'alimentation ?

Dans le chapitre précédent, nous avons vu comment il était possible d'instrumenter un programme avec des points de sauvegarde pour rendre possible son exécution de manière intermittente, avec Schematic pour exemple. Avec ce type de méthodes, la procédure en cas de perte d'alimentation est simple : attendre que le condensateur se remplisse, puis reprendre l'exécution du programme depuis le dernier point de sauvegarde.

Dans ce chapitre, nous nous interrogeons sur la pertinence d'attendre que le condensateur soit plein avant de reprendre l'exécution du programme : Est-ce toujours nécessaire ? Est-ce bénéfique en termes de performance ? Pour répondre à ces questions, nous avons mené une courte étude expérimentale pour observer l'impact du choix d'une tension de réveil sur l'exécution d'un programme intermittent. Fort des conclusions de cette étude, nous proposons EarlyBird.

EarlyBird est, à notre connaissance, la première technique à aborder le problème de la sélection de la tension de réveil dans les systèmes intermittents qui utilisent des points de sauvegarde placés statiquement. EarlyBird sélectionne automatiquement une tension de réveil pour chaque point de sauvegarde du programme, adaptée à la portion de code à exécuter à la reprise de l'exécution.

Dans la suite de ce chapitre, nous explorerons tout d'abord l'impact de la tension de réveil sur l'exécution de programmes intermittent section 4.1 pour comprendre les motivations derrière EarlyBird, puis nous présenterons en détail EarlyBird et son fonctionnement section 4.2. Par la suite, nous procéderons à une évaluation expérimentale d'EarlyBird section 4.3, en comparant EarlyBird aux tension de réveil les plus courantes, puis en mesurant comment EarlyBird peut améliorer les performances de techniques de l'état de l'art. Enfin, nous conclurons section 4.4.

4.1 Impact de la tension de réveil sur les programmes intermittents

Pour déterminer la quantité d'énergie récoltée, les capteurs sans batterie mesurent la tension du condensateur, qui est directement liée à son état de charge. Les programmeurs/concepteurs de capteurs sans batterie peuvent alors décider de la *tension de réveil* du capteur, c'est-à-dire de la tension en dessous de laquelle l'appareil est éteint et récolte de l'énergie. Le choix d'une telle tension est complexe et a une incidence sur plusieurs paramètres.

Dans cette section, nous nous proposons d'illustrer l'impact du choix de la tension de réveil sur l'exécution d'un programme intermittent. Pour cela, nous identifions tout d'abord section 4.1.1 les paramètres qui pourraient être impactés par une variation de la tension de réveil. Ensuite, nous explorons les liens entre tension d'alimentation d'un microcontrôleur et consommation d'énergie section 4.1.2. Enfin, nous évaluons les impacts de la tension de réveil sur l'exécution d'un programme, tout d'abord qualitativement section 4.1.3 puis quantitativement section 4.1.4.

Pour illustrer l'impact de la tension de réveil sur l'exécution d'un programme intermittent, nous nous baserons sur le code d'exemple figure 4.1, dont l'exécution avec deux tensions différentes sont visibles figure 4.1a (tension de réveil basse) et figure 4.1b (tension de réveil haute).

4.1.1 Contexte et Hypothèses

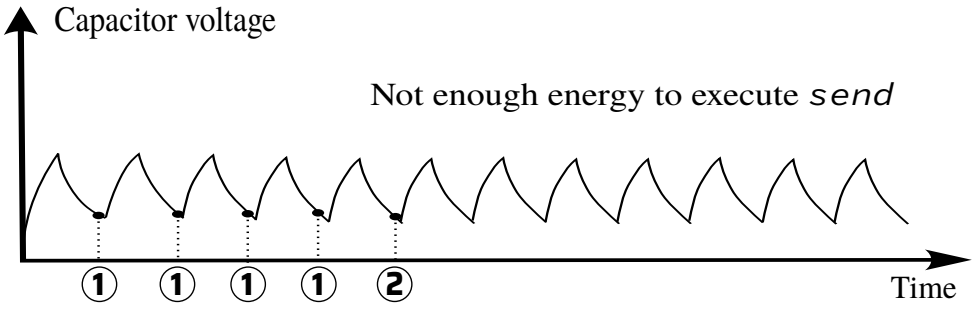
Nous avons identifié trois paramètres d'un programme intermittent qui pourraient être impactés par le choix d'une tension de réveil :

P_1 – *Consommation d'énergie (ou "l'énergie appartient à ceux qui se lèvent tôt")* : Les microcontrôleurs acceptent généralement une large plage de tension d'entrée (par exemple de 1,8 V à 3,6 V), mais leur consommation d'énergie varie en fonction de cette tension. Ainsi, diminuer la tension d'entrée du microcontrôleur réduit la consommation énergétique de celui-ci, ce qui le rend plus efficace. Ceci est illustré sur la figure 4.1, où l'exécution atteint le point de sauvegarde ② plus rapidement avec une faible tension de réveil (figure 4.1a) qu'avec une tension plus élevée (figure 4.1b).

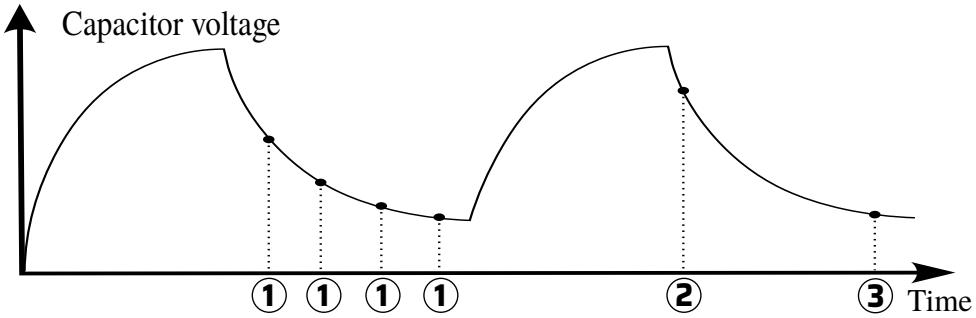
P_2 – *Progression du programme* : le choix d'une tension de réveil a un impact sur la capacité à assurer la progression du programme. Une tension de réveil trop basse peut me-

```
int sum = 0;
for(int i = 0; i<4; i++){
    checkpoint(); ①
    sum += array[i] ⚡
}
checkpoint(); ②
send(sum); ⚡⚡⚡
checkpoint(); ③
```

Programme instrumenté pour une exécution intermittente. Le programme contient trois points de sauvegarde, étiquetés ①, ② et ③. La fonction `send` doit s'exécuter de manière atomique – son exécution complète doit se terminer avant une perte d'alimentation – et nécessite plus d'énergie que le calcul de la somme.



(a) Profil d'exécution avec une tension de réveil basse.



(b) Profil d'exécution avec une tension de réveil haute.

FIGURE 4.1 – Exemple de programme instrumenté pour l'exécution intermittente et de possibles profils d'exécution

ner à un scénario dans lequel l'état du programme ne peut pas être sauvegardé avant une perte d'alimentation - empêchant ainsi tout progrès. Dans la figure 4.1a, la faible tension de réveil ne permet pas d'exécuter l'ensemble du programme, car l'énergie consommée par la fonction `send` dépasse l'énergie disponible au redémarrage. Il est donc essentiel que la tension de réveil soit adaptée à la charge à exécuter.

P_3 – *Régularité de la progression* : il est parfois important que les programmes s'exécutent aussi régulièrement que possible, notamment pour réagir aux événements externes. Cependant, lorsque l'appareil est éteint à la suite d'une perte d'alimentation, ce dernier ne peux pas réagir à ces événements. Afin d'assurer la régularité de la progression, un système intermittent doit avoir des phases de récolte courtes. Le choix d'une tension de réveil définit la quantité d'énergie à récolter avant de redémarrer le système et a donc une incidence directe sur la durée des phases de récolte. Il est donc avantageux de choisir une tension de réveil basse pour obtenir des phases de récolte courtes.

Dans les sections suivantes, nous étudions expérimentalement l'impact de la tension de réveil sur ces trois propriétés.

4.1.2 Lien entre tension d'alimentation et consommation énergétique

Comme exprimé dans P1, la consommation d'énergie d'un microcontrôleur dépend de sa tension d'entrée, une tension plus élevée entraînant une augmentation de la consommation d'énergie. Dans cette section, nous vérifions cette affirmation et quantifions les économies d'énergie réalisables avec une tension d'entrée plus faible. Pour ce faire, nous avons mesuré la puissance consommée instantanée d'un microcontrôleur TI msp430fr5969 sur sa plage de tension d'alimentation nominale (de 1,8 V à 3,6 V) avec un analyseur de puissance N6705A [KEY]. Le msp430fr5969 (" MSP430 " en abrégé) est directement alimenté par l'analyseur de puissance et exécute une boucle infinie d'instructions NOP. Les résultats sont visible figure 4.2.

Nous observons que la puissance consommée par le microcontrôleur augmente à peu près linéairement avec la tension jusqu'à 3,3 V, après quoi elle augmente plus rapidement. Au total, on observe un facteur quatre entre la puissance consommée liée à la tension la plus basse et celle liée à la tension la plus haute. Si l'on ne considère que la partie linéaire, la consommation de puissance est multipliée par deux entre 1,8 V et 3,3 V. Sur cette plage, le microcontrôleur se comporte comme un consommateur de courant constant.

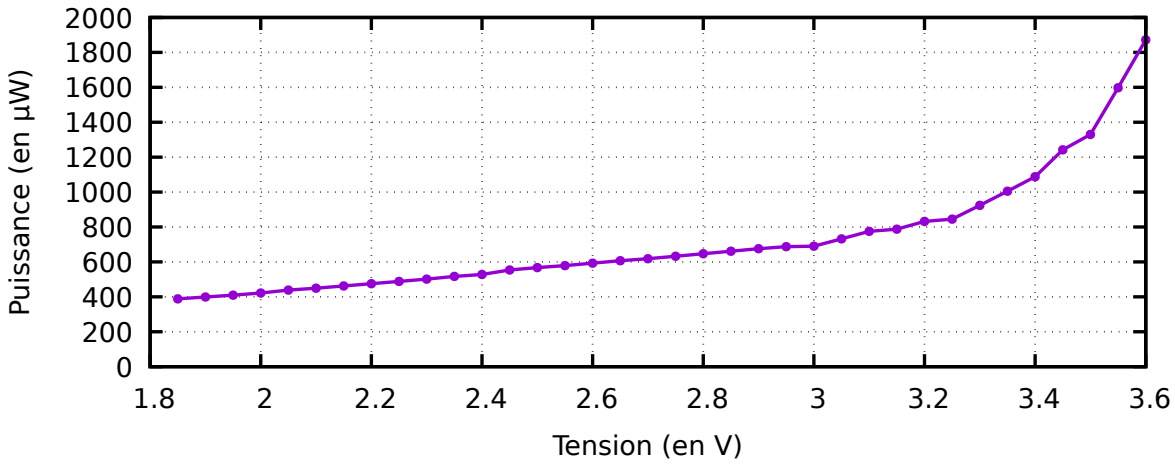


FIGURE 4.2 – Évolution de la puissance consommée sur la plage de tension de fonctionnement du msp430fr5969 à 1 MHz

Il est important de noter que durant toute l'expérience, la fréquence d'opération ne varie pas (1MHz), donc la variation de consommation d'énergie pour un programme donné sera la même que la variation de puissance.

Au vu de ces résultats, il apparaît clairement que des économies d'énergie considérables peuvent être obtenues en faisant fonctionner le microcontrôleur à une tension plus faible. Cela nous incite à explorer plus en détail l'impact de la tension de réveil sur l'exécution du programme.

4.1.3 Influence de la tension de réveil sur le profil d'exécution

La tension de réveil dicte la quantité d'énergie disponible pour un cycle d'exécution ainsi que le temps nécessaire pour récolter cette énergie. Par conséquent, le choix d'une tension de réveil a des effets significatifs sur le comportement du programme, notamment sa capacité à assurer la progression du programme (P_2) et sa régularité (P_3). Dans cette section, nous étudions l'impact de deux stratégies de tension de réveil couramment utilisées sur le comportement du programme.

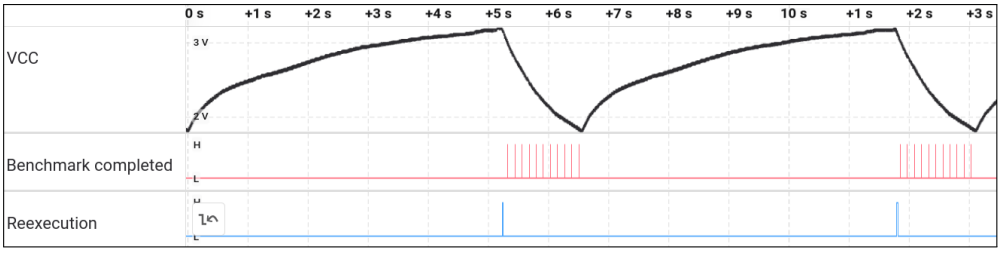
La première stratégie, appelée V-HIGH, surveille la tension du condensateur et reprend l'exécution lorsque le condensateur est presque plein (3.3 V). Plusieurs travaux [YR20 ; ABA+19 ; CKL+22] s'appuient sur cette hypothèse pour garantir la progression, dont SCHEMATIC [RBB+24b].

La stratégie V-LOW, en revanche, repose sur l'hystérésis entre la tension de mise sous

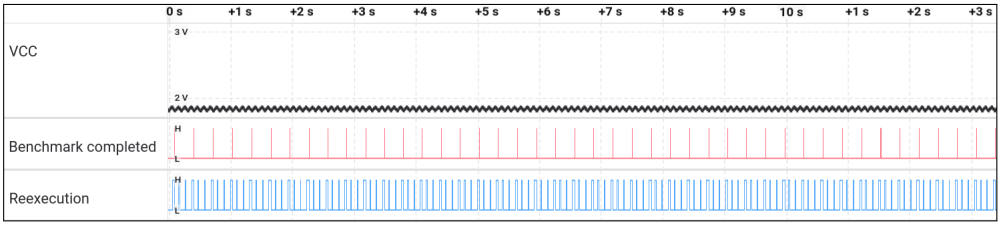
tension (1,88 V) et la tension de mise hors tension (1,8 V) du MSP430 . Cette hystérésis, implémentée en matériel, vise à éviter les oscillations dans l'alimentation du microcontrôleur. Dans le cas d'une exécution intermittente, elle devient un moyen simple de s'assurer qu'une certaine quantité d'énergie a été récoltée avant de redémarrer après une perte d'alimentation. Cela impose une tension de réveil de 1,88 V. Comme elle ne nécessite aucune mesure de la tension du condensateur du côté logiciel, cette stratégie est la plus facile à mettre en place, et elle est utilisée dans plusieurs travaux tels que ceux de Van Der Woude et Hicks, ou Yıldırım *et al.*, [WH16 ; YMP+18].

Pour évaluer l'effet de la tension de réveil sur le profil d'exécution, nous avons exécuté le benchmark *aes* de manière itérative en utilisant les stratégies V-High et V-Low. Le benchmark est extrait de la suite de benchmarks Mibench2 [Hic16] et est instrumenté avec des points de sauvegarde insérés avant les retours de fonction. À l'aide d'un analyseur de signaux, nous avons surveillé l'évolution de la tension aux bornes du condensateur, ainsi qu'un signal émis à chaque fois que le benchmark était exécuté avec succès.

L'observation de la figure 4.3 nous donne un aperçu de l'impact des stratégies de réveil sur l'exécution du programme.



(a) Profil d'exécution avec la strategie V-High



(b) Profil d'exécution avec la strategie V-Low

Figure 4.3 – Profil d'exécution pour les stratégies V-High et V-Low. La courbe *VCC* représente la tension aux bornes du condensateur ; le signal *Benchmark Completed* est émis lorsque le benchmark s'est terminé avec succès ; le signal *Reexecution* est émis lorsque la plateforme commence à réexécuter le code.

Ce qui frappe immédiatement, c'est la différence d'échelle de la durée des cycles d'in-

termittence entre les deux stratégies : les phases d'exécution pour la stratégie V-High durent environ 1 seconde, alors qu'elles ne durent que 50 ms avec V-Low. En conséquence, nous observons que les exécutions des benchmarks sont réparties uniformément dans le temps avec la stratégie V-Low alors qu'elles sont regroupées avec la stratégie V-High. De plus, les longues phases de récolte dans V-High réduisent la régularité de la progression (P_3). Néanmoins, comme la stratégie V-Low subit des pertes d'alimentation plus fréquentes, elle passe un temps important à réexécuter des portions de code dont la progression n'a pas pu être sauvegardée.

4.1.4 Influence de la tension de réveil sur les performances du programme

Dans cette section, nous comparons quantitativement les stratégies V-High et V-Low. Tout d'abord, nous évaluons la capacité de la stratégie à assurer la progression du programme, visible dans le tableau 4.1.

Tableau 4.1 – Capacité des stratégies V-Low et V-High à assurer la progression du programme. Pour plus de détails, les méthodes de placement de points de sauvegarde (*LL/FR*) et (*S-10%/S-100%*) sont décrites section 4.3.

	Sauvegarde fréquente (*LL/FR*)			Sauvegarde occasionnelle (*S-10%/S-100%*)		
	aes	*crc*	*rc4*	*aes*	*crc*	*rc4*
V-Low	✓	✓	✓	✗	✓	✗
V-High	✓	✓	✓	✓	✓	✓

Comme attendu, la stratégie V-High assure la progression du programme sur les trois benchmarks. Malheureusement, la stratégie V-Low ne parvient pas à le faire avec *aes* et *rc4* lorsque les points de sauvegarde sont éloignés les uns des autres.

Pour étudier la performance de chaque stratégie en termes de travail effectué, nous avons mesuré le nombre d'exécutions du benchmark *aes* qu'elles peuvent achever en deux minutes. Nous avons observé une différence de débit significative, avec une moyenne de 190 exécutions pour V-High et 295 pour V-Low.

Pour expliquer cette différence, nous devons nous pencher sur l'activité du capteur pendant l'expérience. La figure 4.4 affiche, pour les stratégies V-High et V-Low, la proportion de temps passé à exécuter le code et à sauvegarder les données volatiles (Exécution), à dormir en attendant que le condensateur se remplisse (Veille), à redémarrer et à charger la sauvegarde précédente (Restauration) et à réexécuter les portions de code dont la progression n'a pas pu être sauvegardée (Réexecution).

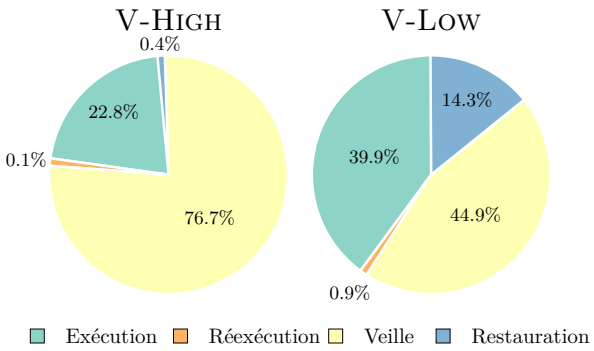


Figure 4.4 – Activité du capteur durant l'expérience avec les tensions de réveil V-High et V-Low.

Tout d'abord, nous pouvons constater que la stratégie V-Low passe environ deux fois plus de temps à s'exécuter que V-High, ce qui explique la différence du débit entre les deux techniques. Nous pouvons également observer qu'en raison de sa consommation d'énergie plus élevée, la stratégie V-High passe 76,7 % de son temps à récolter de l'énergie, alors que la stratégie V-Low n'en passe que 44,9 %.

D'un autre coté, la stratégie V-Low rencontre des pertes d'alimentation plus fréquentes, elle passe donc une partie plus importante du temps à redémarrer le msp430 et à restaurer l'état volatil du programme sauvegardé précédemment. Le temps de réexécution est marginal ici, car l'état du programme est sauvegardé fréquemment, mais ce n'est pas toujours le cas, comme nous le verrons dans l'évaluation expérimentale de EarlyBird.

Dans l'ensemble, cette étude confirme que la tension de réveil est un paramètre crucial pour les capteurs sans batterie, car elle a un impact considérable sur le comportement du programme et ses performances.

Au vu de la complexité du choix de la tension de réveil, laisser au développeur le soin d'effectuer cette tâche risque d'entraîner de mauvaises performances, voire de nuire à la progression du programme. En outre, l'exploration de différentes tensions de réveil et de leur impact sur l'exécution du programme est un processus qui prend du temps et qui doit

être effectué à chaque fois que le programme change. Les techniques existantes utilisent donc une seule tension de réveil pour l'ensemble du programme, sans tenir compte des avantages potentiels de l'adaptation de la tension de réveil à la charge de travail à exécuter. Nous soutenons qu'une telle adaptabilité est cruciale pour améliorer les performances du programme, c'est pour cela que nous proposons EarlyBird.

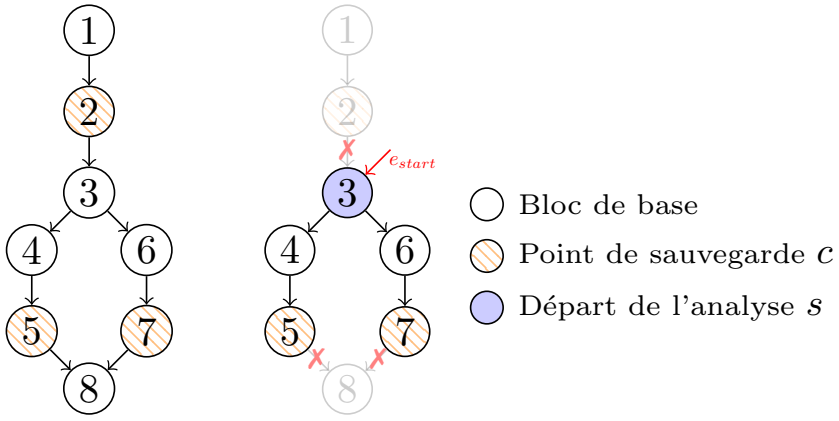
4.2 EarlyBird

EarlyBird sélectionne automatiquement la tension de réveil adaptée à chaque endroit du programme à partir duquel l'exécution peut être reprise (*i.e.*, après les points de sauvegarde). L'objectif est de réveiller le système le plus tôt possible après une perte d'alimentation, afin d'améliorer la régularité de la progression, et de bénéficier de réduction d'énergie due à l'exécution à une tension plus basse. Cependant, EarlyBird doit également s'assurer que la tension de réveil est suffisamment élevée pour atteindre le(s) prochain(s) point(s) de sauvegarde, afin d'assurer la progression du programme.

Pour ce faire, EarlyBird s'appuie sur deux composants : une analyse statique opérant sur le binaire d'un programme, qui détermine la tension de réveil minimale pour chaque point de sauvegarde et une bibliothèque pour appliquer les tensions de réveil sélectionnées au moment de l'exécution.

4.2.1 Détermination de la tension de réveil à l'aide de l'analyse statique

EarlyBird prend comme entrée un placement statique de points de sauvegarde dans le code de l'application. Ce placement peut être effectué manuellement, ou par l'intermédiaire d'un outil automatique comme Schematic. La détermination de la tension de réveil est effectuée pour chaque point de sauvegarde, à l'aide d'une analyse statique du code binaire du programme. L'analyse, pour un point de sauvegarde donné, détermine la plus grande quantité d'énergie nécessaire pour atteindre le(s) point(s) de sauvegarde suivant(s). Pour cela, elle opère sur le graphe de flot de contrôle du programme (*Control Flow Graph* en anglais, abrégé CFG). La figure 4.5a représente un exemple de CFG. Les blocs de base hachurés représentent des appels de fonction, dans cet exemple simple, des appels à la routine de sauvegarde `checkpoint()`.



(a) `main` CFG (b) Analyse depuis le bloc de base 3

FIGURE 4.5 – Graphe de flot de contrôle de la fonction `main` (a), et représentation d'une analyse de cette fonction quand l'exécution reprend du bloc de base 3 (b).

L'analyse statique de EarlyBird, pour chaque point de sauvegarde c (par exemple le bloc de base 2) vise à trouver la consommation d'énergie maximale parmi tous les chemins p tels que :

— p commence directement après le point de sauvegarde donné c (dans notre exemple le bloc de base 3) ;
— p se termine par un point de sauvegarde accessible depuis c (dans notre exemple, les blocs de base 5 ou 7) ;
— Les blocs de base de p ne contiennent aucun point de sauvegarde à l'exception du dernier bloc de base.

L'énumération de tous les chemins pour identifier celui qui consomme le plus d'énergie peut conduire à une explosion combinatoire car, en général, le nombre de chemins dans un programme est exponentiel. Par conséquent, l'analyse statique de EarlyBird s'appuie sur l'énumération implicite des chemins (*Implicit Path Enumeration Technique* en anglais, abrégé IPET) [LM95], conçue à l'origine pour l'estimation du temps d'exécution pire cas (WCET). Ici, cette technique est modifiée pour nous permettre d'identifier la quantité d'énergie nécessaire pour atteindre les points de sauvegarde suivants à partir d'un point de sauvegarde donné, dans le pire cas.

Formulation mathématique de la technique d'énumération implicite des chemins (IPET)

La technique IPET explore implicitement tous les chemins d'exécution possibles d'un programme afin de déterminer son temps d'exécution ou sa consommation d'énergie pire cas. L'IPET fonctionne en résolvant un problème de programmation linéaire en nombres entiers (*Integer Linear Programming* en anglais, abrégé ILP). Les variables de ce problème sont les nombres d'exécution des blocs de base (n_i) et des arcs ($e_{i \to j}$) du CFG. Le problème de programmation linéaire contraint les valeurs de ces variables pour modéliser :

1. *Le flot de contrôle entre les blocs de base* :

$$n_i = \sum_{j \in pred(i)} e_{j \to i} = \sum_{j \in succ(i)} e_{i \to j} \tag{4.1}$$

   exprime le fait que le nombre d'exécution d'un noeud n_i est égal au nombre de fois que ses arcs entrants et sortants sont exécutés ;

2. *Les appels de fonction*, pour chacun des contextes d'appels :

$$n_{call\_site} = n_{callee\_entry\_point} \tag{4.2}$$

3. *Les bornes de boucles*, pour chaque noeud n_i dans un corps de boucle :

$$n_i \leq M \times \sum_j e_{j \to k} \tag{4.3}$$

   avec M le nombre maximal d'itération de la boucle et $e_{j \to k}$ les arcs d'entrée de l'entête de boucle ;

4. *Le point d'entrée du programme.* Le nombre d'exécution du premier bloc de base s de la fonction dont on veut calculer le WCEC est défini à 1 :

$$n_s = 1 \tag{4.4}$$

Le solveur ILP maximise ensuite la fonction objectif $\sum n_i \times C_i$, avec C_i la constante représentant le WCEC du bloc de base n_i.

Modifications apportées par EarlyBird à l'IPET

La formulation IPET a été conçue pour déterminer le WCET/WCEC d'une fonction entière (généralement *main()*), et explore (implicitement) les chemins, sans autre contrainte que les contraintes de flot de contrôle. Dans EarlyBird, les chemins à explorer sont contraints comme suit :

C1 Les chemins considérés commencent après un appel à la routine `checkpoint()` et se terminent par un appel à la routine `checkpoint()`, mais ne doivent pas contenir d'autre appel à `checkpoint()` ;

C2 Le point d'entrée de l'analyse (après un appel à la routine `checkpoint()`, par exemple le bloc de base 3) peut se trouver à n'importe quel endroit du code (à l'intérieur d'une fonction, à l'intérieur d'une boucle), et n'est pas nécessairement le premier bloc de base d'une fonction.

La contrainte C1 est prise en compte en élaguant (virtuellement) le CFG (c'est-à-dire en ne générant aucune contrainte IPET) pour les nœuds et les arcs qui violent la contrainte C1. Appelons s le point d'entrée de l'analyse (bloc de base suivant un point de sauvegarde). Un algorithme de parcours en largeur part de s et explore le CFG (en suivant les arcs du CFG et les retours de fonctions). Tous les nœuds et arcs atteints par le parcours sont ajoutés à un ensemble atteignable R. La recherche s'arrête lorsqu'un point de sauvegarde est atteint. Les contraintes ne sont générées que pour les nœuds/arcs de l'ensemble R. La figure 4.5b représente le CFG considéré dans l'analyse, après élagage (les nœuds/arcs élagués sont grisés).

Prendre en compte la contrainte C2 est plus délicat, en raison de l'emplacement arbitraire du bloc de base de départ s dans le CFG.

Pour tenir compte du cas où s est situé à l'intérieur d'une fonction `g` (c'est-à-dire que le chemin commence dans le corps d'une fonction), la contrainte d'appel de fonction de l'IPET original (équation 4.2) doit être modifiée, car la contrainte existante relie uniquement l'exécution du bloc de base appelant la fonction `g` et le point d'entrée de `g`, sans aucune contrainte pour modéliser le retour de la fonction.

De l'*inlining* "virtuel" est utilisé pour gérer les appels de fonction. En effet, l'équation 4.2 est supprimée et remplacée par de nouvelles contraintes. Ces contraintes représentent des arcs "virtuels" qui lient le bloc de base appelant la fonction à l'entrée de la fonction, et le retour de la fonction aux blocs de base suivant l'appel de la fonction. Ceci est illustré dans la figure 4.6, où le bloc de base 3 qui appelle la fonction `g` est remplacé par le corps de `g`.

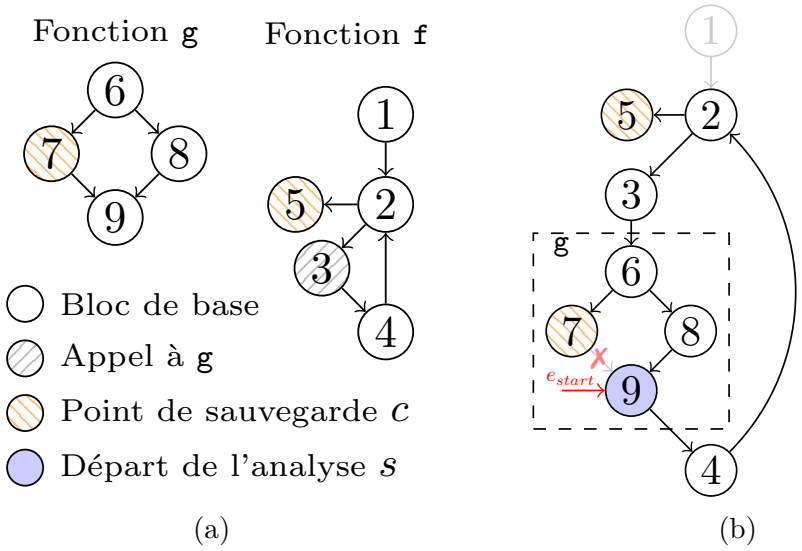


FIGURE 4.6 – Graphe de flot de contrôle de deux fonctions `f` et `g`. `f` appelle `g` dans le bloc de base 3 (a). L'analyse lorsque l'on reprend l'exécution depuis le basic block 9 est visible figure (b).

Il peut également arriver que le bloc de base de départ s soit situé à l'intérieur d'une boucle. Dans ce cas, le point d'entrée peut être exécuté plusieurs fois (voir le bloc de base 9 de la figure 4.6b). Cette situation est incompatible avec la formulation actuelle de la contrainte de point d'entrée ($n_s = 1$), nous introduisons donc un nouvel arc fictif appelé e_{start} qui pointe vers le point d'entrée n_{start}, et nous fixons le nombre d'exécutions de cet arc à 1.

De plus, comme l'entrée de la boucle est maintenant n_{start} et non l'entrée originale de la boucle, la contrainte IPET sur les limites de la boucle (équation 4.3) doit être modifiée, et est maintenant $n_i \leq M \times (e_{start} + \sum_j e_{j \to k})$ pour tous les blocs de base n_i dans la boucle.

4.2.2 Appliquer une politique de tension de réveil au moment de l'exécution

Appliquer un choix de tension de réveil à l'exécution pose deux problèmes. Tout d'abord, il est nécessaire de stocker l'information des tensions de réveil sélectionnées dans le binaire du programme. Or, la recompilation d'un programme pour intégrer les tensions de réveil calculées hors ligne risque de modifier le binaire du programme, et donc sa consommation d'énergie, ce qui n'est pas souhaitable. Deuxièmement, il est nécessaire de trouver un moyen efficace du point de vue énergétique pour surveiller la tension du

condensateur.

Ces deux défis sont relevés par la mise à disposition d'une bibliothèque de sauvegarde avec deux routines simples : `checkpoint(VOLTAGE)` et `checkpoint()`. La première routine permet de sélectionner manuellement une tension de réveil pour le point de sauvegarde donné. Elle permet à l'utilisateur de garder le contrôle de la tension de réveil et de contourner EarlyBird si nécessaire. Par exemple, cela peut être utilisé pour s'assurer qu'un périphérique s'exécute dans sa plage de tension recommandée.

La seconde routine, `checkpoint()`, permet à l'utilisateur de déléguer à EarlyBird la charge de la sélection d'une tension de réveil. Au moment de la compilation, la routine `checkpoint()` est remplacée par une routine `checkpoint(<PLACEHOLDER>)`. Cette valeur temporaire `PLACEHOLDER` est ensuite substituée dans le binaire final par la valeur de la tension de réveil calculée. Cette opération est automatisée et transparente pour l'utilisateur.

Ainsi, à l'exécution, lorsqu'un point de sauvegarde est atteint, la tension de réveil est donnée comme paramètre de la routine de sauvegarde et sauvegardée avec les données volatiles. Au redémarrage, la tension de réveil est lue à partir de la sauvegarde effectuée, et il incombe à la routine `restore` de s'assurer que suffisamment d'énergie a été récoltée avant de reprendre l'exécution du programme. Pour ce faire, nous proposons deux mécanismes : une solution *assistée par logiciel* et une solution *uniquement matérielle*.

La solution assistée par logiciel est conçue pour être compatible avec la plupart des microcontrolleurs, car il suffit que le microcontrolleur soit équipée d'un convertisseur analogique-numérique (*Analog to Digital Converter* en anglais, abrégé ADC). L'horloge du microcontrôleur réveille alors périodiquement le processeur, qui déclenche une mesure de la tension aux bornes du condensateur et vérifie si elle est supérieure au seuil sélectionné.

La solution uniquement matérielle nécessite un ADC avec un comparateur programmable, capable de se déclencher via un signal d'horloge. L'ADC est activé régulièrement par un signal d'horloge pour effectuer une mesure de la tension aux bornes du condensateur, sans intervention nécessaire du processeur. Le comparateur est programmé avec le seuil de tension de réveil désiré et déclenche une interruption lorsque la tension mesurée dépasse ce seuil. Par conséquent, le matériel une fois configuré est autonome et le processeur n'est réveillé que lorsque le seuil est atteint.

4.3 Évaluation expérimentale de EarlyBird

4.3.1 Dispositif expérimental

Nous avons évalué EarlyBird sur le microcontrôleur msp430fr5969 (msp430 en abrégé). Le msp430 est populaire dans la communauté des microcontrôleurs sans batterie car il est doté d'une RAM ferromagnétique non volatile (FRAM) économe en énergie qui permet des sauvegardes à faible coût. Dans nos expériences, le dispositif est alimenté par un système de récolte d'énergie RF Powercast P2110, utilisé fréquemment dans l'état de l'art [MCL17 ; YMP+18 ; CKL+22]. La source RF est un émetteur RF de 3 W placé à 1 mètre du système de récolte d'énergie. Un condensateur de 100 µF sert de stockage d'énergie. La figure 4.7 décrit le dispositif expérimental.

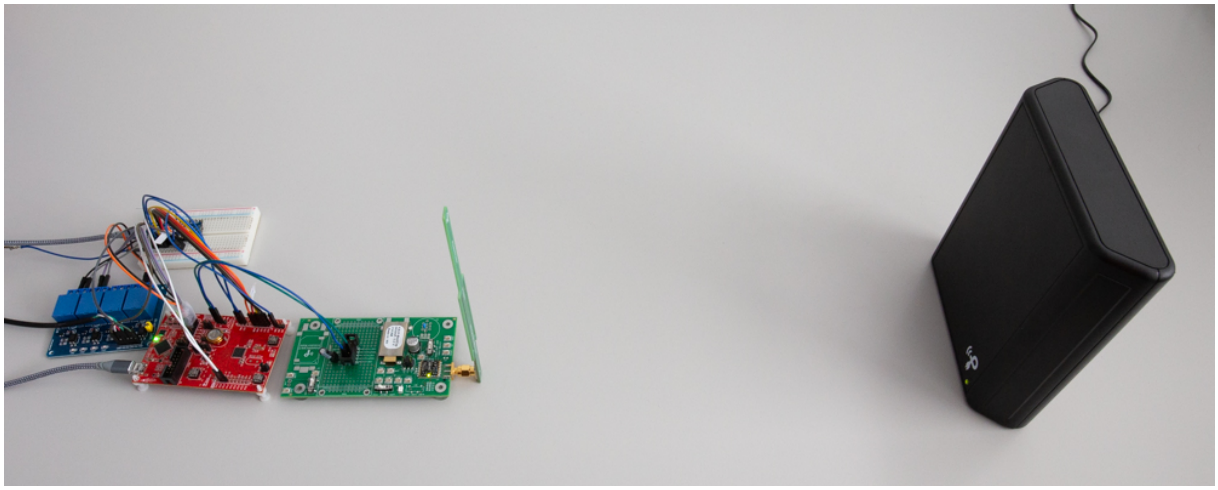


Figure 4.7 – Dispositif expérimental. Le msp430 (en rouge) est alimenté par le dispositif de récolte d'énergie RF P2110 (en vert), avec l'émetteur RF visible sur la droite (la distance entre l'émetteur et le dispositif de récolte d'énergie n'est pas à l'échelle ici). Le msp430 est contrôlé par un Arduino RP2040, et est connecté à un ordinateur portable via plusieurs relais (en bleu) pour téléverser de nouveaux programmes. Ces relais isolent le msp430 lors de l'exécution d'une expérience.

Nous avons implémenté EarlyBird sous la forme de deux outils distincts. L'analyse binaire de EarlyBird est implémentée comme une extension de l'outil open-source d'estimation du temps d'exécution pire cas Heptane [HRP17]. La partie "en ligne" de EarlyBird est implémentée comme une bibliothèque C pour le microcontrôleur msp430. Toutes les sources sont disponibles sur gitlab [1].

1. `https://gitlab.inria.fr/early-wake-up/early-wake-up-experimental-setup/`

Nous avons évalué les techniques sur trois benchmarks de la suite de benchmarks Mibench2 [Hic16] : *aes*, *crc* et *rc4*. Des points de sauvegarde ont été placés dans le code des benchmarks à l'aide de quatre heuristiques de placement de points de sauvegarde :

— Les deux premières heuristiques placent les points de sauvegarde en fonction de la structure du code. Ces heuristiques, empruntées à Mementos [RSF11], sont appelées *LL* (pour Loop Latch) et *FR* (pour Function Return). Elles insèrent des points de sauvegarde respectivement à l'intérieur de chaque boucle (à chaque bloc de base précédent l'arc retour. Ce bloc de base est appelé *Loop Latch*) et avant chaque retour de fonction. Dans la pratique, avec ce type de techniques l'état du programme est sauvegardé fréquemment.
— Les deux autres heuristiques de placement des points de sauvegarde prennent en compte la taille du condensateur. Ce placement fait en sorte qu'à partir d'un point de sauvegarde il soit possible d'atteindre le(s) point(s) de sauvegarde suivant(s) avec l'énergie contenue dans le condensateur. En utilisant la technique de placement des points de sauvegarde de Schematic [RBB+24b], nous avons généré deux placements de points de sauvegarde : l'un pour la taille réelle du condensateur et l'autre en supposant que le condensateur est 10 fois plus petit qu'il ne l'est en réalité. Nous les appelons *S-100%* (condensateur complet) et *S-10%* (10% du condensateur).

Le modèle de consommation énergétique pire cas utilisé dans notre expérience estime la consommation d'énergie d'un bloc de base sur la base de son pire temps d'exécution et des données relatives à la consommation de courant du microcontrôleur figurant dans sa fiche technique [Ins12]. Ce modèle est construit à l'aide de techniques de machines learning, comme présenté dans notre publication WORTEX [RAP24].

Le protocole expérimental consiste à exécuter un benchmark autant de fois que possible, pendant deux minutes, en utilisant l'énergie récoltée. Le nombre d'exécutions du benchmark est utilisé comme mesure principale pour évaluer la performance. Pour surveiller l'état de l'exécution, la bibliothèque d'exécution est instrumentée pour signaler l'état de l'exécution (veille, réexécution, sauvegarde...).

Dans toutes les expériences, toutes les données du programme, à l'exception du code, sont stockées en mémoire volatile.

Un mot sur le suivi de la tension du condensateur

Avant d'entamer l'évaluation de EarlyBird à proprement dit, nous avons d'abord évalué les mécanismes de suivi de la tension *assisté par logiciel* et *matériel uniquement* évoqués plus tôt. Pour chaque mécanisme, nous avons mesuré le temps nécessaire pour remplir le condensateur de 1,8 V à 3 V. Avec les deux stratégies, la tension du condensateur est échantillonnée à une fréquence fixe, définie par sa période T. Nous avons effectué des expériences avec T variant de 5 à 100 ms.

Étonnamment, la technique *assistée par logiciel* s'est avérée la plus efficace, avec un temps de remplissage du condensateur plus court de 40 % en moyenne par rapport à la technique *matériel uniquement* (1,2 s, contre 1,7 s en moyenne). En effet, bien qu'il faille régulièrement réveiller le processeur pour mesurer la tension, la technique assistée par logiciel permet d'éteindre l'ADC lorsqu'il n'est pas utilisé. En revanche, la technique matérielle exige que l'ADC soit toujours actif. Dans l'ensemble, les économies d'énergie réalisées en éteignant l'ADC pendant les périodes d'inactivité l'emportent sur la consommation d'énergie supplémentaire associée au réveil fréquent du processeur. Pour les expériences suivantes, la technique assistée par logiciel est utilisée, avec une période d'échantillonnage T de 40 ms.

4.3.2 Impact de EarlyBird sur la performance

Nous évaluons EarlyBird par rapport à deux stratégies de tension de réveil, nommées dans la suite V-High et V-Low. Comme présenté dans la section 4.1, les deux stratégies sélectionnent une tension de réveil unique pour le programme : 3,3 V pour V-High et 1,88 V, la tension de réveil par défaut du msp430 , pour V-Low. La figure 4.8 montre l'amélioration globale du débit des stratégies V-Low et EarlyBird par rapport à la stratégie V-High. Cette amélioration est calculée comme le ratio $r = \frac{N_{strategy}}{N_{\text{V-High}}}$, avec N_X le nombre de benchmarks exécutés en utilisant la stratégie X. Le nombre de benchmarks exécutés pour chaque technique est disponible à la fin de la section, dans le tableau 4.3.

La stratégie V-Low offre une amélioration des performances par rapport à V-High dans certains cas, mais échoue à garantir la progression avec certains placements de points de sauvegarde. En outre, l'amélioration obtenue avec V-Low est limitée, car une portion du temps considérable est consacrée à la réexécution de portions de code dont la progression n'a pas pu être sauvegardée. En revanche, la stratégie EarlyBird est plus performante que V-High et V-Low. EarlyBird démontre une amélioration significative

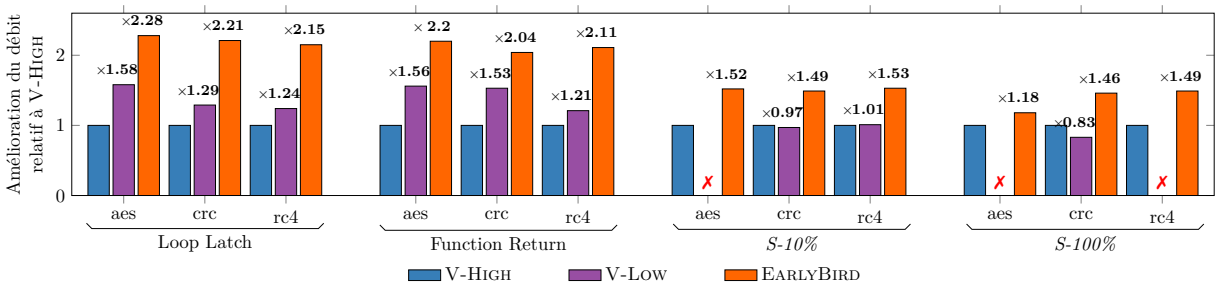


FIGURE 4.8 – Amélioration de débit par rapport à V-HIGH (ratio du nombre de benchmarks exécutés par rapport à V-HIGH). Une croix rouge (✗) indique que l'exécution n'a pas pu se terminer. Notez que les barres provenant de différentes stratégies ne peuvent pas être directement comparées en raison de la normalisation par rapport à V-HIGH.

des performances sur tous les benchmarks par rapport à V-HIGH, avec une amélioration du débit de 1,76× (moyenne géométrique). Lorsqu'il est comparé à V-LOW dans les contextes où V-LOW assure la progression, EARLYBIRD fournit une amélioration de la moyenne géométrique de 1,57×.

L'amélioration des performances d'EARLYBIRD peut être expliquée en étudiant la répartition du temps d'expérience entre le temps de sommeil, le temps de calcul, le temps de réexécution et le point de sauvegarde, visible figure 4.9.

Bien qu'EARLYBIRD soit agnostique à la méthode de placement des points de sauvegarde, le choix du placement des points de sauvegarde a un impact considérable sur l'exécution d'un programme intermittent. Avec des points de sauvegarde fréquents (*LL* et *FR*), une partie importante du temps est consacrée à la sauvegarde. Cela devient extrême avec *LL*, où les calculs ne sont presque pas visibles sur la figure. Avec des points de sauvegarde occasionnels, le temps consacré aux sauvegardes est beaucoup plus court, mais le temps consacré à la ré-exécution du code est plus important.

Néanmoins, quelle que soit la technique de sauvegarde, une part importante du temps est consacrée au remplissage du condensateur (veille). V-HIGH passe la plupart de son temps à dormir parce que sa consommation d'énergie est plus élevée en raison du fait que le MSP430 fonctionne à une tension plus élevée en moyenne. Contrairement à V-HIGH, V-LOW et EARLYBIRD bénéficient du fait d'être actifs une plus grande partie du temps. Cependant, en raison de sa faible tension de réveil, V-LOW rencontre des pertes d'alimentation plus fréquemment, et passe donc une partie importante du temps à restaurer l'état sauvegardé du programme, et à ré-exécuter le code dont la progression n'a pas été sauvegardée.

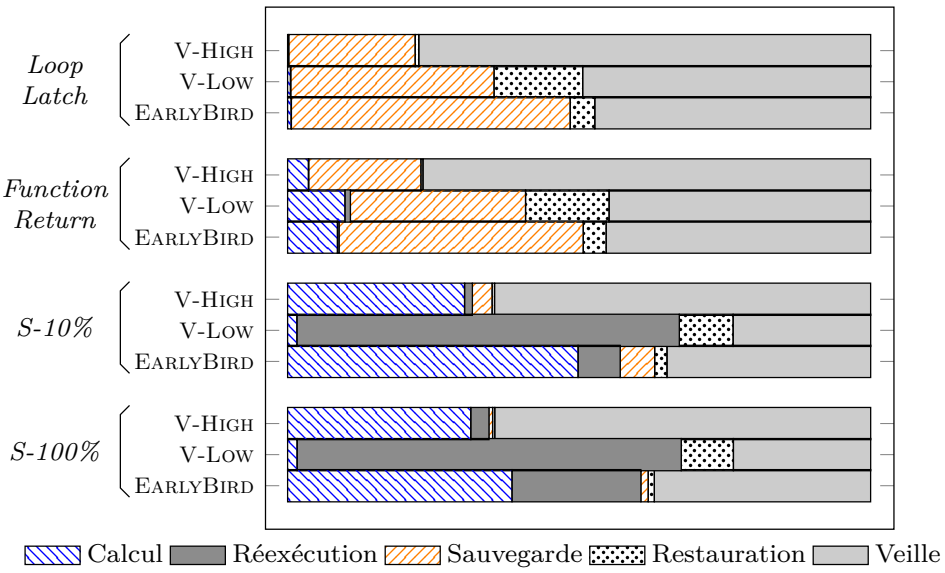


Figure 4.9 – Répartition du temps d'expérience pour l'exécution du benchmark *aes*.

Il ressort de ces expériences qu'en adaptant la tension de réveil à chaque point de sauvegarde, EarlyBird :

— assure la progression du programme dans tous les benchmarks, avec toutes les heuristiques de placement de points de sauvegarde ;
— réduit le temps de sommeil, car l'exécution à basse tension réduit la consommation d'énergie du capteur ;
— augmente le temps d'exécution, ce qui conduit à une augmentation significative du nombre de benchmarks qu'il est capable d'exécuter par rapport à une stratégie qui réapprovisionne complètement le condensateur (V-High).

Comme indiqué précédemment, la tension de réveil a également une incidence sur la durée des phases de sommeil. De longues phases de sommeil mènent l'appareil à ne pas être conscient de son environnement pendant de longues périodes. Nous proposons d'illustrer cela en mesurant le temps entre deux exécutions de benchmark pour les différentes techniques, visible dans le tableau 4.2.

Tableau 4.2 – Intervalle entre deux exécutions de benchmark, en millisecondes (Mediane, Quantile 95 % et Quantile 99 %). Le placement de points de sauvegarde est *S-10%*.

Benchmark	Tension de réveil	Mediane	95 %	99 %
aes	V-High	117	1 020	1 064
	V-Low		Pas de progression	
	EarlyBird	208	318	327
crc	V-High	40	40	927
	V-Low	40	91	92
	EarlyBird	40	40	139
rc4	V-High	9 240	9 533	9 662
	V-Low	9 062	9 327	9 375
	EarlyBird	6 115	6 251	6 337

En regardant la valeur médiane, nous observons que pour un benchmark comme *crc*, ayant un temps d'exécution court, la plupart du temps l'intervalle entre les exécutions de benchmark est de 40 ms, ce qui correspond à la durée du benchmark. En effet, le benchmark peut être exécuté plusieurs fois dans un cycle de calcul, même avec un condensateur qui n'est pas plein. Dans ce cas, les intervalles entre les exécutions du benchmark qui sont supérieurs au quantile 99 % correspondent au temps de sommeil. Pour *crc*, la stratégie V-High montre un temps de sommeil important par rapport au temps d'exécution du benchmark, et la stratégie V-Low montre une certaine régularité.

Les autres benchmarks (*aes* et *rc4*) ont un temps d'exécution plus long et ne peuvent pas être exécutés entièrement avec un condensateur plein. Dans les deux cas, EarlyBird maintient un faible temps entre chaque exécution du benchmark, démontrant ainsi sa réactivité.

Tableau 4.3 – Nombre de benchmarks complétés avec les stratégies de réveil V-High, V-Low et EarlyBird.

Placement des points de sauvegarde	Benchmark	Nombre de benchmarks complétés V-High	V-Low	EarlyBird
LL	aes	15	24	34
	crc	10	12	21
	rc4	21	27	46
FR	aes	190	295	415
	crc	3 934	6 000	8 035
	rc4	44	53	92
S-10%	aes	344	0	524
	crc	1 426	1 389	2 122
	rc4	12	13	20
S-100%	aes	372	0	440
	crc	1 580	1 308	2 303
	rc4	848	0	1 266

4.3.3 Capacité de EarlyBird à améliorer les techniques de l'état de l'art

Dans cette section, nous évaluons comment EarlyBird peut améliorer deux techniques de l'état de l'art, à savoir Mementos [RSF11] et Schematic [RBB+24b]. Par rapport aux stratégies de placement de point de sauvegarde utilisées dans la section 4.3.2, qui étaient déjà clairement inspirées de Mementos et Schematic, nous utilisons ici toutes les optimisations mises en œuvre dans Mementos et Schematic, par exemple la capacité de Mementos à sauter des points de sauvegarde au moment de l'exécution.

Mementos place des points de sauvegarde à la compilation, avant les arcs retours de boucles (placement *LL*) ou avant les retours de fonction (*FR*). Ensuite, à l'exécution, Mementos économise de l'énergie en évitant de sauvegarder si ce n'est pas nécessaire, c'est-à-dire tant que la tension aux bornes du condensateur ne passe pas sous un seuil de tension (ici 2 V). Nous avons amélioré Mementos avec EarlyBird : plutôt qu'attendre le remplissage complet du condensateur, nous reprenons l'exécution du programme dès que suffisamment d'énergie a été récoltée. De plus, nous avons adapté la prise de décision pour la sauvegarde : au lieu de tester si l'énergie disponible est supérieure à un seuil fixe, nous testons qu'il y a assez d'énergie pour atteindre le(s) point(s) de sauvegarde suivant(s) (seuil variable).

De plus, nous avons étendu Schematic, présenté dans le chapitre 3 avec les capacités d'EarlyBird : au lieu d'attendre que le condensateur soit plein, nous attendons qu'il

ait assez d'énergie pour atteindre le(s) point(s) de sauvegarde suivant(s). Ici, nous avons utilisé Schematic dans sa version Tout NVM.

Les résultats pour Mementos sont visibles dans le tableau 4.5. Lorsqu'il est combiné à EarlyBird, Mementos connaît une amélioration importante du débit avec tous les benchmarks et les placements de points de sauvegarde, avec en moyenne 1,74× plus de benchmarks exécutés. Les résultats pour Schematic sont visibles dans le tableau 4.5. L'amélioration de Schematic avec EarlyBird permet une amélioration des performances supérieure à quatre (moyenne géométrique de $\times 4,63$).

Tableau 4.4 – Nombre de benchmarks exécutés avec Mementos et Mementos + EarlyBird.

Benchmark	Placement Pt de sauvegarde	Nb Benchmarks exécutés Mementos	Mementos +EarlyBird	Amélioration
aes	LL	21	38	×**1.78**
	FR	137	228	×**1.66**
crc	LL	19	34	×**1.79**
	FR	992	1 773	×**1.79**
rc4	LL	45	71	×**1.79**
	FR	81	131	×**1.62**

Tableau 4.5 – Nombre de benchmarks exécutés avec Mementos et Mementos + EarlyBird., avec le placement de point de sauvegarde *S-100%*.

Benchmark	Nb Benchmarks complétés Schematic	Schematic +EarlyBird	Amélioration
aes	142	549	×**3.87**
crc	407	2 299	×**5.65**
rc4	296	1 351	×**4.56**

4.4 Conclusion

Ce chapitre a présenté EarlyBird, une nouvelle technique pour améliorer les techniques de calcul intermittent basée sur du placement de point de sauvegarde statique. En calculant automatiquement une tension de réveil adaptée à chaque point de sauvegarde, EarlyBird tire parti des spécificités des microcontrolleurs basse consommation pour améliorer drastiquement les performances d'un programme intermittent. Les résultats expérimentaux ont démontré l'impact de EarlyBird sur le débit et la régularité de l'exécution et montrer sa capacité à assurer la progression de programmes avec des placements de point de sauvegarde de l'état de l'art. Bien que EarlyBird améliore déjà de manière significative les performances des techniques existantes, nous pensons que d'autres améliorations sont possibles. Les recherches futures pourraient explorer l'incorporation du contexte d'exécution, tel que les itérations de la boucle, pour affiner la tension de réveil au moment de l'exécution. En outre, l'amélioration des techniques de surveillance du condensateur, en prédisant le moment où la tension du condensateur atteindra le niveau requis, pourrait réduire les mesures de tension inutiles et optimiser davantage la consommation d'énergie.

Bilan et Perspectives

Bilan des travaux

L'alimentation énergétique des objets connectés est un enjeu économique et écologique d'aujourd'hui et de demain. Pour répondre à cet enjeu, les capteurs sans batterie se positionnent comme une alternative aux capteurs classiques à piles ou à batteries. Ces capteurs récoltent l'énergie ambiante pour s'exécuter, en utilisant simplement un condensateur comme réserve d'énergie. Seulement, l'énergie disponible dans l'environnement étant limitée, ces capteurs subissent des pertes d'alimentation fréquentes, entravant la progression des programmes exécutés. Le développement des capteurs sans batterie doit ainsi être accompagné du développement de mécanismes, qu'ils soient matériels ou logiciels, pour assurer la progression d'un programme malgré les pertes d'alimentation.

Dans cette thèse, nous avons proposé deux techniques complémentaires pour assurer la progression d'un programme intermittent de manière sûre et efficace. La première, Schematic, place des points de sauvegarde automatiquement dans le code du programme, et effectue l'allocation mémoire des variables de manière simultanée. De cette manière, Schematic réussit à garantir la progression du programme même dans des conditions d'extrême intermittence et à considérablement réduire la quantité d'énergie consommée par ce programme. La seconde technique, EarlyBird, s'appuie sur un placement de point de sauvegarde déjà établi, et choisit la tension de réveil adaptée à chaque point de sauvegarde, en fonction de la charge de calcul à exécuter. De cette manière, EarlyBird améliore grandement le débit d'exécution des capteurs sans batterie.

Ces deux contributions illustrent le fait que prendre une approche *consciente de l'énergie* permet de répondre au défi de la progression des programmes intermittents de manière sûre et efficace.

Limites et perspectives des approches conscientes de l'énergie

Modèles de consommation d'énergie

Dans cette thèse, nous avons fait le choix d'une approche *consciente de l'énergie*. Notre approche repose en effet sur des modèles de consommation énergétique pour la prise de décision. Cependant, la conception de tels modèles est un défi, car les documentations des processeurs ou microcontrôleurs ne contiennent que rarement des informations détaillées sur leur consommation d'énergie. Si elles sont données, c'est rarement à la granularité de l'instruction, ce qui nous intéresse dans le cas présent. De plus, a contrario du temps d'exécution qui reste relativement prévisible sur les architectures simples, la consommation énergétique varie grandement. Comme illustré dans l'étude préliminaire d'EarlyBird, la tension d'entrée du microcontrôleur a un impact considérable sur sa consommation d'énergie. De plus, la consommation énergétique dépend aussi de la température [2], ce qui rend la conception d'un tel modèle encore plus complexe.

Dans l'optique de proposer une façon simple et universelle pour prédire la consommation énergétique de l'exécution d'un programme sur microcontrôleur, nous avons proposé avec Abderaouf Nassim Amalou une méthode basée sur des techniques d'apprentissage automatique [RAP24]. Cette technique permet d'estimer la consommation énergétique pire cas à l'échelle des blocs de base d'un programme. Elle est conçue pour les cas où la documentation du microcontrôleur ne fournit pas d'information sur la consommation énergétique de ce dernier, et se base sur des mesures pour constituer un jeu de données d'apprentissage. Cependant, une fois l'apprentissage terminé, le modèle généré ne nécessite plus de mesure et peut être intégré à des outils à la compilation, comme Heptane [HRP17] ou EarlyBird [RBB+24a].

Raisonner sur l'énergie n'est pas suffisant

Au-delà des problématiques de modèles d'énergie, les approches conscientes de l'énergie sont aussi sensibles aux chutes de tension dues à la résistance équivalente du condensateur utilisé comme réserve d'énergie.

En effet, comme mis en évidence par Ruppel *et al.*, [RSD+22], la résistance série équivalente des condensateurs utilisés dans les systèmes sans batterie (jusqu'à 10 Ω), combinée à la consommation de courant importante des périphériques (≈ 50 mA pour l'utilisation

2. On observe 30% d'augmentation du courant consommé entre 0 °C et 50 °C sur le msp430fr5969[Ins12]

d'une radio) mène à des chutes de tension non négligeables ($\approx 500\,\text{mV}$). Dans ce cas, même si le condensateur contient plus d'énergie que nécessaire pour exécuter une portion de code, cette chute de tension peut mener à une extinction du capteur, car la tension d'alimentation de ce dernier passe sous le seuil d'arrêt.

Perspectives

Les approches *conscientes de l'énergie* sont dépendantes des modèles d'énergie sur lesquels elles reposent. Il existe donc un besoin de modèles de consommation énergétique fiables et précis, en particulier pour les microcontrôleurs complexes. Là où les méthodes analytiques ne sont pas applicables à cause d'un manque de documentation, les méthodes basées sur de l'apprentissage automatique permettent de générer des modèles à partir de mesures. Cependant, il est difficile d'apprécier la fiabilité de tels modèles « boîtes noires ». Un premier pas vers la compréhension de ces derniers passe par l'utilisation de méthodes d'explicabilité (*explainability* en anglais), qui tentent d'approcher ces modèles boîtes noires par des modèles explicables [LL17 ; RSG16]. Il est important de développer ces méthodes d'explicabilité pour améliorer la confiance des développeurs dans les modèles générés. Ce premier pas n'est cependant pas suffisant et des travaux doivent être menés pour améliorer la fiabilité des approches basées sur de l'apprentissage automatique.

Aussi, les techniques de prédiction de consommation d'énergie comme WORTEX nécessitent pour le moment une tension d'opération fixe ainsi qu'une température fixe. De futurs travaux pourraient explorer l'utilisation de la température ainsi que de la tension d'exécution comme paramètres d'entrée de ces modèles. Enfin, les approches *conscientes de l'énergie* reposent aujourd'hui en majorité sur des raisonnements par rapport à l'énergie et négligent l'importance de la tension.

Les travaux futurs basés sur des techniques conscientes de l'énergie devront donc, pour supporter des périphériques consommateurs de puissance, raisonner non seulement au niveau de l'énergie, mais aussi au niveau de la tension.

Adaptation à l'énergie disponible

Contrairement aux capteurs sur batteries ou à piles, les capteurs sans batterie ne peuvent rester actifs qu'une courte période sans récolte d'énergie avant de subir une perte d'alimentation. Cela entraine de longues périodes d'inactivité.

Il est crucial de considérer l'impact de ces pertes prolongées d'alimentation sur les

cas d'usage considérés. Par exemple, les capteurs sans batterie sont parfaitement adaptés à la récolte de données statistiques indépendantes de l'heure, comme l'usure d'un pont par exemple. Cependant, avec des cas d'usage où la réactivité est importante, comme la détection de feux de forêts, les capteurs sans batterie ne sont pas préconisés, sauf si certains mécanismes sont mis en place.

Un exemple de mécanisme pour améliorer la réactivité des capteurs sans batterie consiste à différer le traitement d'une donnée pour conserver de l'énergie afin de pouvoir réagir aux événements extérieurs. Par exemple, les auteurs de CatNap [ML20] proposent de stopper tout traitement de données lorsque l'énergie contenue dans le condensateur passe sous un certain seuil. De cette manière, ils disposent d'un certain budget énergétique pour réagir aux événements qui pourraient survenir dans l'environnement.

Perspectives S'adapter à l'énergie disponible est essentiel pour garantir une bonne qualité de service. Pour cela, il est possible d'adapter les traitements à la quantité d'énergie récoltée [BRY+21 ; BCL+22], en réduisant la qualité du traitement (par exemple moins de pixels sur une image ou une classification moins précise). Seulement, certains traitements ne sont pas adaptables. Une autre solution est de reporter le traitement des données à un moment où l'énergie est abondante. Malheureusement, la mémoire souvent limitée des capteurs sans batterie impose de faire un choix sur les données à stocker pour un traitement ultérieur. Il est alors nécessaire d'évaluer la pertinence de la donnée mesurée ainsi que l'évolution des conditions de récolte d'énergie.

Bien que la pertinence des données soit dépendante de l'application, il est indispensable de fournir au développeur des outils pour exprimer cette pertinence et son impact désiré sur l'exécution. Pour cela, il est nécessaire de proposer des modèles d'exécution et des techniques d'ordonnancement *conscients des données* pour les capteurs sans batterie, qui tiennent compte de la nature et de l'importance des données. Ces aspects restent encore à explorer et à développer.

La prédiction des conditions de récolte, donc de la météo pour les conditions de récolte les plus utilisées, est un problème très étudié. Seulement, les capteurs sans batterie nécessitent des modèles de prédiction court terme, avec une empreinte mémoire et processeur minimale, pour ne pas perturber l'exécution de l'application principale. Ces modèles restent encore à développer.

L'adaptation à l'énergie récoltée représente un défi majeur pour les capteurs sans batterie. Ce défi a déjà été abordé dans le contexte des capteurs alimentés par batteries,

mais les échelles de temps sont radicalement différentes. Pour les capteurs avec batteries, l'adaptation se fait généralement à l'échelle de l'heure ou de la journée, permettant une gestion plus flexible de l'énergie. En revanche, pour les capteurs sans batterie, l'adaptation doit se faire à l'échelle de la dizaine de minutes, voire de la minute, en raison de la capacité limitée des condensateurs.

Capteurs sans batterie, quel impact écologique ?

Les capteurs sans batterie offrent la promesse d'une alternative plus écologique aux capteurs à batteries, sur plusieurs points :

- la diminution de l'impact environnemental de la fabrication, car la batterie y est remplacée par un condensateur de petite taille ;
- la diminution, voire la suppression, du besoin de maintenance régulière des systèmes sur batteries ;
- la diminution de l'impact dû à la décomposition si le capteur n'est pas recyclé en fin de vie.

Malgré ces promesses, le recyclage de ces capteurs pose question. En effet, la suppression des batteries permet néanmoins de miniaturiser d'autant plus ces capteurs et pourrait complexifier leur recyclage [DO06].

Cependant, le recyclage n'est pas forcément au menu. En effet, certains universitaires rêvent d'une « Smart Dust », une nuée de microcapteurs communicants, dispersés dans l'environnement pour mesurer différentes grandeurs physiques, comme la luminosité, la température, ou les vibrations. Cette idée, popularisée en partie par des œuvres de science-fiction, est étudiée dans différentes universités. Ces capteurs sont destinés à être dispersés dans le vent [IGD+22 ; JAF+23 ; KLK+21], dans des liquides (eau, huile, sang) [JR05] ou dans l'espace. La question suivante demeure cependant : *qui ramassera cette « poussière intelligente » ?* À une époque où les matériaux nécessaires à la conception de tels systèmes sont précieux, il semble inconcevable de les disperser dans le vent.

Perspective

On peut raisonnablement penser que les capteurs sans batterie auront un impact écologique moindre que des capteurs sur batterie. Cependant, en l'absence de données quantitatives, il est impossible ni de prouver, ni d'évaluer cet impact. Pour cela, il est nécessaire d'appliquer des méthodes d'analyse de cycle de vie aux capteurs sans batterie, et

d'effectuer un comparatif entre les deux technologies. Cela requiert néanmoins de connaitre la durée de vie de tels systèmes. Or, même si certains travaux parlent d'une durée de vie *quasi-infinie* [BD20 ; RSR16], il est important d'effectuer des études de la longévité de ces systèmes. Enfin, et comme toujours, il est nécessaire de questionner les usages de ces capteurs sans batterie, aussi bien sur le plan écologique que social.

Vers un monde sans batterie ?

Même si les batteries restent indispensables pour de nombreux usages, les supprimer peut se révéler bénéfique dans de nombreux cas. Cette thèse s'est concentrée sur les capteurs sans batterie, qui ont déjà prouvé leur efficacité sur le terrain [ABC+20]. Cependant, les techniques développées dans cette thèse s'inscrivent dans un champ plus large : celui des *systèmes sans batterie*.

Les travaux dans le domaine des capteurs sans batterie s'attaquent à des architectures de plus en plus complexes et hétérogènes, notamment avec l'apparition d'accélérateurs de réseau de neurones basse consommation [BGd+22]. Un des prochains défis à relever pour les systèmes sans batterie est l'interaction avec des actionneurs : *Après une perte d'alimentation, comment reprendre ou annuler l'exécution d'une action mécanique ?* Relever ce défi permettrait l'apparition de sondes, de satellites, voire de rovers sans batterie.

Dans un autre registre, le principe de fonctionnement intermittent pourrait être appliqué à nos appareils ménagers. En effet, avec une part croissante de notre électricité provenant de sources d'énergie non pilotables, il devient nécessaire d'adapter notre consommation d'énergie à l'énergie disponible. On pourrait par exemple imaginer que nos machines à laver soient équipées de systèmes informatiques intermittents pour supporter des pertes d'alimentation fréquentes et ainsi s'adapter aux conditions météorologiques.

De plus, l'apparition d'appareils ménagers « intermittents » laisserait entrevoir la possibilité de maisons autonomes en énergie, fonctionnant sans batterie [3]. De la même manière qu'un capteur sans batterie s'adapte aux conditions de récolte d'énergie, nos maisons pourraient adopter un fonctionnement « intermittent ».

La mise en œuvre de systèmes sans batterie dans le contexte des capteurs pave ainsi la voie à une adoption plus large de ces technologies dans divers domaines, offrant des solutions durables pour les défis énergétiques futurs.

3. Voir le très bon article : `https://solar.lowtechmagazine.com/fr/2024/01/direct-solar-power-off-grid-without-batteries/`. Ce site peut se retrouver momentanément hors-ligne, car il est alimenté par des panneaux solaires.

BIBLIOGRAPHIE

[ABA+19] Saad AHMED, Naveed Anwar BHATTI, Muhammad Hamad ALIZAI, Junaid Haroon SIDDIQUI et Luca MOTTOLA. Efficient Intermittent Computing with Differential Checkpointing, in *International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2019, DOI : 10.1145/3316482.3326357.

[ABC+20] Mikhail AFANASOV, Naveed Anwar BHATTI, Dennis CAMPAGNA, Giacomo CASLINI, Fabio Massimo CENTONZE, Koustabh DOLUI, Andrea MAIOLI, Erica BARONE, Muhammad Hamad ALIZAI, Junaid Haroon SIDDIQUI et Luca MOTTOLA. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus, in *Conference on Embedded Networked Sensor Systems (SenSys)*, 2020, DOI : 10.1145/3384419.3430722.

[AY22] Khakim AKHUNOV et Kasim Sinan YILDIRIM. AdaMICA : Adaptive Multicore Intermittent Computing, *Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2022, DOI : 10.1145/3550304.

[BBF+23] Antoine BERNABEU, Mikaël BRIDAY, Sébastien FAUCOU, Jean-Luc BÉCHENNEC et Olivier H. ROUX. Cost-optimal timed trace synthesis for scheduling of intermittent embedded systems, *Discrete Event Dynamic Systems*, 2023, DOI : 10.1007/s10626-022-00372-6.

[BCL+22] Fulvio BAMBUSI, Francesco CERIZZI, Yamin LEE et Luca MOTTOLA. The Case for Approximate Intermittent Computing, in *International Conference on Information Processing in Sensor Networks (IPSN)*, 2022, DOI : 10.1109/IPSN54338.2022.00044.

[BD20] Mukesh BATHRE et Pradipta Kumar DAS. Hybrid Energy Harvesting for Maximizing Lifespan and Sustainability of Wireless Sensor Networks : A Comprehensive Review & Proposed Systems, in *International Conference on Computational Intelligence for Smart Power System and Sustainable Energy (CISPSSE)*, 2020, DOI : 10.1109/CISPSSE49931.2020.9212287.

[BDH+20] Amira Boulmaiz, Noureddine Doghmane, Saliha Harize, Nasreddine Kouadria et Djemil Messadeg. Chapter 9 - The Use of WSN (Wireless Sensor Network) in the Surveillance of Endangered Bird Species, in *Advances in Ubiquitous Computing*, 2020, DOI : 10.1016/B978-0-12-816801-1.00009-8.

[BDM+19] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset et Guillaume Salagnac. Sytare : A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems, *Transactions on Computers*, 2019, DOI : 10.1109/TC.2018.2889080.

[Ber23] Antoine Bernabeu. *Support d'exécution Pour Les Systèmes Intermittents*, thèse de doctorat, Centrale Nantes, 2023.

[BGd+22] Abu Bakar, Rishabh Goel, Jasper de Winkel, Jason Huang, Saad Ahmed, Bashima Islam, Przemysław Pawełczak, Kasım Sinan Yildirim et Josiah Hester. Protean : An Energy-Efficient and Heterogeneous Platform for Adaptive and Hardware-Accelerated Battery-Free Computing, in *Conference on Embedded Networked Sensor Systems*, 2022, DOI : 10.1145/3560905.3568561.

[BGW11] Michael Buettner, Ben Greenstein et David Wetherall. Dewdrop : An Energy-Aware Runtime for Computational RFID, in *Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[BM16] Naveed Anwar Bhatti et Luca Mottola. Efficient State Retention for Transiently-powered Embedded Sensing, in *International Conference on Embedded Wireless Systems and Networks EWSN*, 2016, ISBN : 978-0-9949886-0-7.

[BM17] Naveed Anwar Bhatti et Luca Mottola. HarvOS : Efficient Code Instrumentation for Transiently-Powered Embedded Sensing, in *Conference on Information Processing in Sensor Networks (IPSN)*, 2017, DOI : 10.1145/3055031.3055082.

[BM18] Sara S. Baghsorkhi et Christos Margiolas. Automating Efficient Variable-Grained Resiliency for Low-Power IoT Systems, in *International Symposium on Code Generation and Optimization (CGO)*, 2018, DOI : 10.1145/3168816.

[BRY+21] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim et Josiah Hester. REHASH : A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing, *Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2021, DOI : 10.1145/3478077.

[BWM+15] Domenico Balsamo, Alex Weddell, Geoff Merrett, Bashir Al-Hashimi, Davide Brunelli et Luca Benini. Hibernus : Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems, *Embedded Systems Letters*, 2015, DOI : 10.1109/LES.2014.2371494.

[CKK+19] Joyraj Chakraborty, Andrzej Katunin, Piotr Klikowicz et Marek Salamak. Early Crack Detection of Reinforced Concrete Structure Using Embedded Sensors, *Sensors*, 2019, DOI : 10.3390/s19183879.

[CKL+20] Jongouk Choi, Larry Kittinger, Qingrui Liu et Changhee Jung. Compiler Directed Speculative Intermittent Computation, 2020, DOI : 10.48550/arXiv.2006.11479.

[CKL+22] Jongouk Choi, Larry Kittinger, Qingrui Liu et Changhee Jung. Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity, in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, DOI : 10.1109/RTAS54340.2022.00012.

[CL16] Alexei Colin et Brandon Lucia. Chain : Tasks and Channels for Reliable Intermittent Programs, in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, DOI : 10.1145/2983990.2983995.

[CRL18] Alexei Colin, Emily Ruppel et Brandon Lucia. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices, in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, ISBN : 978-1-4503-4911-6.

[DO06] Arnaud De Grave et Stig Irving Olsen. Challenging the Sustainability of Micro Products Development, *International Conference on Multi-material Micro Manufacturing*, 2006, DOI : https://doi.org/10.1016/B978-008045263-0/50064-7.

[Don] John Donovan. New Applications for Energy Harvesting | Mouser Electronics, URL : https://www.mouser.fr/applications/energy-harvesting-new-applications/.

[EC16] Dina El-Damak et Anantha P. Chandrakasan. A 10 nW–1 µW Power Management IC With Integrated Battery Management and Self-Startup for Energy Harvesting Applications, *Journal of Solid-State Circuits*, 2016, DOI : 10.1109/JSSC.2015.2503350.

[Hic16] Matthew Hicks. MiBench2 : MiBench benchmark suite ported for IoT devices. 2016, URL : https://github.com/impedimentToProgress/MiBench2.

[Hic17] Matthew Hicks. Clank : Architectural Support for Intermittent Computation, in *International Symposium on Computer Architecture*, 2017, DOI : 10.1145/3079856.3080238.

[HRP17] Damien Hardy, Benjamin Rouxel et Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool, in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2017, DOI : 10.4230/OASIcs.WCET.2017.8.

[HSS15] Josiah Hester, Lanny Sitanayah et Jacob Sorber. Tragedy of the Coulombs : Federating Energy Storage for Tiny, Intermittently-Powered Sensors, in *Conference on Embedded Networked Sensor Systems (SenSys)*, 2015, DOI : 10.1145/2809695.2809707.

[HSS17] Josiah Hester, Kevin Storer et Jacob Sorber. Timely Execution on Intermittently Powered Batteryless Sensors, in *Conference on Embedded Network Sensor Systems (SenSys)*, 2017, DOI : 10.1145/3131672.3131673.

[IGD+22] Vikram Iyer, Hans Gaensbauer, Thomas L. Daniel et Shyamnath Gollakota. Wind Dispersal of Battery-Free Wireless Devices, *Nature*, 2022, DOI : 10.1038/s41586-021-04363-9.

[Ins12] Texas Instruments. MSP430FR5969 datasheet, 2012.

[JAF+23] Kyle Johnson, Vicente Arroyos, Amélie Ferran, Raul Villanueva, Dennis Yin, Tilboon Elberier, Alberto Aliseda, Sawyer Fuller, Vikram Iyer et Shyamnath Gollakota. Solar-Powered Shape-Changing Origami Microfliers, *Science Robotics*, 2023, DOI : 10.1126/scirobotics.adg4276.

[JOW+02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh et Daniel Rubenstein. Energy-Efficient Computing for Wildlife Tracking : Design Tradeoffs and Early Experiences with ZebraNet, in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, DOI : 10.1145/605397.605408.

[JR05] Michael J. Sailor et Jamie R. Link. "Smart Dust" : Nanostructured Devices in a Grain of Sand, *Chemical Communications*, 2005, DOI : 10.1039/B417554A.

[JRR14] Hrishikesh Jayakumar, Arnab Raha et Vijay Raghunathan. QUICKRECALL : A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers, in *International Conference on VLSI Design and International Conference on Embedded Systems*, 2014, DOI : 10.1109/VLSID.2014.63.

[JSW23] Kang Eun Jeon, James She et Bo Wang. Information-Aware Sensing Framework for Long-Lasting IoT Sensors in Greenhouse, in *Wireless Communications and Networking Conference (WCNC)*, 2023, DOI : 10.1109/WCNC55385.2023.10118723.

[KCN09] Byoung Chul Ko, Kwang-Ho Cheong et Jae-Yeal Nam. Fire Detection Based on Vision Sensor and Support Vector Machines, *Fire Safety Journal*, 2009, DOI : 10.1016/j.firesaf.2008.07.006.

[KEY] KEYSIGHT. N6705a DC Power Analyzer.

[KGH+22] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni et Przemysław Pawełczak. WARio : Efficient Code Generation for Intermittent Computing, in *International Conference on Programming Language Design and Implementation (PLDI)*, 2022, DOI : 10.1145/3519939.3523454.

[KLK+21] Bong Hoon Kim, Kan Li, Jin-Tae Kim, Yoonseok Park, Hokyung Jang, Xueju Wang, Zhaoqian Xie, Sang Min Won, Hong-Joon Yoon, Geumbee Lee, Woo Jin Jang, Kun Hyuck Lee, Ted S. Chung, Yei Hwan Jung, Seung Yun Heo, Yechan Lee, Juyun Kim, Tengfei Cai, Yeonha Kim, Poom Prasopsukh, Yongjoon Yu, Xinge Yu, Raudel Avila, Haiwen Luan, Honglie Song, Feng Zhu, Ying Zhao, Lin Chen, Seung Ho Han, Jiwoong Kim, Soong Ju Oh, Heon Lee, Chi Hwan Lee, Yonggang Huang, Leonardo P.

Chamorro, Yihui Zhang et John A. Rogers. Three-Dimensional Electronic Microfliers Inspired by Wind-Dispersed Seeds, *Nature*, 2021, doi : 10.1038/s41586-021-03847-y.

[KRJ09] Johnsen Kho, Alex Rogers et Nicholas R. Jennings. Decentralized Control of Adaptive Sampling in Wireless Sensor Networks, *Transactions on Sensor Networks*, 2009, doi : 10.1145/1525856.1525857.

[KYB+20] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester et Przemysław Pawełczak. Time-Sensitive Intermittent Computing Meets Legacy Software, in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, doi : 10.1145/3373376.3378476.

[LA04] Chris Lattner et Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation, in *International Symposium on Code Generation and Optimization (CGO)*, 2004, doi : 10.1109/CGO.2004.1281665.

[LBB+21] Vincent Lostanlen, Antoine Bernabeu, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou et Mathieu Lagrange. Energy Efficiency Is Not Enough :Towards a Batteryless Internet of Sounds, in *Audio Mostly*, 2021, isbn : 978-1-4503-8569-5.

[LL17] Scott M Lundberg et Su-In Lee. A unified approach to interpreting model predictions, *Advances in neural information processing systems*, 2017.

[LM95] Y.-T.S. Li et S. Malik. Performance analysis of embedded software using implicit path enumeration, in *Workshop on Languages, compilers, & tools for real-time systems (LCTES)*, 1995, doi : 10.1145/216633.216666.

[LQZ+19] Fuyang Li, Keni Qiu, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yong Guan et Chun Jason Xue. Checkpointing-Aware Loop Tiling for Energy Harvesting Powered Nonvolatile Processors, *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019, doi : 10.1109/TCAD.2018.2803624.

[LZH+15] Qingan Li, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yanxiang He et Chun Jason Xue. Compiler Directed Automatic Stack Trimming for Ef-

ficient Non-Volatile Processors, in *Annual Design Automation Conference (DAC)*, 2015, DOI : 10.1145/2744769.2744809.

[Mai21] Andrea MAIOLI. ScEpTIC Repository, 2021, URL : https://bitbucket.org/neslabpolimi/sceptic/.

[Max18] MAXIM INTEGRATED. MAX77650/MAX77651 - Ultra-Low Power PMIC with 3-Output SIMO and Power Path Charger for Small Li+, 2018.

[MCL17] Kiwan MAENG, Alexei COLIN et Brandon LUCIA. Alpaca : Intermittent execution without checkpoints, *ACM on Programming Languages*, *OOPSLA*, 2017, DOI : 10.1145/3133920.

[ML18] Kiwan MAENG et Brandon LUCIA. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing, in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, ISBN : 978-1-939133-08-3.

[ML19] Kiwan MAENG et Brandon LUCIA. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints, in *Conference on Programming Language Design and Implementation (PLDI)*, 2019, DOI : 10.1145/3314221.3314613.

[ML20] Kiwan MAENG et Brandon LUCIA. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution, in *Conference on Programming Language Design and Implementation (PLDI)*, 2020, ISBN : 978-1-4503-7613-6, DOI : 10.1145/3385412.3385998.

[MM21] Andrea MAIOLI et Luca MOTTOLA. ALFRED : Virtual Memory for Intermittent Computing, in *Conference on Embedded Networked Sensor Systems*, 2021, DOI : 10.1145/3485730.3485949.

[MMA+21] Andrea MAIOLI, Luca MOTTOLA, Muhammad Hamad ALIZAI et Junaid Haroon SIDDIQUI. Discovering the Hidden Anomalies of Intermittent Computing, in *International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2021.

[MTD10] Vincenzo MUSOLINO, Enrico TIRONI et Politecnico DI MILANO. A Comparison of Supercapacitor and High-Power Lithium Batteries, in *Electrical Systems for Aircraft, Railway and Ship Propulsion*, 2010, DOI : 10.1109/ESARS.2010.5665263.

[NCA+23] Antonio Joia Neto, Adam Caulfield, Chistabelle Alvares et Ivan De Oliveira Nunes. $\mathcal{D}$iCA : A Hardware-Software Co-Design for Differential Check-Pointing in Intermittently Powered Devices, in *International Conference on Computer Aided Design (ICCAD)*, 2023, doi : 10.1109/ICCAD57390.2023.10323895.

[NNS+17] G. Nikolic, T. Nikolic, M. Stojcev, B. Petrovic et G. Jovanovic. Battery Capacity Estimation of Wireless Sensor Node, in *International Conference on Microelectronics (MIEL)*, 2017, doi : 10.1109/MIEL.2017.8190121.

[ODE+22] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka et Nicolas Tsiftes. The Contiki-NG open source operating system for next generation IoT devices, *SoftwareX*, 2022, doi : 10.1016/j.softx.2022.101089.

[PHL19] R.H.Y. Perdana, Hudiono Hudiono et A.F.N. Luqmani. Water Leak Detection and Shut-Off System on Water Distribution Pipe Network Using Wireless Sensor Network, in *International Conference on Advanced Mechatronics, Intelligent Manufacture and Industrial Automation (ICAMIMIA)*, 2019, doi : 10.1109/ICAMIMIA47173.2019.9223386.

[PMS21] Davide Pala, Ivan Miro-Panades et Olivier Sentieys. Freezer : A Specialized NVM Backup Controller for Intermittently Powered Systems, *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021, doi : 10.1109/TCAD.2020.3025063.

[RAA+14] Habib F. Rashvand, Ali Abedi, Jose M. Alcaraz-Calero, Paul D. Mitchell et Subhas Chandra Mukhopadhyay. Wireless Sensor Systems for Space and Extreme Environments : A Review, *Sensors Journal*, 2014, doi : 10.1109/JSEN.2014.2357030.

[RAP24] Hugo Reymond, Abderaouf Nassim Amalou et Isabelle Puaut. WORTEX : Worst-Case Execution Time and Energy Estimation in Low-Power Microprocessors Using Explainable ML, in *International Workshop on Worst-Case Execution Time Analysis*, 2024, doi : 10.4230/OASIcs.WCET.2024.1.

[RBB+24a] Hugo Reymond, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Isabelle Puaut et Erven Rohou. EarlyBird : Energy Belongs to Those Who Wake Up Early, in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'24)*, 2024.

[RBB+24b] Hugo Reymond, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Isabelle Puaut et Erven Rohou. SCHEMATIC : Compile-Time Checkpoint Placement and Memory Allocation for Intermittent Systems, in *International Symposium on Code Generation and Optimization (CGO)*, 2024, DOI : 10.1109/CGO57630.2024.10444789.

[RBM+18] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett et Alex S. Weddell. RESTOP : Retaining External Peripheral State in Intermittently-Powered Sensor Systems, *Sensors*, 2018, DOI : 10.3390/s18010172.

[RL14] Benjamin Ransford et Brandon Lucia. Nonvolatile memory is a broken time machine, in *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014, DOI : 10.1145/2618128.2618136.

[RSD+22] Emily Ruppel, Milijana Surbatovich, Harsh Desai, Kiwan Maeng et Brandon Lucia. An Architectural Charge Management Interface for Energy-Harvesting Systems, in *International Symposium on Microarchitecture (MICRO)*, 2022, DOI : 10.1109/MICRO56248.2022.00034.

[RSF11] Benjamin Ransford, Jacob Sorber et Kevin Fu. Mementos : system support for long-running computation on RFID-scale devices, in *International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011, DOI : 10.1145/1950365.1950386.

[RSG16] Marco Tulio Ribeiro, Sameer Singh et Carlos Guestrin. " Why should i trust you ?" Explaining the predictions of any classifier, in *International Conference on Knowledge Discovery and Data Mining*, 2016, DOI : 10.1145/2939672.2939778.

[RSR16] R. Ramya, G. Saravanakumar et S. Ravi. Energy Harvesting in Wireless Sensor Networks, in *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, 2016, DOI : 10.1007/978-81-322-2656-7_76.

[Spa17] Philip Sparks. The Route to a Trillion Devices, *White Paper, ARM*, 2017.

[SPS20] Nicholas Shoemaker, Ruzica Piskac et Mark Santolucito. Towards Checkpoint Placement for Dynamic Memory Allocation in Intermittent Computing, in *International Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2020, doi : 10.1145/3427764.3428323.

[SSD+22] Adnan Sabovic, Ashish Kumar Sultania, Carmen Delgado, Lander De Roeck et Jeroen Famaey. An Energy-Aware Task Scheduler for Energy Harvesting Battery-Less IoT Devices, *Internet of Things Journal*, 2022, doi : 10.1109/JIOT.2022.3185321.

[STS+17] Sophiane Senni, Lionel Torres, Gilles Sassatelli et Abdoulaye Gamatie. Non-Volatile Processor Based on MRAM for Ultra-Low-Power IoT Devices, *Journal on Emerging Technologies in Computing Systems*, 2017, doi : 10.1145/3001936.

[The19] The Australian Acoustic Observatory. Deployment Manual For Solar Powered Acoustic Sensors, 2019, url : https://acousticobservatory.org/deployment-information/.

[WH16] Joel Van Der Woude et Matthew Hicks. Intermittent Computation without Hardware Support or Programmer Intervention, in *Symposium on Operating Systems Design and Implementation OSDI*, 2016, isbn : 978-1-931971-33-1.

[YCY22] Eren Yildiz, Lijun Chen et Kasim Sinan Yildirim. Immortal Threads : Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers, in *Operating Systems Design and Implementation (OSDI 22)*, 2022, isbn : 978-1-939133-28-1.

[YMP+18] Kasım Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak et Josiah Hester. InK : Reactive Kernel for Tiny Batteryless Sensors, in *Conference on Embedded Networked Sensor Systems (SenSys)*, 2018, doi : 10.1145/3274783.3274837.

[YR20] Bahram Yarahmadi et Erven Rohou. Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems, in *Embedded Computer Systems : Architectures, Modeling, and Simulation*, 2020, doi : 10.1007/978-3-030-60939-9_12.

[YR21] Bahram Yarahmadi et Erven Rohou. So Far So Good : Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code, in *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2021, DOI : 10.1145/3493229.3493300.

[YWM05] Liyang Yu, Neng Wang et Xiaoqiao Meng. Real-Time Forest Fire Detection with Wireless Sensor Networks, in *International Conference on Wireless Communications, Networking and Mobile Computing*, 2005, DOI : 10.1109/WCNM.2005.1544272.

[YY20] Eren Yildiz et Kasim Sinan Yildirim. Defragmenting Energy Storage in Batteryless Sensing Devices, in *International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys)*, 2020, DOI : 10.1145/3417308.3430271.

[ZFL+17] Mengying Zhao, Chenchen Fu, Zewei Li, Qingan Li, Mimi Xie, Yongpan Liu, Jingtong Hu, Zhiping Jia et Chun Jason Xue. Stack-Size Sensitive On-Chip Memory Backup for Self-Powered Nonvolatile Processors, *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017, DOI : 10.1109/TCAD.2017.2666606.

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Modèle d'exécution conscient de l'énergie pour les systèmes intermittents

Mot clés : Calcul intermittent, récolte d'énergie, compilation, systèmes embarqués

Résumé : Les capteurs sans batterie exploitent l'énergie ambiante pour fonctionner dans des lieux isolés, où le remplacement des batteries est impossible. L'énergie récoltée, stockée dans un condensateur, est insuffisante pour une exécution continue, entraînant des pertes d'alimentation qui effacent les données volatiles (registres processeur et mémoire vive), empêchant la progression des programmes. Pour répondre à ce défi, nous présentons SCHEMATIC, une technique qui insère automatiquement des points de sauvegarde dans le code. Lorsqu'un de ces derniers est atteint, l'état du programme est sauvegardé dans une mémoire non volatile. Pour diminuer la taille des sauvegardes SCHEMATIC alloue certaines données en mémoire non volatile, de manière automatique. Ensuite, contrairement aux méthodes existantes qui reprennent l'exécution une fois le condensateur plein, nous proposons EARLYBIRD, qui ajuste le réveil pour reprendre l'exécution dès que suffisamment d'énergie a été récoltée pour atteindre le prochain point de sauvegarde. De cette manière, EARLYBIRD permet une reprise plus rapide de l'exécution et une réduction de la consommation d'énergie. À travers ces travaux, nous démontrons que les approches *conscientes de l'énergie* permettent une exécution sûre et efficace des programmes sur capteurs sans batterie.

Title: Energy-aware execution model for intermittent systems

Keywords: Intermittent computing, battery-less, energy harvesting, compilation, embedded systems

Abstract: Battery-less sensors harness ambient energy to operate in isolated locations where replacing batteries is impossible. The harvested energy, stored in a capacitor, is insufficient for continuous operation, leading to power losses that wipe volatile data (processor registers and RAM), thereby preventing program progression. In this thesis we present SCHEMATIC, a technique that automatically inserts checkpoints into the code. When a checkpoint is reached, the program state is saved in non-volatile memory. To reduce checkpoint size, SCHEMATIC automatically allocates certain data to non-volatile memory. Then, unlike existing methods that resume execution only once the capacitor is fully charged, we propose EARLYBIRD, which adjusts the wake-up threshold to resume execution as soon as enough energy has been harvested to reach the next checkpoint. This approach, allows for quicker execution resumption and reduced energy consumption. Through this work, we demonstrate that *energy-aware* approaches enable safe and efficient execution of programs on battery-less sensors.