Optimisation de la consommation des noeuds de réseaux de capteurs sans fil

Aurélien Buhrig

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA** dans le cadre de
l'École Doctorale d'Électronique, d'Électrotechnique, d'Automatique et de Traitement du Signal

présentée et soutenue publiquement par

Aurélien Buhrig

le 29 Avril 2008

Titre :

OPTIMISATION DE LA CONSOMMATION DES NŒUDS DE RÉSEAUX DE CAPTEURS SANS FIL

Directeur de thèse : M. Marc Renaudin
Co-directeur : M. Laurent Fesquet

Jury

M. Frédéric Pétrot,	Président
M. Michel Auguin,	Rapporteur
M. Olivier Sentieys,	Rapporteur
M. Dominique Barthel,	Examinateur
M. Antoine Fraboulet,	Examinateur
M. Laurent Fesquet,	Co-directeur
M. Marc Renaudin,	Directeur de Thèse

Remerciements

Ce travail de thèse est le fruit de trois années et demie de recherche effectuées au sein du laboratoire TIMA de l'Institut National Polytechnique de Grenoble, en collaboration avec France Télécom R&D. Je souhaite remercier M. Bernard Courtois, directeur du laboratoire TIMA au début de cette thèse, ainsi que Mme Dominique Borrione, actuelle directrice, pour leur accueil.

Je remercie chaleureusement Marc Renaudin, mon directeur de thèse, professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir donné tant de libertés sur ce sujet, et pour m'avoir fait confiance depuis toutes ces années pendant, et après la thèse.

Je remercie également Laurent Fesquet, professeur à l'institut National Polytechnique de Grenoble, pour avoir co-encadré mes travaux de thèse. Ses conseils et corrections apportés à ce manuscrit m'ont été précieux.

Je souhaite remercier les membres de mon jury de thèse :

M. Olivier Sentieys, professeur à l'université de Renne I à Lannion, pour avoir accepté de rapporter ma thèse.

M. Michel Auguin, directeur de recherche au CNRS à Sophia Antipolis, pour m'avoir fait l'honneur de participer à mon jury de thèse en tant que rapporteur malgré son emploi du temps très chargé.

M. Frédéric Pétrot, professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir fait l'honneur de présider mon jury de thèse.

M. Dominique Barthel, directeur du pôle Pacific chez France Télécom R&D à Meylan, pour avoir accepté d'être examinateur de ma thèse, mais surtout pour ses très précieux conseils lors des nombreuses réunions de travail sur les réseaux de capteurs.

M. Antoine Fraboulet, maître de conférence à l'INSA de Lyon, pour avoir accépté d'être examinateur de ma thèse.

Je tiens également à remercier tous les partenaires du projet ARESA qui ont contribué à valoriser ce travail.

Je remercie toutes les personnes du TIMA et du CMP qui m'ont épaulé pendant ces années : Isabelle, Patricia, Chantal, Ahmed (merci pour la basse !), Joëlle, Anne-Laure, Sophie M., Fred et Nicolas.

Une pensée pour tous les anciens du groupe CIS, en particulier Kamel, Manu (ahh, les soirées cloub...), Fabien et ses peluches, Dhanistha, Salim, Joao, Yann et tous les autres ! Merci aussi à Nico, mon vis-à-vis d'un temps, pour ces conseils et ces magnifiques moments passés à nous compter ces voyages au travers de ses photos.

Je souhaite remercier chaleureusement tous les membres actuels de l'équipe CIS, et plus particulièrement Gilles pour sa bonne humeur, Cédric et ses tongues pour son amnésie des prénoms, David et les bons moments passés en conf', Jérémie et sa Bretagne, Estelle pour sa gaité, Vivian et ses kebabs, Franck et ses petites blagues, Eslam et Saeed pour leur gentillesse, et les petits derniers : Taha, Rodrigo, et tous les autres !

Merci à Fraidy et Sophie, pour leurs conseils, leur gentilesse, leur joie de vivre et tous les bons moments passés en leur compagnie. Merci à Greg, (le claqueur) et ses rollers auto-guidés pour les bons moments passés lors des diverses soirées et les longues parties de Lazer Game.

Je tiens également à remercier Livier pour son humour francisé, les soirées mexicaines mémorables, sa délicieuse téquila, et tous les bons moments passés. Merci aussi à Sophie pour tous les moments partagés, sa créativité, les soirées, les répèt... Un grand merci, à Bertrand, pour son "aide" à la rédaction, son soutien lors du grand saut (le mien était vraiment mieux), et toutes les très bonnes soirées passées, et à Yannick, pour ses conseils en musique (pour moins vous casser les oreilles), les parties de tennis, et tant de souvenirs.

Je remercie ma famille, Sébastien, Quentin, Josselin et plus particulièrement mes parents, pour m'avoir toujours soutenu.

Pour terminer, je tiens à remercier du fond du cœur ma petite femme Audrey, pour son soutien de tous les instants et pour tous les moments chaleureux qu'elle m'apporte au quotidien.

Résumé

Les réseaux de capteurs sans fil, de part l'absence d'infrastructure de communication, posent de nombreux défis de conception. Ils doivent en particulier capter les informations provenant de l'environnement, traiter les données acquises, recevoir et retransmettre celles-ci tout en ayant une durée de vie qui doit pouvoir atteindre plusieurs dizaines d'années selon les applications, sans intervention extérieure. Dans ce contexte, il est nécessaire d'optimiser la consommation d'énergie à tous les niveaux de conception, en définissant des protocoles d'auto-organisation et des protocoles de communication efficaces en énergie, ou en définissant une plateforme logicielle et matérielle faible consommation.

Ce travail de thèse propose de réduire la consommation d'énergie de la partie numérique d'un nœud de réseau de capteurs sans fil. Pour cela, la logique asynchrone quasi-insensible aux délais très faible consommation d'énergie est utilisée. Associée à un partitionnement logiciel/matériel judicieux de l'application, celle-ci permet de s'affranchir du surcoût de communication entre le processeur et ses périphériques en remplaçant, lorsque c'est possible, le mécanisme d'interruptions par l'utilisation de communications synchrones utilisées intrinsèquement par le matériel asynchrone. Par ailleurs, un moyen efficace de réduire la consommation d'énergie est d'utiliser la technique d'adaptation dynamique de la vitesse et de la tension du processeur (DVS). Pour cela, un coprocesseur est spécifié. Il prend en entrée une consigne de vitesse définie par le logiciel et contrôle la tension d'alimentation du microcontrôleur afin d'asservir sa vitesse.

De plus, dans le contexte des réseaux de capteurs sans fil où les tâches sont généralement de très faible complexité par rapport à un système temps réel classique, il est important de contrôler efficacement la vitesse du processeur. Ce travail propose une méthodologie de simulation permettant d'évaluer l'impact sur la consommation d'énergie des algorithmes d'ordonnancement temps réel de tâches et de gestion du DVS, ainsi que les modèles de tâches s'exécutant sur le système.

Ce travail contribue donc à définir des solutions logicielles et matérielles exploitant le matériel sans horloge pour réduire la consommation des systèmes embarqués communicants.

Table des matières

Liste des figures

Liste des tableaux

Liste des Algorithmes

,

Introduction générale

Les réseaux de capteurs et d'actionneurs, qui seront bientôt techniquement et économiquement viables pour une large gamme d'applications, permettent de répondre d'une nouvelle manière aux problématiques récurrentes telles que, par exemple, la gestion de biens de production, la surveillance environnementale, l'agriculture de précision, les infrastructures pour la communication ambiante, etc.

Nous considérons, dans le cadre de cette thèse, de grands réseaux de capteurs comportant typiquement plusieurs centaines ou milliers de nœuds. Ces nœuds, les capteurs, sont considérés essentiellement comme "jetables" et redondants, c'est-à-dire que la mort ou la défaillance de l'un d'entre eux ne doit pas appeler systématiquement à son remplacement. De plus, ces nœuds, physiquement dispersés dans l'espace, sont en nombre tel qu'il n'est pas facile ou possible de les recharger périodiquement. On suppose donc que les nœuds disposent initialement d'une réserve d'énergie finie et qu'ils meurent lorsque cette énergie est épuisée.

Dans ce contexte, l'optimisation de la consommation n'est pas seulement un facteur accessoire de confort pour l'utilisateur, mais un facteur essentiel pour l'acceptabilité d'un réseau de capteurs. Cette optimisation de l'énergie doit avoir lieu à tous les niveaux de conception du système. En particulier, il est indispensable d'optimiser la partie numérique des nœuds composant les réseaux de capteurs sans fil, qui inclut notamment des périphériques numériques, le microcontrôleur, mais aussi le logiciel applicatif qui s'exécute sur celui-ci ainsi que les différents algorithmes de gestion de l'énergie.

L'optimisation de la consommation de la partie numérique comprend plusieurs aspects. Tout d'abord, il est nécessaire d'utiliser du matériel ultra faible consommation. La technologie asynchrone quasi-insensible aux délais permet naturellement de limiter l'activité électrique des circuits. Son utilisation permet ainsi de s'affranchir de techniques telles que le "clock gating" ou les modes de veilles profondes mises en place pour la faible consommation dans le cadre d'une conception synchrone.

Par ailleurs, définir un partitionnement judicieux de l'application est un moyen très efficace de réduire la consommation d'énergie du système. En effet, certaines fonctions de l'application, telles que le codage et le décodage canal ou le chiffrement des données par exemple, sont coûteuses en énergie lorsqu'elles sont exécutées en logiciel. L'implantation en matériel permet alors de réduire la dépense

d'énergie. Cependant, dans le contexte d'un réseau de capteurs, il est nécessaire de prendre en considération le coût énergétique induit par la synchronisation du matériel et du logiciel qui peut être très important dans le cas d'interruptions et peut même représenter la majorité de l'énergie consommée par la tâche partitionnée. Ce type de synchronisation coûtant cher, il est nécessaire de définir des interfaces logiciel-matériel permettant de synchroniser le processeur et ses périphériques à coût énergétique nul.

Un autre moyen efficace de réduire la consommation d'énergie dans un système embarqué est d'utiliser la technique de l'adaptation dynamique en tension (DVS) permettant de modifier la vitesse du système en fonction des contraintes logicielles. Cependant, dans le cadre d'une conception à base de processeur asynchrone où il n'y a pas d'horloge donnant facilement l'information de temps, il convient de trouver un moyen efficace de mesurer et de contrôler la vitesse. Pour se faire, nous avons spécifié et simulé un coprocesseur DVS permettant d'asservir la vitesse du processeur en fonction d'une consigne logicielle. De cette manière, il n'est pas nécessaire de considérer des caractérisations pires cas du processeur. Au contraire, cette approche permet de conserver un comportement en moyenne et de réduire la consommation d'énergie.

La technique de gestion du DVS suppose dès lors de calculer la consigne de vitesse au niveau logiciel. Cette consigne dépend des contraintes temps réel des différentes tâches présentes sur le système. Il convient alors d'évaluer les différentes stratégies possibles au niveau système et algorithmique pour contrôler la vitesse du processeur de la manière la plus efficace en énergie.

Ce sont ces aspects de l'optimisation de la consommation des nœuds de réseaux de capteurs sans fil que nous proposons d'aborder dans ce travail de thèse, qui s'inscrit dans le contexte d'un projet multi-partenaires grenoblois sur les réseaux de capteurs et actionneurs, initié et financé par France Télécom Division R&D avec les laboratoires LIG/LSR, Verimag et TIMA.

Ce manuscrit est divisé en sept chapitres. Le chapitre 1 décrit les réseaux de capteurs sans fil. Il introduit le contexte de l'étude, quelques applications et montre les problématiques et les défis actuellement posés.

Le chapitre 2 présente les concepts de base régissant le fonctionnement des circuits asynchrones et montre l'intérêt de l'utilisation d'une telle technologie dans les systèmes embarqués, et plus particulièrement dans le domaine des réseaux de capteurs.

Une architecture typique d'un nœud de réseaux de capteurs est présentée dans le chapitre 3. Les consommations typiques de différents organes du système sont exposées et comparées. Au niveau logiciel, plusieurs systèmes d'exploitation sont décrits pour montrer l'avantage, du point de vue de la consommation, d'utiliser des systèmes dédiés aux réseaux de capteurs.

Le but du chapitre 4 est d'évaluer l'intérêt du partitionnement logiciel-matériel. En particulier, ce chapitre met en évidence au travers d'exemples, l'importante consommation d'énergie mise en jeu avec des mécanismes de synchronisation traditionnels entre le processeur et ses périphériques – comme les interruptions – et propose une solution à base de communications synchrones.

Différentes méthodes existantes pour réduire la consommation d'énergie dans les systèmes embarqués sont de plus abordées au chapitre 5. En particulier, la technique de l'adaptation dynamique en tension (DVS) est détaillée, ainsi qu'un moyen de la mettre efficacement en œuvre avec un processeur asynchrone, grâce notamment à l'utilisation du coprocesseur DVS évoqué précédemment permettant de gérer la vitesse du microcontrôleur en fonction d'une consigne logicielle en vitesse.

Pour calculer cette vitesse, il est nécessaire de définir des algorithmes d'ordonnancement et de gestion du DVS, ainsi que des modèles de tâches qui minimisent l'énergie consommée. Le chapitre 6 s'occupe de formaliser les différentes approches et de comparer différents algorithmes et modèles afin de trouver la solution la moins consommatrice d'énergie, ainsi que son implantation dans un contexte de réseaux de capteurs sans fil.

Enfin, le chapitre 7 montre comment concilier les deux approches précédemment citées pour réduire la consommation d'énergie : le partitionnement logiciel-matériel avec synchronisation à base de communications synchrones, et la gestion temps réel du DVS. En particulier, il permet de définir un moyen simple de réduire la consommation globale du système en appliquant la technique du DVS sur les périphériques matériels.

Enfin, les différents points abordés dans ce manuscrit sont synthétisés dans une conclusion qui présente également quelques perspectives pour la suite de ce travail.

Chapitre 1

Les réseaux de capteurs sans fil

Ce chapitre s'inspire en partie de la proposition d'un projet ANR RNRT numéro ANR-05-RNRT-01703 appelé ARESA [24].

1.1 Contexte

Dans cette thèse, nous nous intéressons aux réseaux maillés sans fil constitués d'un grand nombre de petits dispositifs électroniques (systèmes enfouis) répartis physiquement dans l'espace. Pour avoir une utilité pratique, ces dispositifs interagissent avec le monde physique au moyen de capteurs ou d'actionneurs. Au minimum, ils sont attachés à un objet physique et peuvent transmettre leur identifiant ainsi qu'un ensemble de données (mesures, alarmes, notifications) sur requête arrivant par le réseau. En fonction des circonstances, les capteurs sont enchâssés dans des matériaux de construction, épandus dans la nature par un avion ou installés manuellement par un être humain ou un robot. Par conséquent, dans la plupart des cas, les capteurs sont difficiles d'accès et donc coûteux à changer, à recharger et à administrer. Ainsi, pour que les réseaux de capteurs soient rentables, la durée de leur bon fonctionnement doit être maximale. Le réseau permet de drainer les informations collectées par les dispositifs vers un ou plusieurs points de collecte ou de propager celles fournies par une ou plusieurs sources, ces points de collecte et sources étant généralement regroupés sous forme de points d'accès à un réseau global informatique classique. La figure 1.1 présente un réseau de systèmes enfouis dans une architecture de communication à domicile qui intègre d'autres types de réseaux et l'interconnexion avec l'extérieur.

Souvent l'espace à couvrir par ces systèmes enfouis est relativement grand ce qui empêche l'utilisation d'un nœud central pour piloter tous les dispositifs. L'organisation du réseau doit alors passer par des approches qui ne reposent pas sur une infrastructure fixe – des réseaux sans fil ad hoc. Un nombre quelconque d'objets équipés de radios et de protocoles adéquats suffit pour former un réseau ad hoc qui constitue la solution alternative aux réseaux classiques sans fil quand l'accès à l'infrastructure est impossible ou inefficace. De plus, en s'organisant de façon spontanée et totalement distribuée, les réseaux

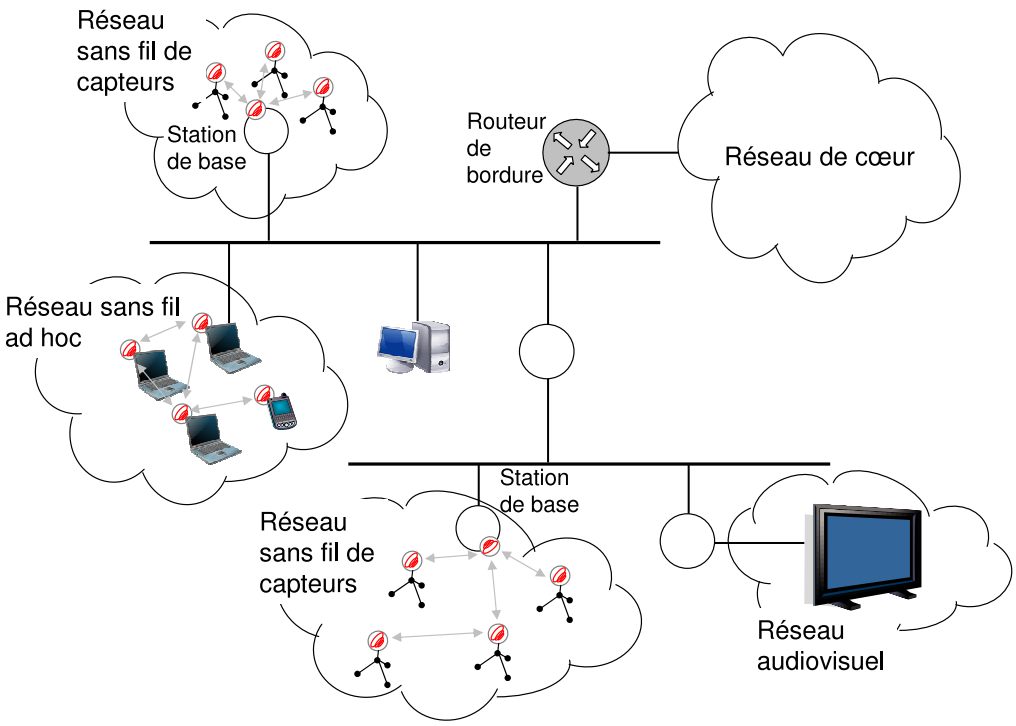


FIG. 1.1 – Réseau de systèmes enfouis.

ad hoc offrent plus de flexibilité (déploiement et redéploiement rapides) et de robustesse (tolérance aux pannes).

1.2 Applications

Avec les progrès technologiques récents permettant la réalisation d'objets très petits, consommant peu d'énergie et équipés de capteurs/déclencheurs, l'idée d'utiliser un réseau ad hoc de capteurs [38] pour accomplir certaines tâches applicatives a commencé à devenir réaliste. On peut notamment citer le "monitoring" d'habitats naturels [47], le contrôle de la climatisation des bâtiments ou la surveillance militaire [35].

Certaines des applications citées ont des enjeux économiques importants [60]. Par exemple, le contrôle fin de la climatisation des bâtiments à l'Université de Californie à Berkeley permet d'économiser de très grandes quantités d'énergie souvent gaspillées à cause d'un contrôle non optimal de la température.

On peut également mentionner des réseaux de capteurs qui sont déployés aux États-Unis : l'US Navy opère des réseaux ad hoc d'étiquettes électroniques entre les conteneurs dans les soutes de ses navires. Expérimentalement, des réseaux ad hoc de capteurs sont déployés par l'US Army pour détecter les passages de camions et de chars dans des zones désertiques, par détection sonore ou magnétique.

Plus pacifiquement, des réseaux ad hoc de capteurs surveillent l'environnement dans des réserves naturelles (à Great Duck Island et dans le parc national des volcans à Hawaï), détectent et avertissent

d'un risque de gel dans des vignes de l'état de Washington, ou peuvent surveiller et contrôler le trafic routier pour avertir les conducteurs en cas d'accidents, compter les places de parking restantes, etc.

Les activités commerciales autour des réseaux de capteurs sans fil se développent de plus en plus. Par exemple, la société française Coronis Systems a déployé plus de 1 300 000 nœuds connectés en réseau ad hoc à la fin de l'année 2007, principalement du "metering" dans le domaine du relevé de compteurs d'eau.

À plus long terme, des réseaux de capteurs/actionneurs seront une brique cruciale au développement de l'intelligence ambiante, où l'environnement sera capable de comprendre l'activité de l'utilisateur et lui présenter les interfaces de communication adaptées à chaque instant.

1.3 Problématiques

Le domaine des réseaux de capteurs est récent et fait l'objet d'une activité de recherche grandissante essentiellement à cause de nouveaux problèmes et de défis que ce type d'équipement pose. En effet, les systèmes enfouis utilisés dans ces réseaux ont des caractéristiques différentes par rapport aux systèmes embarqués étudiés depuis de nombreuses années et ils imposent de nouvelles contraintes sur toutes les parties : matériel, protocoles de communication, logiciels système et applicatifs. Comme il a été mentionné dans l'introduction, la contrainte essentielle est celle de la consommation d'énergie qui conditionne l'acceptabilité de tels réseaux.

La prise en compte de l'énergie est donc une contrainte forte qui n'est pas spécifique au composant ou au protocole mais qui apparaît bien comme une problématique récurrente et transversale. Il est donc nécessaire de repenser tous les aspects : le processeur doit consommer le moins d'énergie possible, la couche de transmission radio doit être spécifique, les protocoles de communication doivent permettre des périodes d'inactivité prolongée pour économiser de l'énergie. Le grand nombre de nœuds déployés impose des architectures réseau qui permettent le passage à l'échelle. Si on ajoute à ces aspects, la simplicité de chaque dispositif, il faut rechercher des structures réseau sur la base des mécanismes d'auto-configuration et d'autonomie qui assurent un fonctionnement automatique sans intervention humaine. Par conséquent, il est nécessaire de concevoir de nouvelles architectures matérielles et logicielles, ainsi que de nouveaux protocoles de communication et structures d'organisation pour des réseaux de capteurs intelligents afin de rendre les réseaux de capteurs plus ouverts et facilement utilisables par un utilisateur équipé d'un terminal mobile.

Les contraintes qui portent sur ce type de système enfoui – telles que la faible énergie, la dépendance forte de la consommation sur l'utilisation de la radio, la petite taille ou encore le faible coût – ne sont donc pas les mêmes que pour des systèmes embarqués critiques (temps réel dur, sûreté de fonctionnement). Les méthodes qui ont été développées pour les systèmes critiques ne sont pas directement applicables. La complexité provient ici de la taille (quelques milliers de noeuds) et des inter-dépendances fortes entre les différentes couches du réseau (nœuds de base, couches protocolaires, logiciel d'application).

L'optimisation globale de la consommation d'énergie dans un réseau ad hoc de capteurs requiert des travaux d'optimisation à plusieurs niveaux :

- au niveau de la réalisation physique des nœuds : technologie des semi-conducteurs, circuiteries électroniques (par exemple, circuits asynchrones), microarchitecture des sous-systèmes, architecture globale du noeud, etc.
- au niveau des algorithmes d'accès au canal radio partagé et de routage des messages dans le réseau de capteurs sans fil.
- au niveau de l'organisation du réseau afin de permettre un comportement auto-organisant et autonome.
- au niveau "application", par le traitement dans le réseau des informations recueillies par les capteurs, plutôt que de les propager sous forme brute vers le point de collecte.

Ce cadre montre aussi la nécessité de s'attaquer au problème de la modélisation globale et du prototypage virtuel des réseaux de capteurs, de manière à fournir un environnement de développement dans lequel les solutions d'économie d'énergie des différents niveaux peuvent être modélisées et assemblées avec un modèle du monde physique. Cette approche permet de plus d'observer très tôt dans le cycle de développement les conséquences d'un choix architectural, logiciel ou matériel particulier.

1.3.1 Architectures logicielles et matérielles à très faible consommation

Compte tenu du taux d'activité très faible d'un nœud au sein du réseau, il est souhaitable du point de vue de la consommation, et donc de la durée de vie du réseau, de minimiser l'activité électrique des circuits, notamment lors des périodes d'inactivité. Il faut donc que le nœud soit réactif aux stimuli générés par son environnement (radio, capteur, récupération d'énergie, etc.) et consomme très peu d'énergie en période de veille. Pour cela, plutôt qu'une architecture matérielle basée sur une approche synchrone, comme celle utilisée par la plupart des nœuds actuels, il est souhaitable d'adopter une architecture basée sur des composants sans horloge, dits asynchrones (voir chapitre 2).

Par ailleurs, implanter un traitement en matériel permet de réduire considérablement le coût énergétique de l'opération, tout en augmentant les performances. Cependant, ce partitionnement logiciel/matériel nécessite de prendre en considération le coût de la synchronisation entre le logiciel et le matériel afin de concevoir des interfaces adaptées entre le processeur et ses périphériques.

Au niveau système, la charge du processeur peut varier de manière importante dans le temps. Pour réduire la consommation, il est alors possible de ralentir le processeur lorsque cela est possible, en utilisant la technique du DVS. Il se pose alors le problème de comment gérer efficacement cette technique dans le contexte des réseaux de capteurs sans fil, et de comment la faire fonctionner conjointement avec du matériel dédié issu du partitionnement par exemple.

Au niveau de la couche physique, les radios actuelles utilisent des procédés de modulation/démodulation en bande étroite, qui nécessitent des oscillateurs, amplificateurs, mélangeurs et filtres, c'est-à-dire

des circuits linéaires qui requièrent des courants de polarisation constants, que le circuit soit ou non en train d'émettre ou de recevoir activement des informations. Avec l'ouverture par la FCC aux techniques ultra-large bande, des dispositifs de transmission par impulsions ultra-brêves laissent envisager la possibilité de réaliser des circuits radio faible consommation [20] idéaux pour les réseaux de capteurs.

1.3.2 Méthodes d'accès au canal et protocoles de routage

Au niveau de la couche "lien", les sources de dissipation (gaspillage) d'énergie les plus importantes sont [43] :

1. L'écoute passive : dans les protocoles d'accès au médium classiques (ex : IEEE 802.11 DCF), l'émetteur-récepteur radio est prêt à recevoir (allumé) en permanence. Ce mode "prêt à recevoir" consomme beaucoup d'énergie. Cette énergie est simplement gaspillée s'il n'y a aucune transmission sur le canal. Ce problème est nettement plus visible dans les réseaux ad hoc de capteurs où le canal est libre la plupart du temps.
2. Les collisions : l'énergie fournie pour envoyer une trame est simplement perdue quand cette trame est cassée en cours de route par une collision.
3. L'écoute inutile ("overhearing") : est définie par la réception et décodage de trames inutiles. On peut classer l'écoute inutile en trois catégories :
   - Réception et décodage d'une trame destinée à un autre nœud.
   - Réception et décodage de trames redondantes (essentiellement envoyées en broadcast) dont le contenu est sans intérêt pour le routage ou pour l'application car il a déjà été déjà reçu par le nœud considéré.
   - Réception et décodage de trames lorsqu'on est en attente de la libération du canal (c'est typiquement le cas du protocole CSMA pour l'accès au canal).
4. Le surcoût des trames de contrôle : quelques protocoles d'accès au médium (ex : IEEE 802.11 DCF) utilisent des trames de contrôle (ex : RTS/CTS) pour l'évitement de collisions. Ces trames de contrôle ne véhiculent aucune information utile pour les applications bien que leurs échanges (envoi/réception) coûtent de l'énergie.

L'élaboration d'une couche "lien" efficace adaptée aux réseaux de capteurs est donc primordiale pour réduire la consommation d'énergie et augmenter la durée de vie du réseau.

La consommation d'énergie est minimisée de plus par le choix d'algorithmes d'accès au canal radio (partagé) et de routage des messages dans le réseau. Par ailleurs, il est généralement considéré que la couche réseau et la couche lien doivent communiquer plus étroitement entre elles que dans le modèle classique de couches isolées. Par exemple, en radiofréquence, un lien avec un nœud distant peut parfois être établi, au prix certes d'une dépense énergétique importante, si la couche réseau estime ce lien fortement désirable (par exemple, dernier lien subsistant entre deux sous-réseaux). Réciproquement, l'usage d'une puissance élevée pour maintenir en activité un lien non essentiel peut créer de l'interférence avec un autre lien voisin plus important pour le réseau. Ainsi, il est nécessaire d'élaborer de nouveaux protocoles qui tiennent compte de la consommation d'énergie.

1.3.3 Architectures logicielles des applications

À la couche application, le réseau ad hoc de capteurs peut être vu comme une base de données distribuées [9] où les capteurs répondent aux requêtes formulées par le ou les collecteurs de données. Ceci rend la majorité des communications en multipoints : plusieurs points vers un ou plusieurs points.

Pour permettre une communication efficace en multipoints, il est nécessaire de concevoir des protocoles efficaces de manière à effectuer un traitement sur les données à l'intérieur du réseau (nœuds intermédiaires). Ceci peut être réalisé typiquement par agrégation de données pour réduire le volume d'information à envoyer à travers le réseau et permet donc d'économiser de l'énergie, particulièrement dans les réseaux denses.

1.3.4 Auto-configuration, auto-organisation, auto-gestion, auto-réparation

Il s'agit de l'étude des mécanismes d'autonomie des réseaux de capteurs et d'objets communicants. L'auto-configuration consiste avant tout à permettre la communication des nœuds, et donc l'affectation d'adresses aux interfaces réseaux, la diffusion des préfixes des sous-réseaux et des passerelles d'interconnexion. Il s'agit ensuite de diffuser les informations comme le DNS aux nœuds du réseau. Les problématiques de découverte de services apparaissent immédiatement après la mise en place de la topologie spontanée. En revanche, l'auto-configuration ne doit pas se faire au détriment de la sécurité.

L'auto-organisation cherche à structurer la topologie du réseau en tirant partie des propriétés des noeuds tels que l'énergie résiduelle, la densité, etc. Par exemple, est-il préférable de proposer une architecture à plat du réseau ou au contraire est-il nécessaire de proposer des mécanismes regroupant les nœuds suivant des critères comme les services, les interfaces radio, les capacités, l'énergie disponible, etc ? L'auto-organisation doit également répondre à la prise en compte efficace de la dynamicité du réseau : il faut minimiser les conséquences des disparitions de nœuds et accepter le déploiement de nouveaux capteurs sans nécessairement reconstruire la topologie du réseau.

L'auto-gestion a pour objectif de proposer une supervision autonome : une fois un réseau spontané auto-configuré et auto-organisé, il est nécessaire de fournir des algorithmes et des protocoles permettant sa surveillance, sa maintenance réactive. Il faut investir sur les solutions de collecte d'informations remontées par les couches inférieures, les analyser, prendre les décisions.

L'auto-réparation permet de pallier certaines imperfections auxquelles sont exposés des noeuds dans un réseau de capteurs. D'abord, les pannes ou la mort de certains nœuds induisent également des variations accidentelles de topologie du réseau. Leur déploiement peut aussi être imparfait, par exemple, certaines zones ne sont pas couvertes par des capteurs, des nœuds sont trop éloignés pour pouvoir communiquer ou des obstacles empêchent cette communication. Au cours de leur vie, des capteurs peuvent tomber en panne, ne pas avoir des ressources suffisantes pour fonctionner, ou subir des attaques qui les privent de certaines capacités (attaques de déni de service). Enfin, dans certains types de réseau, des

nœuds sont mobiles (capteurs portatifs pour le monitoring de l'état de santé, pour des opérations de sauvetage ou sur des champs de bataille) ce qui peut modifier la topologie du réseau et sa capacité à communiquer. Il s'agit donc de concevoir des algorithmes pour assurer le bon fonctionnement du réseau même en cas de la variabilité du réseau et de sa topologie.

Il faut donc étudier de nouvelles structures de réseaux qui permettent l'auto-configuration, et envisager l'autonomie. Si le déploiement d'un réseau de capteurs à large échelle (de l'ordre de 10 000 noeuds, par exemple) est envisageable, la configuration des équipements, l'élaboration d'une topologie efficace pour la mise en place de services ne peut pas se baser sur des interventions humaines. Il est donc nécessaire de définir des méthodes d'auto-configuration permettant aux noeuds de communiquer dans le voisinage puis de mettre en place des structures auto-organisantes. De telles structures visent à sélectionner des noeuds stratégiques dans l'environnement et/ou des liens radio de qualité importante. Les structures proposées doivent offrir une robustesse au changement de topologie et faciliter la mise en place de nouveaux services. Une fois de plus, les algorithmes proposés tiendront compte de l'optimisation de la consommation d'énergie. Ces mécanismes d'organisation doivent de plus prendre en compte le problème de la consommation énergétique qui résulte d'un échange intense de messages de service afin de réduire la consommation.

1.3.5 Architectures d'interconnexion

Les réseaux de capteurs peuvent être homogènes (constitués de dispositifs aux capacités physiques identiques) ou hétérogènes (les nœuds peuvent être d'un type parmi plusieurs ayant des capacités matérielles différentes, par exemple, des dispositifs plus complexes ayant des capacités de traitement et de communication supérieures à celles des nœuds ordinaires). Dans la seconde approche, le réseau sera habituellement considéré comme hiérarchisé, tandis que dans la première, la structure pourra être soit plate soit hiérarchisée par spécialisation logicielle permanente ou temporaire de certains nœuds (élection d'une tête de cluster, figure 1.2).

Dans un réseau où certains nœuds disposent de l'énergie et de la capacité de traitement suffisante, on peut envisager leur interconnexion sans les contraintes discutées précédemment. Il faut alors concevoir une architecture d'interconnexion des têtes de clusters (des points de collecte) qui pilotent des réseaux de capteurs par un réseau sans fil ad hoc. Cette interconnexion n'est pas limitée par les mêmes contraintes qu'un réseau de capteurs, néanmoins, elle nécessite certaines propriétés (organisation spontanée, choix de chemins, utilisation de chemins multiples, robustesse).

1.3.6 Sécurité

Dans un réseau de capteurs physiquement dispersés, la protection physique des nœuds du réseau n'est pas assurée. En plus des attaques à distance classiquement étudiées dans les réseaux, on doit aussi considérer les attaques physiques, les tentatives de substitution de nœuds ou de détournement de fonction

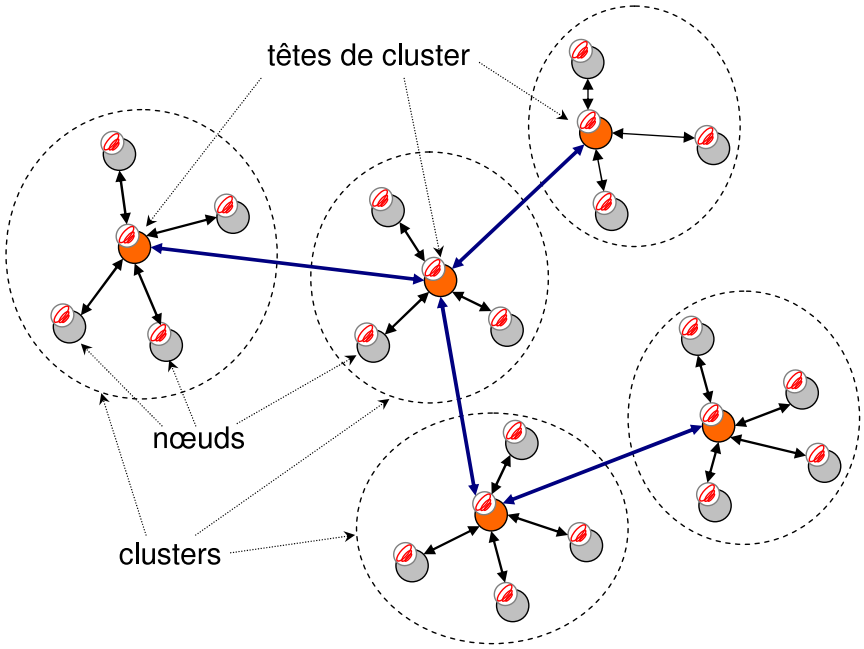


FIG. 1.2 – Exemple d'organisation hiérarchisée en clusters.

du réseau par noyautage de nœuds ennemis. Cet aspect doit être pris en compte selon le type d'application considéré.

1.3.7 Modélisation globale des systèmes de capteurs

La compréhension du fonctionnement global du réseau demande une modélisation informatique précise. La modélisation doit prendre en compte à la fois les noeuds individuels, la structure du réseau, les protocoles, et finalement le monde physique qui agit sur les capteurs.

Une telle modélisation est à la base de deux types d'outils :
– des outils d'analyse fonctionnelle ou de performances,
– des outils de conception et développement des applications à base de réseaux de capteurs.

Il est possible d'utiliser la même modélisation, à la fois en simulation et prototypage virtuel, et comme base des analyses formelles. Ceci est possible si on décrit le système, non pas dans un formalisme mathématique, mais dans un langage réactif exécutable muni d'une sémantique formelle [65, 66].

En ce qui concerne les outils de conception et de développement, il est pratique d'avoir un environnement pour aider le concepteur à effectuer des choix à tous les niveaux : technologie des capteurs eux-mêmes, architecture de l'application logicielle, couches protocolaires. Pour cela, l'environnement offre principalement des fonctions de simulation de l'application complète, en assemblant des modèles de consommation des noeuds, un modèle physique de l'environnement qui agit sur les capteurs, et les descriptions logicielles des différentes couches protocolaires.

En ce qui concerne les outils d'analyse, il est nécessaire de choisir des abstractions qui rendent le

modèle analysable tout en restant réaliste vis-à-vis des objets physiques réels. Lorsqu'on modélise le monde physique ou la consommation énergétique des noeuds, on fait nécessairement des abstractions puisqu'il s'agit de traiter par des outils informatiques des phénomènes physiques. Lorsque l'on inclut dans un modèle global les algorithmes des protocoles, on peut être également conduit à effectuer des abstractions pour maintenir le modèle dans le domaine où les propriétés sont décidables.

1.4 Conclusion

Les réseaux de capteurs ouvrent la voie à des applications variées dans de nombreux domaines et permettront, à plus ou moins long terme, le développement de l'intelligence ambiante, où l'environnement sera capable de comprendre l'activité de l'utilisateur et lui présenter les interfaces de communication adaptées.

Avant d'atteindre ce niveau de maturité, il est nécessaire de développer des plateformes matérielles et des protocoles de communication et d'auto-organisation efficaces en énergie. En effet, dans un réseau de capteurs, l'énergie est la contrainte essentielle qui modifie profondément toutes les hypothèses sur lesquelles sont basées des systèmes enfouis et des réseaux sans fil. L'énergie étant finie pour chaque nœud, l'optimisation de la consommation d'énergie n'est pas seulement un facteur accessoire de confort pour l'utilisateur, mais un facteur essentiel pour la faisabilité et l'acceptabilité d'un réseau de capteurs. Par exemple, pour que les nœuds puissent être noyés dès leur fabrication dans des plaques de placoplâtre ou dans l'épaisseur du papier peint, il faut que la durée de vie soit supérieure à celle généralement admise pour ces matériaux.

Pour économiser l'énergie du réseau, il est nécessaire d'adopter une approche verticale. En effet, il n'est pas souhaitable d'étudier une architecture de noeuds ou de réseaux sans tenir compte de ses performances, de l'environnement et de son impact sur les applications mais il faut au contraire bien repenser une architecture globale de réseaux de capteurs où la maîtrise de l'énergie est au coeur des préoccupations pour, notamment, augmenter la durée de vie du réseau.

Pour cela, il faut s'interroger sur les architectures logicielles et matérielles optimisant l'énergie des éléments de base : les capteurs intelligents. À partir de ces composants, de nouveaux protocoles de communication doivent être élaborés en tenant compte de l'environnement radio et en minimisant l'énergie dissipée en communication. Puis il est nécessaire de concevoir des algorithmes distribués mettant en oeuvre des mécanismes pour l'autonomie du réseau (auto-configuration, auto-organisation, auto-réparation) où le compromis durée de vie/performances/efficacité devra être établi.

C'est ce que propose de réaliser le projet ANR RNRT numéro ANR-05-RNRT-01703, ARESA [24], qui regroupe plusieurs laboratoires universitaires – le LIG-LSR, le TIMA et VERIMAG à Grenoble, le CITI à l'Insa de Lyon – ainsi que l'opérateur France Télécom R&D Meylan et l'entreprise Coronis Systems spécialisée dans les réseaux de capteurs sans fil. Ce projet, qui se terminera en juin 2009, vise plus spécifiquement à :

- explorer de nouvelles architectures logicielles et matérielles évènementielles optimisées pour la très faible consommation,
- proposer de nouveaux protocoles au niveau MAC et réseau qui tiennent compte de la consommation de l'énergie,
- trouver de nouveaux protocoles applicatifs adaptés à la fusion et à l'agrégation de données,
- étudier de nouvelles structures de réseaux qui permettent l'auto-configuration, l'auto-organisation et l'auto-réparation,
- étudier les besoins d'un environnement de développement d'une application à base de réseaux de capteurs, ainsi que fournir des outils de modélisation globale et de validation,
- valider l'ensemble des propositions sur la base d'un scénario applicatif.

Chapitre 2

Les circuits asynchrones

2.1 Introduction

Lorsque la conception de circuits numériques a débuté, il n'existait pas de distinction entre circuit synchrone et asynchrone. Les circuits synchrones correspondent à une classe restreinte de circuits qui sont séquencés par un signal périodique uniformément distribué : l'horloge. Au contraire les circuits asynchrones sont des circuits dont le contrôle est assuré par une toute autre méthode que le recours à un signal d'horloge global. Le contrôle se fait de manière locale par une synchronisation entre blocs fonctionnels. Très vite le style de conception synchrone s'est imposé pour répondre à des besoins de calcul croissants et pour s'adapter à une technologie encore bridée.

Pourtant l'étude des circuits asynchrones a commencé au début des années 1950 pour concevoir des circuits à relais mécaniques. En 1956, Muller et Bartky, de l'université de l'Illinois, ont travaillé sur la théorie des circuits asynchrones. Huffman est le premier à concevoir des machines à états asynchrones en 1968 avec ses travaux en « switching theory ». Ces travaux ont ensuite été étendus par Huffman lui-même, Muller, Unger et Mac Cluskey. Muller fut le premier à proposer d'associer un signal de validité aux données, introduisant ainsi un protocole de communication quatre phases. En 1966, le « Macromodule Project » lancé par W.A. Clark de l'université de Washington, St Louis [15] montre qu'il est possible de concevoir des machines spécialisées complexes par simple composition de blocs fonctionnels asynchrones. Par la suite, Seitz introduit un formalisme proche des réseaux de Petri pour concevoir des circuits asynchrones, ce qui aboutit à la construction du premier calculateur « dataflow » (DDM-1) [18]. Enfin, en 1989, Yvan Sutherland a largement contribué à l'intérêt croissant porté par les institutions académiques mais aussi industrielles à la conception de circuits asynchrones en publiant un article maintenant célèbre intitulé « Micropipeline » [67]. Depuis, les travaux sur la conception de circuits asynchrones ne cessent de s'intensifier [34].

Aujourd'hui, malgré la prépondérance des circuits synchrones, des outils de conception associés à ce type de circuits qui ne cessent de progresser, ainsi qu'une formation d'ingénieurs uniquement consa-

crée à ce style de conception, de plus en plus d'acteurs du domaine du semiconducteur s'intéressent à la conception asynchrone. En effet, le rôle de plus en plus important des variations de process, de température et de tension dans la conception de circuits en technologie submicronique, rend extrêmement complexe la conception de circuits synchrones [16]. Les circuits asynchrones sont donc l'objet d'un regain d'intérêt et d'activité de recherche du fait de leurs bénéfices potentiels en faible consommation, faible bruit, robustesse aux variations (technologiques ou d'utilisation) et modularité [8].

Ce chapitre est largement inspiré du manuscrit de thèse de Bertrand Folco [30] et d'un rapport écrit par Marc Renaudin sur l'état de l'art de la conception de circuits asynchrones [62].

2.2 Avantages des circuits asynchrones

2.2.1 Absence d'horloge

L'avantage évident des circuits asynchrones, en raison de l'absence de signal d'horloge global, est que tous les problèmes liés à la manipulation de l'horloge sont supprimés. Ces problèmes sont de plus en plus présents dans les technologies actuelles car ces dernières sont de plus en plus rapides et autorisent des circuits de plus en plus complexes, tant sur le plan fonctionnel qu'au niveau de leur fabrication. Aujourd'hui, la conception des circuits d'horloge est devenue une question de toute première importance puisqu'ils peuvent directement limiter les performances du système synchrone. Par exemple pour les « systèmes sur puce » (SoC), la multiplicité des horloges entre les différents blocs fonctionnels souvent issus de concepteurs différents et fonctionnant à des fréquences différentes, la propagation d'un signal d'horloge sur une puce de plusieurs millimètres de côté, mais également les problèmes de synchronisation dus aux dispersions technologiques augmentent le coût et la complexité de conception de ces circuits. D'une manière générale, l'approche synchrone a des conséquences à tous les niveaux de la conception : estimation des capacités d'interconnexion avant la phase de synthèse logique, caractérisation des timings des éléments de bibliothèque, en particulier les temps de setup et de hold, estimation de timing après extraction sur layout, confiance dans la caractérisation de la technologie, conception électrique et placement physique des arbres d'horloge pour réduire les problèmes de gigues, génération de vecteurs de test pour valider / tester les chemins critiques et trier les circuits après fabrication en fonction de leurs performances. Les techniques et outils de conception de circuits synchrones évoluent avec les technologies pour estimer de plus en plus précisément les temps d'arrivée de l'horloge sur les bascules d'un circuit.

Au contraire les circuits asynchrones n'utilisent pas d'horloge globale. Les éléments de synchronisation sont distribués dans l'ensemble du circuit, leur conception est plus facile à maîtriser. Comme pour certains circuits asynchrones, le fonctionnement est indépendant des retards qui peuvent être introduits, le problème de temps d'arrivée d'un signal d'un bout à l'autre du circuit n'influence pas la correction fonctionnelle du système. Le principe de synchronisation locale des circuits asynchrones constitue alors directement un outil d'aide à la conception de systèmes complexes.

2.2.2 Faible consommation

Par rapport aux circuits synchrones, plusieurs facteurs de réduction de la consommation sont à prendre en compte. La première conséquence de la suppression de l'horloge est la suppression de la consommation associée à celle-ci. Dans les circuits rapides, la consommation de l'horloge et des éléments de mémorisation peut représenter jusqu'à plus de 50% de la consommation du circuit. Cette consommation est due au chargement des circuits d'horloge et aux transitions dans les bascules. De plus, dans les circuits numériques, une part non négligeable de la consommation provient de transitions inutiles en raison de la présence d'aléas dans les blocs de logiques combinatoires. En synchrone, ces aléas ne sont pas gênants fonctionnellement car ils doivent avoir disparus à l'arrivée du prochain front d'horloge, cependant ils représentent une consommation relativement importante. La conception des circuits asynchrones suppose qu'il n'y ait aucun aléa afin d'obtenir des circuits corrects fonctionnellement (cf. paragraphe 2.3.3). La part de consommation due à ces aléas est donc supprimée.

Un autre aspect important de la réduction de la consommation concerne la mise en veille de la logique asynchrone à tout niveau de granularité. Dans un circuit synchrone, tous les blocs de logique se trouvent alimentés en données et commutent à chaque front d'horloge, que ces transitions soient utiles dans le bloc ou non. Par exemple dans un microprocesseur, la plupart des blocs n'a pas besoin de travailler pour toutes les instructions : le multiplieur est utile lors d'une multiplication, un additionneur lors d'une addition et non le contraire. Comme le mécanisme de synchronisation à horloge est simplifié, tous les blocs fonctionnent alors que fonctionnellement ils n'apportent rien. Au contraire, grâce au mécanisme de synchronisation locale des circuits asynchrones et à leur fonctionnement flot de données, seuls les blocs utiles reçoivent des données et donc consomment. Cet argument de réduction de la consommation des circuits asynchrones est donc particulièrement intéressant dans le cas d'architectures irrégulières. Des optimisations sont cependant possibles en synchrone pour contrôler l'horloge (« gated-clock », voir paragraphe 5.1.1), mais leur conception reste difficile à mettre en œuvre, alors que la mise en veille de la logique à tous les niveaux de granularité est gratuite dans le cas asynchrone. Un autre avantage de cette activité conditionnelle de la logique est que le redémarrage de la logique est instantané et géré au niveau matériel à tous les niveaux de l'architecture : il n'est pas nécessaire de concevoir un logiciel, parfois complexe, pour contrôler l'activation et la désactivation de tout ou partie du système.

Une autre propriété intéressante des circuits asynchrones pour la faible consommation est leur robustesse et leur adaptation aux conditions de fonctionnement. Comme la puissance varie avec le carré de la tension, il est aisé de réduire la tension d'alimentation pour limiter la puissance consommée. Cette réduction de la tension d'alimentation des circuits asynchrones peut se faire avec un matériel et un temps de conception minimum. Contrairement aux circuits synchrones, il n'y a pas besoin d'adapter et de caractériser la fréquence d'horloge aux différentes conditions de fonctionnement car la correction fonctionnelle est garantie quels que soient les délais dans les cellules élémentaires. De manière statique, il est ainsi aisément possible de choisir entre performance et faible consommation dans un circuit asynchrone : le circuit sera d'autant moins consommant à tension réduite que performant à tension nominale. De plus, il est également possible de faire varier dynamiquement la tension d'alimentation d'un circuit en fonction

de son activité pour réduire la consommation. Ce dernier aspect sera développé plus en détail dans la section 5.1.3.

Enfin, dans le cadre de systèmes embarqués, et particulièrement pour les réseaux de capteurs sans fil, la durée de vie de la batterie est un problème clé influant directement sur la durée de vie globale du réseau. Du fait que le contrôle entre les blocs soit géré localement, les circuits asynchrones ont une signature en courant relativement lisse dans le temps, c'est à dire sans pic de courant. Au contraire, dans les circuits synchrones, tous les blocs sont synchronisés par un signal d'horloge unique. Ainsi, un pic de courant important apparait à chaque front montant d'horologe.

Ces pics de courant ont un effet très important sur la durée de vie de la batterie [42]. En effet, si le courant débité est faible, les réactions de réduction au niveau de la cathode de la batterie se font de manière homogène dans le volume de cette cathode poreuse. Au contraire, les pics importants de courant provoquent une réaction de réduction sur la surface externe de la cathode, empèchant l'accès à de nombreux sites internes de la cathode participant à la réaction. En conséquence, à puissance moyenne consommée identique, l'utilisation de matériel asynchrone permet d'augmenter la durée de vie d'un système alimenté sur batterie.

2.2.3 Faible bruit, circuits intrinsèquement résistants

Comme nous venons de le voir, une autre conséquence de la suppression de l'horloge et de la distribution du contrôle dans la structure du circuit est que les problèmes de pics de consommation sont inexistants. En effet, l'activité électrique d'un circuit asynchrone est bien mieux répartie dans le temps que pour un circuit synchrone. Il n'y a pas d'instants prédéfinis pour activer un opérateur comme c'est le cas avec les fronts d'horloge. Ainsi, la consommation dans les lignes du circuit est distribuée dans le temps, que ce soit au niveau de la logique ou des éléments de mémorisation. De plus, comme nous l'avons vu, l'absence de l'horloge supprime sa consommation, ainsi le bruit généré dans les lignes d'alimentation et la puissance des ondes magnétiques émises par le circuit sont limités. Les circuits asynchrones présentent alors une alternative sérieuse pour concevoir des systèmes numériques sous contraintes de limitation de bruit.

Cette propriété est également très intéressante pour l'utilisation des circuits asynchrones dans des produits sécurisés, afin de limiter les attaques par canaux cachés. Ce type d'attaque est basé sur la recherche et l'exploitation de toute corrélation entre les données manipulées par un circuit et ses signaux externes. Ces signaux sont soit les signaux d'alimentation, soit les signaux d'horloge, soit les signaux électromagnétiques des circuits. Ils sont appelés signaux compromettants ou canaux cachés du fait qu'ils peuvent faire fuir des informations indésirables. Les circuits asynchrones sont donc, par construction, plus résistants à ce type d'attaque [11].

2.2.4 Modulaires

La modularité des circuits asynchrones est quasi-parfaite. Elle est due à la localité du contrôle et à l'utilisation par tous les opérateurs d'un protocole de communication bien spécifié. Il est en effet très facile de construire une fonction, et même un système complexe en associant des blocs préexistants [15]. Cette modularité aide aussi à repartir les taches de conception de blocs distincts sur différentes équipes de concepteurs. La spécification explicite du protocole aux interfaces des blocs asynchrones permet de les interconnecter par simple assemblage. Il est ainsi possible de concevoir un système flot de données, tout comme il est possible d'assembler un système composé de blocs synchrones autour d'un bus de communication. Dans le cas asynchrone, la modularité entre blocs est facilitée par deux aspects. Tout d'abord, il n'y a pas besoin de spécifier un bloc en fonction du nombre de cycles d'horloge pour obtenir et échanger des données avec un bloc distant, la synchronisation est effectuée par les données (il s'agit bien d'un modèle flot de données). Ensuite, au niveau implantation d'un circuit asynchrone, lorsqu'aucune hypothèse temporelle ne garantit le fonctionnement, la conception au niveau physique est rendue plus facile. Les phases de placement routage sont moins contraintes, sauf pour des questions de performances ou de sécurité. La correction du système est garantie quels que soient les délais dans les interconnexions aux interfaces.

A l'heure actuelle, les systèmes tout intégrés (System-On-Chip) sont composés de blocs fonctionnels utilisant des horloges définies localement. Il est donc nécessaire d'avoir recours à des systèmes de synchronisation capables de garantir des communications fiables entre blocs contrôlés par des horloges distinctes. Ce type de conception est typiquement du ressort de la conception asynchrone. Il est plus aisé de concevoir des systèmes de communication implantés dans le style asynchrone (notamment par rapport aux problèmes de traitements indéterministes), que de concevoir des systèmes de communications synchrones multi horloges [6, 59].

Pour conclure, les propriétés de robustesse et modularité des circuits asynchrones, en raison de leur synchronisation locale sont particulièrement intéressantes lorsque l'on souhaite promouvoir la réutilisation de blocs dans une entreprise, ou d'un point de vue plus général, l'échange de propriétés intellectuelles (IPs) dans le cadre d'implantation de systèmes complexes.

2.3 Concepts de base des circuits asynchrones

La conception de la plupart des circuits intégrés logiques est facilitée par deux hypothèses fondamentales : les signaux manipulés sont binaires et le temps est discrétisé. La binarisation des signaux permet une implantation électrique simple et offre un cadre de conception maîtrisé grâce à l'algèbre de Boole. La discrétisation du temps permet, quant à elle, de s'affranchir des problèmes de rétroactions et/ou boucles combinatoires, ainsi que des fluctuations électriques transitoires. Cependant, un système fonctionnant sans ces hypothèses peut obtenir de meilleurs résultats. Les circuits asynchrones conservent un codage discret des signaux mais ne font pas l'hypothèse que le temps est discrétisé. Ils définissent ainsi

une classe de circuits beaucoup plus large car leur contrôle peut être assuré par tout moyen autre que l'horloge unique des circuits synchrones.

2.3.1 Mode de fonctionnement asynchrone

Ce paragraphe est destiné à clarifier l'utilisation du terme « asynchrone » dans le contexte de la conception de circuits numériques. Cela nous permet d'introduire le mode de fonctionnement asynchrone et ses différences avec le mode de fonctionnement synchrone.

« Asynchrone » signifie qu'il n'existe pas de relation temporelle a priori entre des évènements. Dans un système intégré, ces évènements sont des évènements au sens large (contrôle ou données) implantés par des signaux électriques. Il faut donc définir ce qu'est un signal « asynchrone ».

Si on prend l'exemple d'un signal d'interruption appliqué à un microprocesseur (que ce dernier soit synchrone ou asynchrone), on le qualifie de signal d'interruption asynchrone par rapport au fonctionnement du microprocesseur. Cette situation est délicate à résoudre puisqu'il faut échantillonner un signal pour mesurer son niveau sous contrôle d'un évènement (par exemple l'horloge du processeur) qui n'a aucune relation temporelle avec ce signal extérieur. Dans cet exemple, le terme asynchrone qualifie une indétermination sur la relation d'ordre entre le signal d'horloge et le signal d'interruption, et donc sur le niveau de ce dernier.

Quand on parle de circuits asynchrones, on qualifie des circuits qui gèrent des signaux asynchrones entre eux mais dont le comportement est parfaitement déterminé. Par exemple, supposons deux signaux qui transportent de l'information sous forme de changement de niveau. La spécification est telle qu'il n'existe pas a priori de relation de causalité entre ces deux signaux, ces derniers sont donc asynchrones. Cependant, il est garanti qu'un évènement doit se produire sur ces deux signaux. Dans ce système, il y a indétermination sur les instants d'occurrence des évènements de chaque signal, mais le fait que les évènements aient lieu est absolument certain. Un élément de base, très utilisé dans les circuits asynchrones, permet naturellement un rendez-vous entre deux signaux asynchrones : la porte de Muller présentée dans le paragraphe 2.3.2.2. De manière générale, on parle de synchronisation d'évènements : un évènement est généré en sortie si et seulement si il y a évènement sur les deux entrées de la porte, quels que soit les instants d'occurrence.

Les circuits asynchrones fonctionnent donc avec la seule connaissance de l'occurrence des évènements, sans connaissance de l'ordre. Le fonctionnement est identique à celui des systèmes flots de données. On peut ainsi spécifier l'enchaînement des évènements sous forme d'un graphe de dépendances (réseau de Petri par exemple). L'évolution du système est garantie par l'évolution conjointe (voire concurrente) des éléments qui le composent. Chaque élément évolue avec les seules informations des éléments auxquels il est connecté. L'analogie avec le modèle des processus séquentiels communicants [37] est très forte. Dans ce modèle, les processus se synchronisent par passage de messages via des canaux de communication. Pour échanger des informations, deux éléments doivent se synchroniser, ils échangent leurs

informations à travers les canaux de communication puis poursuivent leur flot d'exécution de manière indépendante. On peut donc qualifier ce type de communication de synchrone, en opposition avec les communications asynchrones qui s'effectuent généralement par passage de messages via une mémoire intermédiaire, donc sans synchronisation entre les éléments communicants. Le langage CHP, que nous utilisons comme langage de haut niveau pour décrire nos circuits asynchrones, est directement issu de ce modèle [50].

Ainsi dans les circuits asynchrones, la seule connaissance de l'occurrence des évènements est suffisante pour implanter la synchronisation ; il n'est pas nécessaire d'introduire de mécanisme global d'activation du système. C'est effectivement la différence avec les circuits synchrones. Dans un système synchrone, comme nous l'avons vu, tous les éléments évoluent ensemble sur un évènement du signal d'horloge : l'exécution de tous les éléments se trouve synchronisée. Ce mécanisme de synchronisation introduit une contrainte temporelle globale : afin d'obtenir un fonctionnement correct, tous les éléments doivent respecter un temps d'exécution maximum imposé par la fréquence du mécanisme d'activation. A l'opposé, les systèmes asynchrones évoluent de manière localement synchronisée, le déclenchement des actions dépend uniquement de la présence des données à traiter. Ainsi, la correction fonctionnelle est indépendante de la durée de traitement des éléments du système.

2.3.2 Un contrôle local

Comme nous venons de le présenter, le point fondamental du mode de fonctionnement asynchrone est que la synchronisation et le transfert d'informations sont réalisés localement. Ceci est effectué par une signalisation adéquate. Le contrôle local doit remplir les fonctions suivantes : être à l'écoute des communications entrantes, déclencher le traitement local si toutes les informations sont disponibles en entrée et produire des valeurs sur les sorties. De plus, pour que l'opérateur sache qu'il est autorisé à émettre de nouvelles valeurs sur ces sorties, il doit être informé que les valeurs qu'il émet sont bien consommées (reçues) par l'opérateur en aval. Ainsi, pour permettre un fonctionnement correct indépendamment du temps, le contrôle local doit implanter une signalisation bidirectionnelle. Les communications sont dites de type "poignées de mains" ou "requête-acquittement" (figure 2.1).

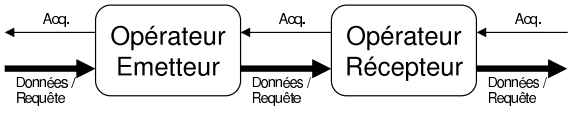


FIG. 2.1 – Communication de type requête-acquittement entre opérateurs asynchrones.

Toute action doit être acquittée par le récepteur afin que l'émetteur puisse émettre de nouveau. Cette signalisation bidirectionnelle offre un mécanisme qui permet de garantir la synchronisation et la causalité des évènements au niveau local et donc la correction fonctionnelle du système dans son ensemble.

2.3.2.1 Protocoles de communication

Pour implanter une signalisation bidirectionnelle, deux protocoles de communication sont couramment utilisés : le protocole deux phases, encore appelé NRZ (Non Retour à Zéro) ou « half-handshake » et le protocole quatre phases, encore appelé RZ (Retour à Zéro) ou « full-handshake ». Ces deux protocoles sont représentés respectivement figure 2.2 et figure 2.3. Dans les deux cas, il faut noter que tout évènement sur un signal de l'émetteur est acquitté par un évènement sur un signal du récepteur, et vice-versa. Ce mécanisme permet de garantir l'insensibilité au temps de traitement de l'opérateur. Dans ce protocole, seul importe l'occurrence des évènements, localement, entre l'émetteur et le récepteur, et non leurs temps relatifs, ni leurs ordres respectifs par rapport à l'entrée de l'émetteur ou à la sortie du récepteur. Le choix du protocole de communication affecte les caractéristiques de l'implantation du circuit (surface, vitesse, consommation, robustesse, etc.).

Le protocole deux phases constitue la séquence d'échange d'information minimale nécessaire à une communication : les données sont représentées par des fronts (montants et descendants).

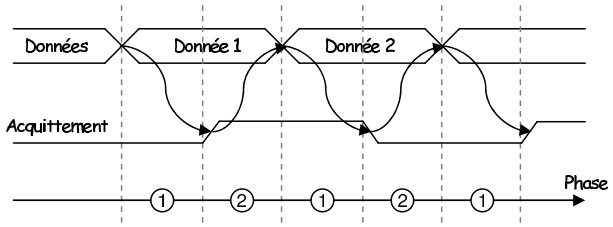


FIG. 2.2 – Protocole deux phases.

Les deux phases sont les suivantes :

– Phase 1 : c'est la phase active du récepteur qui détecte la présence d'une donnée, effectue le traitement et génère le signal d'acquittement.
– Phase 2 : c'est la phase active de l'émetteur qui détecte le signal d'acquittement et émet une nouvelle donnée si elle est disponible.

Ce protocole, malgré son apparente efficacité, n'est que peu utilisé en raison des difficultés rencontrées dans la logique nécessaire à son implantation. L'asymétrie entre les deux phases successives (montée ou descente de l'acquittement) se révèle un obstacle important à la réalisation de ce protocole.

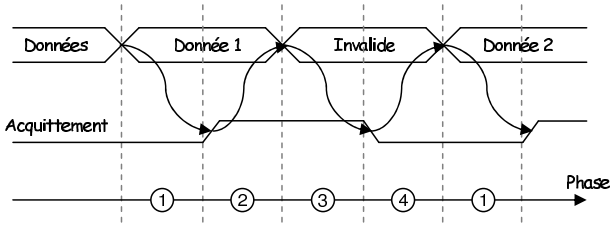


FIG. 2.3 – Protocole quatre phases.

Les différentes phases du protocole quatre phases sont les suivantes :

– Phase 1 : le récepteur détecte une nouvelle donnée, effectue le traitement et génère le signal d'ac-

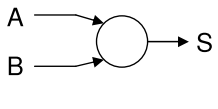
quittement.

– Phase 2 : l'émetteur détecte le signal d'acquittement et invalide les données.
– Phase 3 : le récepteur détecte l'état invalide et désactive le signal d'acquittement.
– Phase 4 : l'émetteur détecte l'état invalide du signal d'acquittement et émet une nouvelle donnée si elle est disponible.

A l'opposé du protocole deux phases, le protocole quatre phases est le plus utilisé en raison de la symétrie de ses phases. En doublant leur nombre, l'état du système est invariablement le même au début et à la fin d'une communication.

2.3.2.2 Implantation du protocole : la porte de Muller

Pour implanter de tels protocoles, les portes logiques élémentaires ne suffisent pas. Ce paragraphe présente rapidement le fonctionnement des portes de Muller (autrement appelées « C element »), proposées par Muller dans [54]. Elles sont essentielles pour l'implantation de circuits asynchrones : elles réalisent le rendez-vous entre plusieurs signaux. La sortie copie la valeur des entrées lorsque celles-ci sont identiques, sinon elle mémorise la dernière valeur calculée. La figure 2.4 représente le symbole d'une porte de Muller à deux entrées et sa spécification sous forme de table de vérité.



B \ A	0	1
0	0	S^{-1}
1	S^{-1}	1

FIG. 2.4 – Symbole et spécification de la porte de Muller.

A partir de cette spécification, la figure 2.5 décrit plusieurs implantations possibles : sous forme de portes logiques élémentaires et sous forme de transistors.

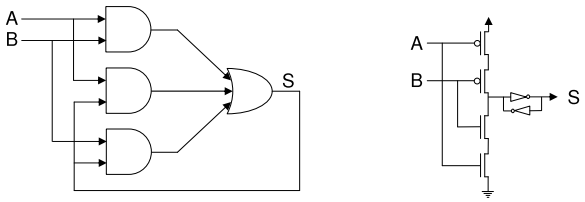


FIG. 2.5 – Implantations de la porte de Muller.

Il existe des variantes de la porte de Muller [63]. La porte de Muller généralisée (traduit de l'anglais « generalized C element ») [48] est une porte de Muller dans laquelle les signaux qui font monter la sortie à 1 ne sont pas toujours les mêmes que ceux qui la font descendre à 0. Dans ce cas, la porte de Muller est également dite dissymétrique. A titre d'exemple, la figure 2.6 illustre une porte de Muller généralisée (son symbole, sa spécification et sa réalisation au niveau transistors). Dans cet exemple, il suffit que les entrées B et C aient la valeur 1 pour que la sortie passe à 1 et que les entrées A et B aient la valeur 0 pour que la sortie passe à 0. Sinon la sortie est mémorisée.

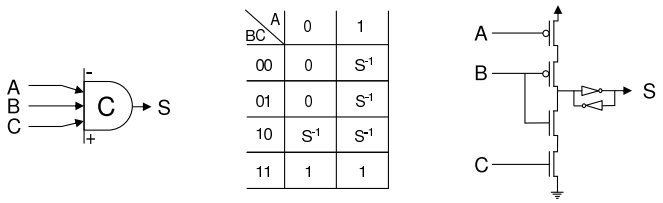


FIG. 2.6 – Porte de Muller dissymétrique.

2.3.2.3 Codage des données

Comme nous venons de le voir, les protocoles de communication doivent détecter la présence d'une donnée sur leur entrée et leur disponibilité. Pour résoudre ce problème, il est nécessaire d'adopter un codage particulier pour les données. Il est en effet impossible d'utiliser un fil unique par bit de donnée : cela ne permet pas de détecter que la nouvelle donnée prend un état identique que la valeur précédente. Deux types de solutions sont possibles, soit la création d'un signal de requête associé aux données, soit l'utilisation d'un codage insensible aux délais bifilaire ou double-rail par bit de donnée [61].

2.3.2.4 Codage « données groupées »

La façon la plus évidente de caractériser la validité des données consiste simplement à ajouter un fil au bus de données afin de spécifier si les données y sont valides (figure 2.7(a)). Les bus de données utilisent le traditionnel schéma de la logique synchrone : un fil par bit de données, ce qui s'appelle parfois mono-rail (« single rail ») dans la littérature. Le fil spécifiant la validité des données, dit signal de requête, est typiquement implanté avec le retard adéquat. Ce retard est conçu égal ou supérieur au temps de calcul dans le pire cas.

Efficace en termes de surface (en termes de nombre de fils et donc en nombre de portes pilotant ces fils), ce type de codage permet une bonne réalisation des circuits asynchrones. Cependant, les inconvénients de ce codage dans certains circuits sont qu'en utilisant le retard assorti, le fonctionnement est fixé au pire cas : le fonctionnement ne dépend donc pas de la propagation réelle des données à l'intérieur d'un étage ; et les circuits obtenus ne sont plus insensibles aux délais.

2.3.2.5 Codage insensible aux délais

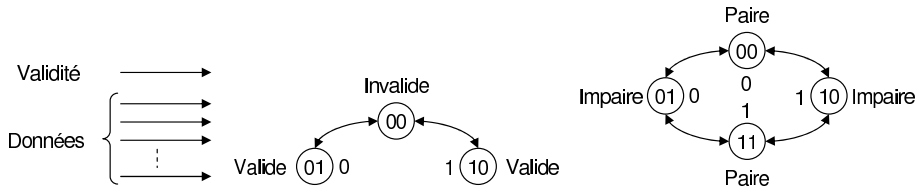
Contrairement au codage « données groupées » où les données et leur validité sont totalement séparées, une autre approche plus complexe consiste à intégrer l'information de la validité dans les données. Cette approche possède la propriété suivante : les données sont détectées à l'arrivée sans que cela ne repose sur aucune hypothèse temporelle. Ceci implique donc robustesse, portabilité et facilité lors de la conception.

- Codage 4 états : dans ce codage, chaque bit de donnée est représenté par deux fils. Parmi les quatre

états possibles pour ces 2 fils, la moitié est réservée à la valeur 0, l'autre à la valeur 1 (figure 2.7(c)). L'émission d'une nouvelle donnée se traduit par le changement d'un seul fil : celui de droite pour exprimer la même valeur que précédemment, celui de gauche pour exprimer la valeur opposée. La validité des données est donc assurée par le changement du couple de fils.

- Codage 3 états : dans ce codage également, chaque bit est représenté par 2 fils. En revanche, les valeurs représentées ne sont pas dupliquées : une seule combinaison représente chaque valeur, tandis que la troisième indique l'état invalide et la quatrième reste inutilisée (figure 2.7(b)). Ainsi, comme on le voit sur le schéma, le passage d'une valeur valide à l'autre se traduit nécessairement par le passage par l'état invalide.

Parmi tous ces codages, le codage 3 états, qui peut par ailleurs sembler le moins naturel, est aujourd'hui de loin le plus utilisé pour des raisons d'implantation et de sécurité. En effet les deux autres posent des problèmes en termes de logique additionnelle : dans le codage « données groupées », il faut recombiner les données avec le signal de validité et dans le codage 4 états, un étage de logique supplémentaire est nécessaire pour tester la parité du couple de fils.



(a) donnée groupée. (b) codage 3 états. (c) codage 4 états.

FIG. 2.7 – Codages les plus utilisés.

Enfin, le codage 3 états, souvent appelé codage « double-rail » – en raison des deux fils par bit de données – est un cas particulier du codage « one hot ». En effet, il est possible d'étendre la représentation, pour un digit en base N, N fils seront utilisés : chaque fil représente une valeur et l'état invalide est déterminé par la remise à zéro de tous les fils. Les combinaisons où au moins deux fils sont à 1 sont interdites. Ce codage est également appelé « 1 parmi N » ou encore « multi-rail » dans ce manuscrit. Une dernière extension possible est le codage « M parmi N ».

2.3.3 Conception de circuits asynchrones : le problème des aléas

Dans le sens le plus général, un aléa est une activité non désirée (« glitch ») en réponse à un changement sur certaines entrées. Un aléa peut se produire en raison de la différence de délais entre les portes. La présence d'aléas dans un circuit — notamment pour un circuit asynchrone, très sensible à ces transitions — entraîne des fautes de fonctionnement. Il est possible de détecter la présence d'aléas à différentes étapes de la conception. Un séquenceur peut, par exemple, contenir des aléas fonctionnels qui trouvent leur origine dans la spécification de la fonction elle-même. Si ce problème est réglé au niveau de la spécification, il peut encore apparaître des aléas au niveau de l'implantation. Cela signifie que la conception d'un circuit ne s'arrête pas après la spécification. Il faut encore s'assurer que les techniques de réalisation

choisies sont exemptes d'aléas.

Dans le cadre de la conception de circuits asynchrones, il est important de remarquer que ces circuits requièrent une conception attentive et fine de toutes les parties (combinatoires et séquentielles) du circuit, de la spécification jusqu'à l'implantation matérielle [13, 30].

2.3.4 Classification des circuits asynchrones

Les circuits asynchrones sont communément classifiés suivant le modèle de circuit et d'environnement adopté. La figure 2.8 présente la terminologie habituellement utilisée pour qualifier les circuits asynchrones. Plus le fonctionnement du circuit respecte fondamentalement la notion d'insensibilité aux délais, plus le circuit est robuste et plus il est complexe. Dans la suite, nous présentons brièvement ces catégories de circuits en commençant par les circuits qui respectent fondamentalement la notion d'asynchronisme, puis des circuits dans lesquels sont introduites des hypothèses temporelles de plus en plus fortes. Cette approche va donc des circuits insensibles aux délais vers des circuits s'approchant du synchrone.

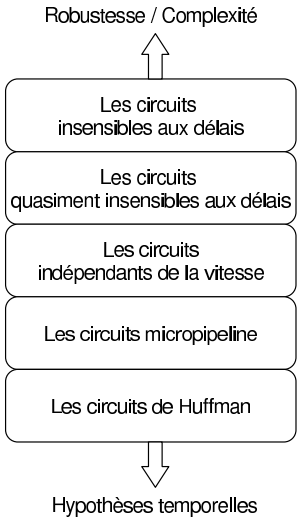


FIG. 2.8 – Classification des circuits asynchrones.

2.3.4.1 Les circuits insensibles aux délais

Les circuits de ce type utilisent un modèle de délai non borné dans les fils et les portes. Aucune hypothèse temporelle n'est introduite lors de la conception de ce type de circuit, cela signifie qu'ils sont fonctionnellement corrects quels que soient les délais introduits dans les éléments logiques et les portes. Basés sur des travaux de Clark Wasley, et re-formalisés dans [68], les circuits DI sont supposés répondre toujours correctement à une sollicitation externe pourvu qu'ils aient assez de temps de calcul. Ceci impose donc que chaque récepteur d'un signal informe toujours l'expéditeur que l'information a été reçue. Les contraintes de réalisation pratique imposées par ce modèle sont très fortes. De plus la plupart des circuits conçus aujourd'hui utilisent des portes logiques qui ne possèdent qu'une seule sortie.

L'adoption du modèle de délai « non borné » ne permet pas leur utilisation. En effet, les portes logiques standard ont leurs sorties qui peuvent changer si une seule de leur entrée change. Comme nous l'avons souligné, toutes les composantes de ce type de circuits doivent s'assurer du changement des entrées pour produire une sortie. Si la sortie change alors qu'une seule des entrées change, seul le changement de l'entrée active est acquitté, sans pouvoir tester l'activité des autres entrées. Les seules portes qui respectent cette règle sont la porte de Muller et l'inverseur. Malheureusement, les fonctions réalisables avec seulement des portes de Muller et des inverseurs sont très limitées [50]. La seule solution est d'avoir recours à un modèle de circuit de type « portes complexes » pour les composants élémentaires. Dans ce cas, la construction de ces circuits se fait à partir de composants standard plus complexes que de simples portes logiques, qui peuvent posséder plusieurs entrées et plusieurs sorties.

2.3.4.2 Circuits quasi-insensibles aux délais (QDI)

Comme son nom l'indique, la classe des circuits QDI est un sur-ensemble de la classe DI. Cette classe adopte le même modèle de délai « non borné » pour les connexions et les portes mais ajoute la notion de fourche isochrone (« isochronic fork ») [7, 50].

Une fourche est un fil qui connecte un émetteur unique à au moins deux récepteurs. Elle est qualifiée d'isochrone lorsque les délais entre l'émetteur et chaque récepteur sont tous identiques. Cette hypothèse a des conséquences importantes sur le modèle et les réalisations possibles. Elle résout notamment le problème de l'utilisation de portes logiques à une seule sortie, causé par la classe DI. Car si les fourches sont isochrones, il est possible de ne tester qu'une seule branche d'une fourche et de supposer que le signal s'est propagé de la même façon dans les autres branches. En conséquence, on peut autoriser l'acquittement d'une seule des branches de la fourche isochrone. Alain Martin a montré dans [49] que l'hypothèse temporelle de fourche isochrone est la plus faible à ajouter aux circuits DI pour les rendre réalisables à partir de portes à plusieurs entrées et une seule sortie. Les circuits QDI sont donc réalisables avec les mêmes portes que celles utilisées pour la conception de circuits synchrones.

Finalement, dans les circuits QDI, les signaux peuvent mettre un temps arbitraire à se propager dans les portes ainsi que dans les connexions (avec l'hypothèse de fourche isochrone) sans que cela ne perturbe la correction fonctionnelle du circuit. De telles réalisations sont, bien entendu, très délicates à mettre en œuvre étant donné les contraintes imposées à la conception. En contrepartie, ces circuits présentent un degré de robustesse quasi parfait. En théorie, aucune erreur inhérente au circuit ne peut se produire. En pratique, la contrainte de fourche isochrone est assez faible et est facilement remplie par une conception soignée, en particulier au niveau du routage et des seuils de commutation. Il suffit, finalement, que la dispersion des temps de propagation jusqu'aux extrémités de la fourche soit inferieure aux délais des opérateurs qui lui sont connectés (au minimum une porte et un fil dont une sortie interagit avec la fourche [7, 50]).

2.3.4.3 Circuits indépendants de la vitesse (SI)

Les circuits SI font l'hypothèse que les délais dans les fils sont négligeables, tout en conservant un modèle « non borné » pour les délais dans les portes. Ce modèle, qui n'est pas réaliste dans les technologies actuelles, est en pratique équivalent au modèle QDI à l'exception des fourches qui sont toutes considérées comme isochrones. Même si pendant longtemps, la communauté a tenté de cerner les différences entre ces deux modèles, il y aujourd'hui un consensus pour considérer les modèles QDI et SI comme équivalents. Dans [34], on trouve un schéma montrant comment une fourche isochrone peut être représentée par un circuit SI (figure 2.9).

Cependant, il semble plus pertinent d'utiliser le modèle QDI car celui-ci différencie les fourches isochrones de celles qui ne le sont pas. Ceci offre le moyen de vérifier les connexions du circuit qui ne sont pas insensibles aux délais au cours des différentes étapes de conception, et seulement celles-ci.

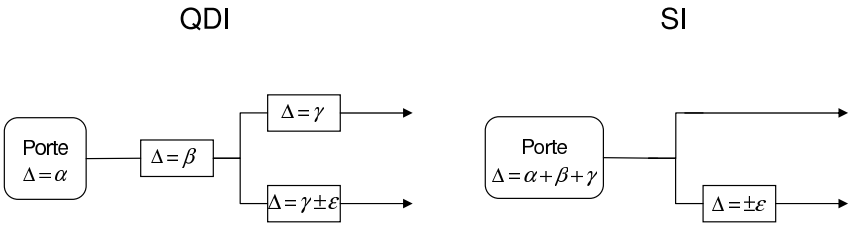


FIG. 2.9 – Équivalence entre les modèles SI et QDI.

2.3.4.4 Les circuits micropipeline

La technologie micropipeline a été initialement introduite par Ivan Sutherland [67]. Il faut en fait considérer que les circuits micropipelines correspondent plutôt à une classe d'architecture de circuits asynchrones, que directement à un modèle de délais de circuits asynchrones. Les circuits de cette classe sont composés de parties de contrôle insensibles aux délais qui commandent des chemins de données conçus en utilisant un modèle de délais bornés. La structure de base de cette classe de circuits est le contrôle d'une file (FIFO). Elle se compose de portes de Muller connectées tête bèche.

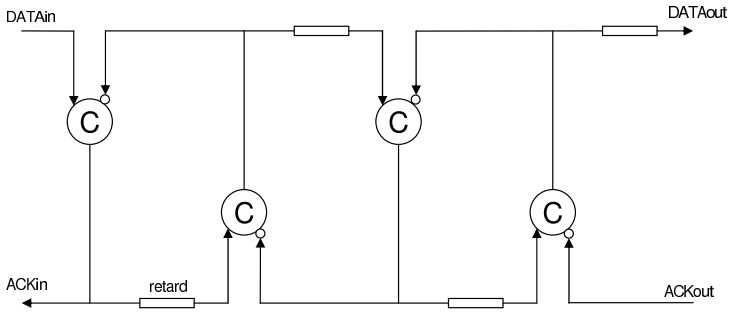


FIG. 2.10 – Structure de base des circuits micropipelines.

Cette structure implante un protocole deux phases. Le circuit réagit à des transitions de signaux et non pas d'états. On parle également d'une logique à évènements : chaque transition étant associée à un

évènement. Ainsi, tous les signaux sont supposés à zéro initialement. Une transition positive sur DATAin provoque une transition positive sur ACKin qui se propage également à l'étage suivant. Le deuxième étage produit une transition positive qui d'une part, se propage à l'étage suivant mais qui, d'autre part, revient au premier étage l'autorisant à traiter une transition négative cette fois. Les transitions de signaux se propagent donc dans la structure tant qu'elles ne rencontrent pas une cellule occupée. Ce fonctionnement de type FIFO est bien insensible aux délais.

Cette structure est une structure de contrôle simple. Elle peut être utilisée pour contrôler un chemin de données possédant des opérateurs de mémorisation et des opérateurs de traitements combinatoires. Dans ce cas, le codage des données et le protocole associé sont du type deux phases « données groupées ».

La motivation première pour le développement de cette classe de circuits était de permettre un pipeline élastique. En effet, le nombre de données présentes dans le circuit peut être variable, les données progressant dans le circuit aussi loin que possible en fonction du nombre d'étages disponibles ou vides. Cependant ce type de circuits révèle un certain nombre d'inconvénients. Il faut remarquer que les problèmes d'aléas ont été écartés en ajoutant des retards sur les signaux de contrôle. Cela permet en fait de se ramener à un fonctionnement en temps discret dans lequel il est autorisé la mémorisation des données seulement lorsqu'elles sont stables (à la sortie des portes combinatoires).

De nombreuses méthodes de conception s'attachent à concevoir des circuits micropipelines avec des modèles de délais plus ou moins différents. L'idée générale de ces approches est de séparer dans la spécification le contrôle des données et de les implanter séparément dans des styles différents. En particulier, les contrôleurs des registres de pipeline peuvent être obtenus avec un grand nombre de méthodes et avec des hypothèses de délais plus ou moins fortes.

2.3.4.5 Circuits de Huffman

Cette catégorie regroupe des circuits qui utilisent un modèle de délais identique aux circuits synchrones. Ils supposent que les délais dans tous les éléments du circuit et dans les connexions sont bornés ou même de valeurs connues. Les hypothèses temporelles sont donc du même ordre que pour la conception de circuits synchrones. Leur conception repose sur l'analyse des délais dans tous les chemins et les boucles de façon à dimensionner les temps d'occurrence des signaux de contrôles locaux. Ces circuits sont d'autant plus difficiles à concevoir et à caractériser qu'une faute de conception ou de délai les rend totalement non fonctionnels.

2.3.5 Le langage CHP

Dans ce paragraphe, nous allons exposer brièvement le langage CHP utilisé pour décrire des circuits asynchrones QDI afin de permettre la compréhension du code CHP qui est présent dans le reste du manuscrit.

2.3.5.1 Types de données

Les types existant en CHP peuvent être regroupés en types non signés (`MR`, `DR`, `NATURAL`, `BIT` et `BOOLEAN`) ou en types signés (`SMR`, `SDR`, et `INTEGER`). En fait, le type de base est `MR[B][L][D]` ou `SMR[B][L][D]`, les autres types étant des alias pour aider le développeur. Avec cette notation, `B` indique la base, `L` la longueur du type (ou nombre de digits) et `D` la dimension du vecteur. Si `D` n'est pas renseigné, on suppose qu'il vaut `1`, de même pour `L`. Ainsi `MR[B]` est équivalent à `MR[B][1][1]`.

`MR[B]` : codage multi rail de base B Ce type représente un nombre entre 0 et $B-1$. Chaque fil représente une des valeurs possibles entre 0 et $B-1$. Seulement un fil peut être à 1 à chaque instant.

`DR` : codage double rail Ce type est équivalent à un type $MR[2][1][1]$.

`BIT` : type binaire Ce type est équivalent à un type DR donc à un type $MR[2][1][1]$.

`SR` : codage simple rail Ce type est équivalent à un type $MR[1][1][1]$. Il permet de faire des synchronisation entre des éléments communicants. Il ne transmet aucune information mais permet d'acquitter un évènement.

`BOOLEAN` : type booléen Ce type est équivalent à un type $MR[2][1][1]$.

`NATURAL[MAX]` : codage d'un nombre naturel de 0 à MAX Ce type est transformé à la compilation en un type $MR[2][\lceil log_2(MAX+1)\rceil][1]$

`SMR[B]` : codage multi rail signé de base B Ce type représente un entier signé dont la valeur est comprise entre $\lfloor -B/2 \rfloor$ et $\lfloor B/2-1 \rfloor$. Seulement un fil peut être à `1` à chaque instant.

`SDR` : codage double rail Ce type est équivalent à un type $SMR[2][1][1]$.

`INTEGER[MAX]` : codage d'un nombre entier signé de -MAX-1 à MAX Ce type est transformé à la compilation en un type $SMR[2][\lceil log_2(MAX+1)+1\rceil][1]$

2.3.5.2 Littéraux

Les littéraux, ou constantes entières, sont notés en CHP avec la syntaxe suivante : `"<digit>.<digit>. ..."[B]`. Par exemple, 2008 pourra être noté “2.0.0.8”[10]

2.3.5.3 Opérateurs

Le CHP définit les opérateurs de comparaison traditionnels tels que `=`, `/=`, `<`, `>`, `<=`, `>=`, les opérateurs logiques `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`, `NOT`, l'affectation `:=`, les opérateurs arithmétiques `+`, `-`, `*`, `/`, `MOD`, `ABS`, les opérateurs de décalage `SRL`, `SRA`, `SLL`, `SLA`, `ROTL`, `ROTR`, et les opérateurs sur les canaux de communications `?`, `!`, `#` définis plus loin.

2.3.5.4 Instructions

On note " ;" l'opérateur de séquence et "," l'opérateur parallèle. Ainsi $I_1; I_2$ signifie que I_2 sera exécuté quand I_1 sera terminé et I_1, I_2 signifie que les deux instructions peuvent être exécutées en même temps.

Structure de contrôle Les structres de contrôle en CHP sont représentées par des commandes gardées dont la syntaxe est la suivante :

```
[ <garde> => <commandes> (LOOP|BREAK) ];
```

`<garde>` étant une expression booléenne et `<commandes>` une suite d'instructions. La structure `LOOP` permet de revenir au début la structure de contrôle lorsque la suite d'instructions est exécutée, permettant de faire une boucle. La structure `BREAK`, au contraire, sort de cette structure de contrôle pour exécuter l'instruction suivante.

Cette structure de contrôle peut être étendue aux instructions de choix dont la syntaxe est la suivante.

```
{@}@ [
      <commandes gardées> (BREAK | LOOP)
      <commandes gardées> (BREAK | LOOP)
    { <commandes gardées> (BREAK | LOOP) }
];
```

Le choix peut être déterministe (`@`) ou indéterministe (`@@`). Dans un choix déterministe, au plus une garde doit être vraie lors de l'évaluation. Dans le cas d'un choix indéterministe, si plusieurs gardes sont vraies en même temps, un tirage aléatoire est réalisé.

Opération sur les ports et les canaux Il y a trois opérateurs sur les canaux : envoie (`!`), réception (`?`) et la "probe" (`#`), dont les syntaxes sont les suivantes :

```
C ?;                    - - pour recevoir une donnée sans la conserver
```

`C?x;`	- - pour recevoir une donnée et la sauver dans la variable x
`C!;`	- - pour émettre un acquittement
`C!x;`	- - pour envoyer une donnée
`C#`	- - retourne `vrai` si une donnée est en attente sur le port
`C# <expression>;`	- - pour lire une donnée sans l'acquitter

2.3.5.5 Exemple

L'exemple montré figure 2.11 décrit un arbitre entre les ports d'entrée `in1` et `in2` donnant l'accès à une ressource partagée connectée sur `out1`.

```
COMPONENT arbitre_SR_2
    PORT ( in1 : IN  DI SR;
           in2 : IN  DI SR;
           out1: OUT DI SR )
BEGIN
    PROCESS arbitre_SR_2
        PORT ( in1 : IN  DI SR;
               in2 : IN  DI SR;
               out1: OUT DI SR)
    BEGIN
        @@ [ in1# => out1! ; in1? ; LOOP
             in2# => out1! ; in2? ; LOOP
        ];
    END;
END;
```

FIG. 2.11 – Exemple de code CHP d'un arbitre.

2.4 Conclusion

Les problèmes liés à l'horloge dans la conception de circuits synchrones de plus en plus complexes suggèrent de se tourner vers la nouvelle approche de la conception de circuits asynchrones. Contrairement au mécanisme d'horloge globale utilisé dans les circuits synchrones, la synchronisation dans les circuits asynchrones est effectuée localement grâce à une signalisation bidirectionnelle entre tous les éléments du circuit. Cette signalisation est implantée via des canaux de communications utilisant un protocole et un codage de données spécifiques. De nombreux types de circuits asynchrones existent : ils sont classés en fonction des hypothèses temporelles émises.

Comme nous l'avons vu, les circuits quasi-insensibles aux délais sont des circuits asynchrones très robustes qui peuvent être alimentés sur une large plage de tension. Ils sont donc la cible idéale pour des techniques de réduction de la consommation telles que l'adaptation dynamique en tension présentée au chapitre 5. De plus, la faible consommation pic de courant des circuits asynchrones permet aux systèmes

fonctionnant sur batterie d'augmenter leur durée de vie. Enfin, la signalisation locale entre les blocs permet aux circuits asynchrones d'avoir une très faible activité électrique, donc une faible consommation d'énergie, de se mettre en veille à tous les niveaux de granularité, et de se réveiller instantanément. Par conséquent, ce type de logique possède un grand intérêt pour les systèmes embarqués, et en particulier pour les systèmes ultra faible consommation que sont les nœuds composant les réseaux de capteurs sans fil.

Chapitre 3

Architecture générale et consommation typique

3.1 Architecture matérielle

Cette section a pour but de montrer les grandeurs typiques de la consommation des différentes parties qui peuvent être présentes dans les nœuds des réseaux de capteurs sans fil. Le schéma d'une architecture typique de réseaux de capteurs sans fil est présenté figure 3.1.

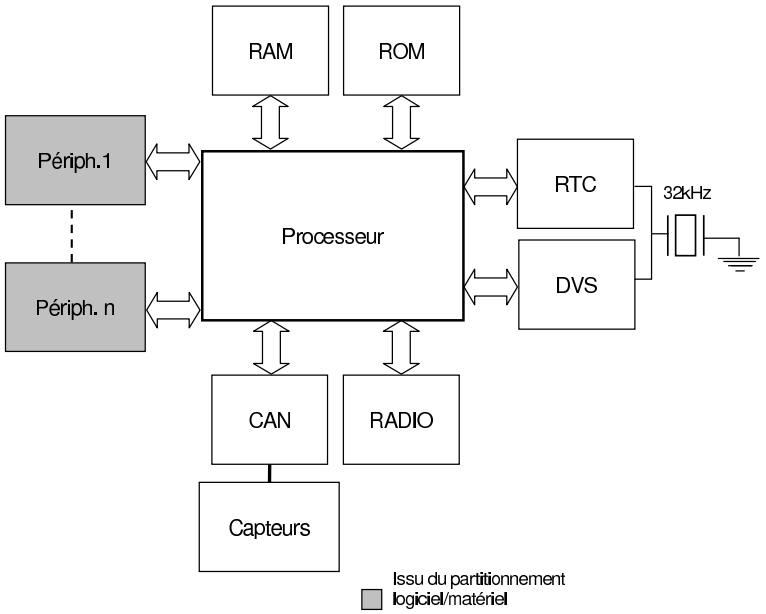


FIG. 3.1 – Architecture typique d'un nœud de réseau de capteurs sans fil.

3.1.1 Microcontrôleurs

3.1.1.1 Msp430

Le msp430 est un microcontrôleur commercial 16 bits fabriqué par Texas Instrument. Il est réputé pour sa très faible consommation énergétique et est intégré dans de nombreuses plateformes de réseaux de capteurs sans fil telles que Telos [58], BSN node [46], les plateformes Wavenis, ou encore eyesIFX. Les instructions peuvent prendre jusqu'à six cycles pour s'exécuter. Ce nombre de cycles dépend du format des instructions et du mode d'adressage utilisé.

Deux familles très faible consommation sont disponibles, la famille des msp430F1xx et la famille des msp430F2xx. La première famille à une puissance dynamique inférieure, mais fonctionne à une cadence plus faible (ne dépasse pas 8 MHz) et met plus de temps à se réveiller. La seconde famille peut fonctionner plus rapidement pour atteindre 16 MHz et possède quelques périphériques supplémentaires.

Les tableaux 3.1 et 3.2 montrent les caractéristiques typiques des ces deux familles de microcontrôleur, selon leurs fiches techniques respectives.

TAB. 3.1 – Caractéristiques du microcontrôleur msp430F1xx

Tension (V)	**Vitesse max. (MHz)**	**Consommation (mW)**	**Énergie par cycle (nJ)**	**Énergie par instruction (nJ)**
3.6	8	13,6	1,7	1,7 à 10,2
3,3	7,3	11,4	1,56	1,16 à 9,69
3	6,7	8,4	1,25	1,25 à 7,5
2,2	5	3	0,6	0,6 à 3,6
1,8	4,15	1,6	0,39	0,39 à 2,34

TAB. 3.2 – Caractéristiques du microcontrôleur msp430F2xx

Tension (V)	**Vitesse max. (MHz)**	**Consommation (mW)**	**Énergie par cycle (nJ)**	**Énergie par instruction (nJ)**
3,6	16*	27*	1,69*	1,69* à 10,14*
3,3	16	22,6	1,41	1,41 à 8,46
3	14	16,2	1,16	1,16 à 6,96
2,7	12	11,2	0,93	0,93 à 5,58
2,2	7,5	4,5	0,6	0,6 à 3,6
1,8	4,15	1,6	0,39	0,39 à 2,34

Dans la mesure où il n'est pas possible de faire fonctionner la famille des msp430F2xx à plus de 16 MHz, comme cette vitesse est déjà atteinte à 3,3 V, il n'y a pas d'intérêt à alimenter ce type de microcontrôleur à sa tension nominale de 3,6 V dans les conditions nominales utilisées pour mesurer ces valeurs.

D'après ces deux tableaux, on peut conclure qu'à tension égale, la famille du msp430F1xx a une consommation d'énergie par instruction toujours supérieure ou égale à celle de la famille msp430F2xx. Cependant, il est possible que les modèles des fiches techniques pour calculer les courants consommés et

les vitesses maximales de fonctionnement en fonction de la tension d'alimentation soient trop simplifiés. Les deux familles de microcontrôleurs sont tout de même proches en termes d'énergie consommée par cycle.

Selon le profil de l'application considérée, l'énergie consommée par instruction exécutée peut varier de façon importante. Comme seules les instructions à adressage direct s'exécutent en un cycle, on peut quand même considérer grossièrement qu'il faut en moyenne 2,5 cycles pour exécuter une instruction.

3.1.1.2 MICA

MICA, est un microcontrôleur universitaire asynchrone QDI 8bit CISC. Il a été conçu au laboratoire TIMA avec la collaboration de France Télécom R&D et ST Microelectronics en 2000 [1]. Il est fabriqué en technologie 0,25 μm de ST Microelectronics avec une bibliotèque de cellules standards. Les données sont d'une largeur de 8 bits et l'adressage sur 16 bits. Il contient 16 ko de mémoire RAM, 2 ko de ROM et un périphérique permettant de gérer jusqu'à 6 ports parallèle de 8 bits et deux ports série.

Ses caractéristiques sont reportées dans le tableau 3.3. La tension nominale du processeur est de 2,5 V. Il a cependant pu être "surcadencé" en augmentant la tension d'alimentation jusqu'à 3,5 V.

TAB. 3.3 – Caractéristiques du microcontrôleur MICA

Tension (V)	**Vitesse (Mips)**	**Consommation (mW)**	**Énergie par instruction (nJ)**
3.5	31.3	77	2,460
3.0	27.8	48,9	1,760
2.5	23.8	28	1,180
2.0	18.6	13,4	0,720
1.5	11.9	4,7	0,390
1.0	4.3	0,8	0,190

Comme on peut le voir, ce processeur consomme moins de 1,2 nJ par instruction à la tension nominale de 2,5 V, à la vitesse de 23,8 Mips. En comparant au msp430, on peut remarquer que pour une énergie par instruction d'environ 1,16 nJ, le msp430F2xx fonctionne à 14 MHz, soit environ 5,5 Mips alors que MICA fonctionne en moyenne à 23,8 Mips, soit plus d'un facteur d'environ 4 en performance. Si on fixe la vitesse, par exemple aux alentours des 4 Mips, soit environ 10 MHz, le msp430 consomme environ 0,8 nJ par instruction. MICA consomme à 0,19 nJ par instruction, soit encore plus d'un facteur 4 en consommation.

3.1.1.3 Lutonium

Le Lutonium [51] est un microcontrôleur universitaire 8051 asynchrone conçu en technologie asynchrone quasi-insensible aux délais au California Institute of Technology. Il a été fabriqué en CMOS 0,18 μm de TSMC en 2003, c'est-à-dire dans une technologie plus récente que celle de MICA, et en *full*

custom. Les caractéristiques tension/vitesse/consommation obtenus avec des simulations bas niveau sont reportées dans le tableau 3.4.

TAB. 3.4 – Caractéristiques du microcontrôleur Lutonium

Tension (V)	**Vitesse (Mips)**	**Consommation (mW)**	**Énergie par instruction (nJ)**
1,8	200	100	0,500
1,1	100	20,7	0,207
0,9	66	9,2	0,139
0,8	48	4,4	0,092
0,5	4	0,17	0,043

En comparant au msp430, on peut remarquer qu'il y a environ un gain en consommation d'un facteur 19 à 4 Mips (10 MHz). À 200 Mips, la consommation par instruction du Lutonium est sensiblement identique à celle du msp430 à 2,2 Mips (5,5 MHz). Le gain en performance est alors supérieur à 90.

3.1.1.4 SNAP

Le processeur SNAP (*Sensor Network Asynchronous Processor*) est un processeur universitaire RISC 16 bits asynchrone QDI conçu spécifiquement pour les réseaux de capteurs sans fil et pour les systèmes multiprocesseur des réseaux sur puce (NoC) [39, 27]. SNAP est muni d'une file matérielle (FIFO) dans laquelle sont stockés des évènements. Ces évènements sont ajoutés à la file soit par le timer lorsqu'il expire, soit par un coprocesseur de messages lorsqu'un paquet provenant de la radio arrive. Les évènements sont traités dans l'ordre. Si un évènement est présent à la tête de la file, le processeur retire cet élément de la queue et l'utilise comme index dans sa table pour exécuter le bon traitant d'évènement. Lorsque la file est vide, le processeur se bloque jusqu'à ce qu'un nouvel évènement arrive. Comme il s'agit d'un processeur asynchrone QDI, il n'y a alors plus aucune activité électrique, le processeur est en veille. Le temps de réveil est négligeable, de l'ordre de quelques nanosecondes. Le tableau 3.5 présente les caractéristiques de ce processeur réalisé en technologie TSMC 0,18 μm en *full custom*.

TAB. 3.5 – Vitesse et consommation du processeur SNAP

Tension (V)	**Vitesse (Mips)**	**Énergie par instruction (pJ)**
1,8	240	218
0,9	61	55
0,6	28	24

La technologie utilisé pour ce processeur est la même que pour le Lutonium. Les vitesses d'exécution sont sensiblement les mêmes à tensions égales. Par contre, au niveau de l'énergie, la consommation par instruction de SNAP est divisée par plus de deux par rapport au Lutonium. Le jeu d'instruction de SNAP est aussi plus modèste que celui du Lutonium.

3.1.1.5 BitSNAP

BitSNAP [28] est une évolution de SNAP dont le cœur du processeur a été modifié pour utilisé un chemin de donnée sérialisé ("bit-serial datapath"). La longueur du chemin de donnée dans le cœur du processeur est alors adaptée en fonction du nombre de bits significatifs de la donnée. Par exemple, la représentation binaire classique de la donnée 16 bits `1111 1111 1101 1100` peut être exprimée par le code $\underline{1}$`01 1100`. Ceci nécessite donc de posséder les délimiteurs {$\underline{0}$,$\underline{1}$} en plus des symboles binaires classiques {`0`,`1`}. Des opérateurs de conversion doivent sérialiser et dé-sérialiser les données, notamment aux interfaces avec la mémoire donnée.

L'utilisation de la technologie asynchrone QDI permet de gérer simplement et efficacement ce codage des quatre symboles {`0`,`1`,$\underline{0}$,$\underline{1}$} grâce à l'utilisation d'un codage 1 parmi 4. Une telle architecture est aussi possible en synchrone et a déjà été étudiée [19, 33]. Cependant, même pour les architectures les plus simples, il est nécessaire d'avoir de multiples horloges, dont une pour le calcul binaire, et une autre pour signaler les bornes de la donnée. Comme la borne peut se trouver n'importe où dans la donnée et peut varier selon que l'on soit en entrée ou en sortie d'un bloc logique (tel qu'une unité arithmétique et logique), la génération des horloges rend sa conception très difficile.

Grâce à l'utilisation d'une telle architecture, le chemin de donnée étant sériel, moins de matériel est nécessaire, ce qui réduit le coût silicium. De plus, ceci permet de compresser significativement les données et donc de réduire globalement la consommation d'énergie. En effet, si l'énergie par bit est un peu plus élevée dans BitSNAP que dans SNAP à cause du coût engendré par le contrôle et la conversion des données sérialisées, la consommation par calcul est sensiblement plus faible car les données sont compressées.

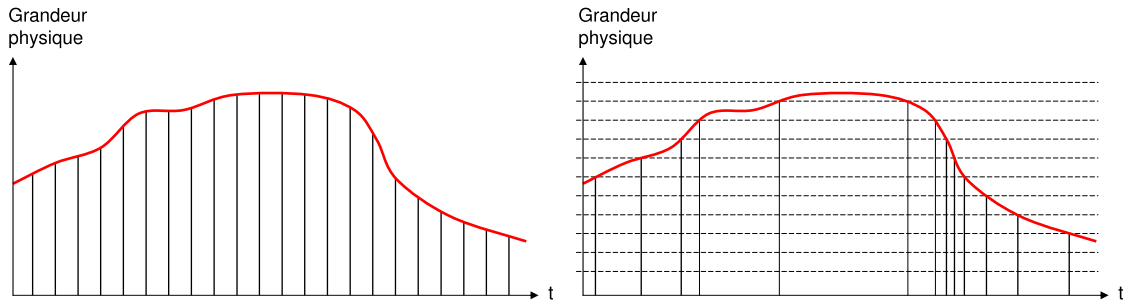
Des évaluations de performances [28] ont été réalisées par simulation, en comparaison de l'architecture parallèle SNAP. Les applications qui ont servi de tests sont des tâches classiquement exécutées dans les réseaux de capteurs sans fil telles que des émissions et réceptions de données, avec l'implantation des protocoles d'accès au médium et de routage. Les résultats montrent que BitSNAP a une consommation par bit équivalente à environ 10 % de l'énergie consommée par le calcul de cette même donnée sur 16 bits par le processeur SNAP. Donc, si les données peuvent être compressée en moins de 10 bits, ce qui est souvent le cas, l'utilisation de BitSNAP permet de réduire la consommation.

En termes de consommation totale, en tenant compte du coût aux interfaces des mémoires, BitSNAP permet de réduire d'environ 30% l'énergie consommée. La consommation moyenne par instruction se situe donc aux environs de 152 pJ par instruction à 1,8 V et 17 pJ par instruction à 0,6 V. Au niveau des performances, BitSNAP fonctionne en moyenne à 54 Mips à 1,8 V et à 6 Mips à 0,6 V.

3.1.2 Convertisseur analogique/numérique et capteur

Pour avoir une utilité pratique, un nœud doit posséder un capteur ou un actionneur. Si on considère l'utilisation d'un capteur, la consommation associée à celui-ci dépend de nombreux paramètres tels la nécessité de polariser le capteur, la nécessité d'utiliser un convertisseur analogique/numérique ou encore la fréquence d'échantillonnage. Tous ces paramètres rendent difficile l'évaluation de la consommation qui est alors très dépendante de l'application considérée.

Cependant, si un convertisseur analogique/numérique est nécessaire, plutôt que d'échantillonner uniformément la grandeur physique, il est plus intéressant d'utiliser un convertisseur analogique/numérique asynchrone [4]. Celui-ci prend un échantillon lorsque la grandeur physique varie de plus d'un quantum d'amplitude prédéfini. La valeur est alors enregistrée en même temps que la date de la mesure. La figure 3.2 montre le principe d'un échantillonnage par traversée de niveaux. Ceci permet de réduire considérablement le nombre d'échantillons, et la consommation. Ensuite, la théorie de traitement du signal sur l'échantillonnage non uniforme peut être utilisée [3, 2]. Par exemple, l'énergie consommée peut être réduite de plus d'un facteur 10 pour une application de traitement de la parole [2].



(a) Echantillonnage uniforme. (b) Echantillonnage non uniforme.

FIG. 3.2 – Comparaison entre un échantillonnage uniforme et un échantillonnage par traversée de niveaux.

3.1.3 Horloge temps réel

Le rôle de l'horloge temps réel est de conserver une information de temps dans le système. Elle peut donc être utilisée par les protocoles de communication pour avoir une heure commune entre les noeuds permettant par exemple de les synchroniser entre eux ; ou bien elle peut servir au système logiciel en tant que "timer" pour réveiller le microcontrôleur à des dates précises.

Comme cette partie matérielle est active en permanence, sa consommation énergétique doit donc être faible pour avoir une durée de vie du nœud acceptable. La consommation typique des horloges temps réel de dernière génération est typiquement de quelques centaines de nano-ampères, ce qui rend sa consommation très comparable aux consommations en veille des autres composants du système tels qu'une radio ou un microcontrôleur synchrone.

3.1.4 Radio

Une partie de cette section est tirée du travail de Franck Paugnat (stagiaire CNAM au TIMA).

3.1.4.1 Caractéristiques

Dans un réseau de capteurs sans fil, la radio est la partie matérielle qui consomme le plus d'énergie. Cette consommation est fonction de nombreux paramètres tels que le débit d'émission, la bande de fréquences utilisée, la sensibilité des récepteurs, la puissance d'émission, etc. Cette section n'a pas pour but d'étudier tous les paramètres influant sur la consommation, mais de donner les ordres de grandeurs caractéristiques des diverses consommations énergétiques mises en jeu par la radio.

Les bandes ISM (industriel, scientifique, et médical) ou les bandes UNII sont des bandes de fréquences libres qui peuvent être utilisées librement (gratuitement, et sans autorisation) pour des applications industrielles, scientifiques et médicales. Ces bandes ne sont pas uniformément réglementées dans le monde. Il y a, au niveau mondial, principalement 4 bandes ISM qui peuvent varier selon les continents :

- la bande des 400 Mhz : cette bande ISM n'est pas disponible aux États-Unis. En Europe, cette bande convient parfaitement aux réseaux de capteurs sans fil à faible débit.
- la bande des 900 Mhz : malgré le fait que les bornes de cette bande ISM varient d'un continent à un autre au point même qu'elles ne se recouvrent pas, cette gamme de fréquence convient bien pour les réseaux de capteurs sans fil.
- la bande des 2,4 Ghz : Cette bande permet d'atteindre des débits relativement élevés. On trouve sur cette bande un grand nombre de normes réseaux telles que le Bluetooth, WLAN (Wifi), ZigBee, les radio-amateurs, ou encore les mmicro-ondes ! Tous ces systèmes ont tendance à créer des collisions ce qui peut être gênant pour les réseaux de capteurs sans fil car il est alors nécessaire de réémettre les paquets perdus ce qui génère une consommation d'énergie supplémentaire.
- la bande des 5 Ghz : cette bande nécessite une modulation à haute fréquence. Elle consomme donc beaucoup d'énergie dans l'état actuel de la technologie silicium. Il n'est donc pas souhaitable pour le moment de l'utiliser dans les nœuds-feuilles des réseaux de capteurs sans fil.

Le tableau 3.6 montre quelques caractéristiques d'émetteurs/récepteurs radio du commerce. Certains de ces composants sont conformes à des normes réseau tels que *Wireless USB* ou *ZigBee*, c'est-à-dire que la couche physique et une partie de la couche MAC sont implantées directement dans le composant. Ceci permet de réduire la complexité du traitement par le microcontrôleur. Ainsi, comme c'est le cas notamment avec le *Chipcon CC2420*, il est possible de configurer le composant radio afin qu'il vérifie s'il y a eu des erreurs de transmission, et acquitte automatiquement les paquets correctement reçus. Ce type de composants permet, de plus, de gérer automatiquement les techniques d'étalement spectrale des normes considérées pour être plus robuste au bruit.

D'autres composants ne sont pas figés à une norme réseau, mais permettent d'effectuer un traitement automatisé sur les paquets reçus ou envoyés. Par exemple, le *CC1100* et le *CC2400* de *Chipcon*

possèdent un mode de gestion automatique des paquets permettant de générer le préambule et le mot de synchronisation, de calculer le CRC d'une donnée lors d'une émission, ou encore de reconnaître le mot de synchronisation et vérifier le CRC lors d'une réception. Inclure ce type de traitement numérique au sein même de la radio pour traiter les paquets permet d'économiser de l'énergie.

TAB. 3.6 – Caractéristiques de différents émetteurs/récepteurs radio.

Marque et modèle	**Bandes ISM**	**Norme réseau**	**Modulation**	**Étalement spectral**
Cypress CYRF6936	2,40 – 2,48 GHz	Wireless USB	GFSK	DSSS
Micrel MICRF600	902 – 928 MHz	—	2-FSK	—
Micrel MICRF620	410 – 450 MHz	—	2-FSK	—
Chipcon CC1020	402 – 470 MHz 804 – 940 MHz	—	OOK, 2-FSK, GFSK	adapté aux sauts de fréq.
Chipcon CC1100	300 – 348 MHz 400 – 464 MHz 800 – 928 MHz	transfert par paquets	OOK/ASK, FSK, GFSK, MSK	adapté aux sauts de fréq.
Chipcon CC2400	2,40 – 2,48 GHz	transfert par paquets	2-FSK, GFSK	adapté aux sauts de fréq.
Chipcon CC2420	2,40 – 2,48 GHz	ZigBee IEEE 802.15.4	MSK	DSSS, FHSS
Ember EM260	2,40 – 2,50 GHz	ZigBee IEEE 802.15.4	MSK	DSSS, FHSS
Atmel AT86RF211S	400 – 480 MHz 800 – 950 MHz	—	FSK	adapté aux sauts de fréq.
Coronis Systems X01 Wavenis	400 – 480 MHz 800 – 960 MHz	—	OOK, FSK, GFSK	adapté aux sauts de fréq.

3.1.4.2 Consommations typiques

Le tableau 3.7 montre la consommation des composants radios décrit dans le tableau 3.6. Comme on peut le remarquer sur les composants pouvant fonctionner dans plusieurs bandes ISM, plus la bande ISM est à des fréquences élevées, plus la consommation est importante. De plus, plus la puissance d'émission est élevée, plus l'émetteur consomme.

On peut constater que l'ordre de grandeur de la consommation à l'émission est environ de 10 mA à 30 mA selon la puissance d'émission. De même à la réception, la consommation est de l'ordre de 15 mA à 20 mA.

Par ailleurs, d'après ce tableau 3.7, il semble que plus le débit d'émission est faible, plus la puissance consommée par le récepteur est faible. Cependant, il faut tenir compte du temps de réception. En effet, plus le débit est élevé, plus le temps d'émission sera court pour des paquets de taille fixe, et donc moins longtemps le circuit consommera. Pour évaluer la consommation, il faut donc trouver une métrique plus

TAB. 3.7 – Consommations typique de différents émetteurs/récepteurs radio à 3 V.

Marque et modèle	**Débit maximum (kbps)**	**Puissance émise (dBm)**	**Conso. émission (mA)**	**Conso. récept. (mA)**	**Conso à l'arrêt (μA)**
Cypress CYRF6936	1 000 (GFSK)	-35 à +4	34 (+4 dBm) 21 (-5 dBm)	21	0,8
Micrel MICRF600	20	-7 à +9	28 (+9 dBm) 14 (-7 dBm)	13,2	0,3
Micrel MICRF620	20	-8 à +10	23 (10 dBm) 10 (-8 dBm)	12	0,3
Chipcon CC1020	153,6	433 MHz : -20 à +10 868 MHz : -20 à +5	433 MHz : 27 (10dBm) 15 (5dBm) 7 (-20dBm) 868 MHz : 25 (5dBm) 14 (-20dBm)	20	0,2
Chipcon CC1100	500	-30 à +10	433 MHz : 29 (10dBm) 13 (-6dBm) 868 MHz : 30 (10dBm) 13 (-6dBm)	16	0,4
Chipcon CC2400	1 000	-25 à 0	19 (0 dBm) 11 (-25 dBm)	24	1,5
Chipcon CC2420	250	-24 à 0	17 (0 dBm) 9 (-24 dBm)	19	0,2
Ember EM260	250	-32 à +3	27 (2 dBm) 20 (-32 dBm)	27	1
Atmel AT86RF211S	100	Selon R ext. Variation de 15 dB. 16 dBm max.	433 MHz : 35 (10dBm) 868 MHz : 42 (10dBm)	24	0,5
Coronis Systems X01 Wavenis	153,6	433 MHz : -20 à +10 868 MHz : -20 à +7	433 MHz : 15 (0dBm) 868 MHz : 25 (7dBm) 17 (0dBm)	18	0,5

adaptée. Une métrique généralement employée est celle du rendement énergétique binaire en émission ou réception, c'est-à-dire le coût énergétique par bit reçu ou transmis (équation 3.1).

$$\eta = \frac{Puissance\,consommée}{Débit\,binaire} \tag{3.1}$$

Le tableau 3.8 montre l'efficacité énergétique binaire en réception de quelques composants. L'efficacité énergétique dépend grandement du débit. En effet, plus le débit est élevé, et plus les composants sont efficaces en énergie. On peut donc penser qu'il est préférable, pour la consommation, d'envoyer les

données rapidement. Si ceci est vrai pour la radio, il faut être en mesure de traiter les données au niveau du microcontrôleur, ce qui peut conduire à surdimensionner ce dernier inutilement, et donc avoir une consommation supplémentaire du processeur.

TAB. 3.8 – Efficacité énergétique et consommations de différents émetteurs/récepteurs radio à 3 V.

Marque et modèle	**Efficacité énergétique en réception**	**Conso. à l'arrêt**	**Conso. avec oscilla-teur HF**	**Conso. avec oscil. HF + polarisation**	**Conso. oscil. HF + polar. + Synth.**
Chipcon CC1020	50 μJ/bit à 1,2 kb/s 600 nJ/bit à 100 kb/s 400 nJ/bit à 150 kb/s	0,2 μA	77 μA	0,5 mA	11,5 mA
Chipcon CC1100	38 μJ/bit à 1,2 kb/s 480 nJ/bit à 100 kb/s 200 nJ/bit à 150 kb/s	0,4 μA	90 μA	1,9 mA	8,7 mA
Atmel AT86RF211S	54 μJ/bit à 1,2 kb/s 650 nJ/bit à 100 kb/s	0,5 μA	150 μA	950 μA	N.D.
Coronis Systems X01 Wavenis	41 μJ/bit à 1,2 kb/s 470 nJ/bit à 100 kb/s 320 nJ/bit à 150 kb/s	0,2 μA	80 μA	500 μA	8 mA

Pour s'activer depuis l'état éteint, une radio passe par ces différentes étapes :

- Démarrage de l'oscillateur externe.
- Polarisation des étages HF de la radio.
- Démarrage du synthétiseur de fréquence, incluant la boucle à verrouillage de phase (PLL).
- Passage en mode émission ou réception.

Chacune de ces étapes prend un temps non négligeable, comme on peut le voir dans le tableau 3.9. Ceci engendre donc une consommation non négligeable à chaque réveil.

TAB. 3.9 – Temps d'établissement de différents émetteurs/récepteurs radio à 3 V

Marque et modèle	**Démarrage du quartz**	**Démarrage du synth. de fréq.**
Chipcon CC1020	0,6 ms – 1,5 ms	0,7 ms – 3,2 ms
Chipcon CC1100	180 μs	90 μs 809 μs avec calibr.
Atmel AT86RF211S	> 8 ms	300 μs
Coronis Systems X01 Wavenis	< 1,5 ms	< 2 ms

Avec les protocoles MAC à échantillonnage de préambules, pour que les nœuds puissent communiquer entre eux, il doivent se réveiller périodiquement pour mesurer la force du signal reçu (RSSI, "Received Signal Strength Indication"). Si le niveau du RSSI dépasse un certain seuil, ceci signifie qu'une émission, et en particulier une émission d'un préambule, est en train de se produire. Dans ce cas, le noeud reste actif en attente de la donnée utile de la transmission. Sinon, il se rendort. Par conséquent, si un nœud veut envoyer une donnée, il doit émettre un préambule au moins supérieur à la période de réveil

de tous les noeuds avant d'envoyer la donnée, pour être sûr que le nœud receveur soit réveillé.

Dès qu'ils détectent un préambule, les nœuds à portée radio restent dans le mode réception, en moyenne, la moitié de la période de réveil. Des travaux permettent de réduire à la fois les phénomènes d'écoute passive et d'écoute inutile ("overhearing") en remplaçant le long préambule traditionnel par une succession de micro-trames [5] permettant à un nœud récepteur de savoir si l'émission le concerne. Si le message ne le concerne pas, il peut alors couper sa radio. Malgré cela, la consomation due à l'écoute passive reste importante.

D'une part, il faut limiter la période de réveil pour limiter le temps où les noeuds sont en mode actif (émission ou réception) lorsque une transmission a lieu. D'autre part, il faut choisir cette période de réveil suffisamment grande pour limiter la consommation due à l'écoute passive, c'est-à-dire lorsque les nœuds se réveillent périodiquement pour sonder le canal. Cette période doit de plus être choisie en fonction du volume de transmission qu'il se produit dans le réseau. A titre d'exemple, pour des applications de type "monitoring" où le volume des données échangées est faible, la période de réveil est de l'ordre de la seconde.

Ceci signifie que, pour une telle application, chaque nœud doit se réveiller toutes les secondes pour sonder le canal. Dans la mesure où le réveil de la radio consomme une énergie non négligeable, un réveil périodique avec une période d'un tel ordre de grandeur coûte cher en énergie. D'ailleurs, ceci est généralement une source importante de consommation dans les réseaux de capteurs. La consommation en veille du nœud peut alors être négligée comparativement.

3.1.5 Conclusion sur l'architecture matérielle

D'après ce qui précède, le msp430 consomme environ entre 14 mW et 23 mW selon la famille. La radio quant à elle, consomme environ 20 mA à 3 V soit une puissance de l'ordre de 60 mW, ce qui correspond une puissance consommée 3 à 4 fois supérieure.

Cependant, il faut généralement plus de temps à ce type de microcontrôleur, avec une vitesse relativement faible, pour décoder la donnée, la traiter et s'occuper des protocoles réseau, que la radio pour recevoir une trame. On peut alors considérer que le rapport de consommation en énergie lorsqu'il se produit une émission ou une réception est donc plutôt de l'ordre de 2. Ceci montre l'intérêt de réduire la consommation liée au microcontrôleur, ou plus généralement au traitement numérique d'un nœud de réseaux de capteurs.

Il faut néanmoins tenir compte, dans le bilan énergétique, de la consommation de la radio due à l'écoute passive du canal pour vérifier, à intervalle régulier, si un nœud à portée radio veut communiquer. Cette écoute inutile nécessite de réveiller la radio qui doit démarrer son oscillateur, polariser son étage HF, synchroniser sa PLL, et mesurer l'énergie présente sur le canal (RSSI). Ceci consomme à chaque fois une certaine quantité d'énergie, de l'ordre de quelques dizaines de μJ. Comme cette opération est répétée régulièrement avec une fréquence relativement élevée, l'écoute passive peut représenter

une part importante dans le bilan énergétique. Par conséquent, plus il y a de communications radio, plus le microcontrôleur doit effectuer de calculs, et plus la proportion d'énergie consommée par celui-ci est importante sur le bilan énergétique global, et plus il est important d'optimiser la consommation de la partie numérique. Bien que non nulle, la consommation des récepteurs radio en veille peut être négligée en comparaison des autres ordres de grandeurs de consommation.

3.2 Architecture logicielle

3.2.1 Intérêt d'un OS pour réseau de capteurs

Les systèmes d'exploitation embarqués généralistes multitâches ont été développés à l'origine pour une plateforme PC et ont ensuite été modifiés ou adaptés à des systèmes embarqués. Ces OS sont trop généralistes pour être suffisamment efficaces, et les changements de contexte et les mécanismes de synchronisation entre processus génèrent un surcoût inacceptable pour des nœuds de réseaux de capteurs sans fil, à la fois en termes de coût processeur que d'occupation mémoire. Pour résoudre ce problème, des systèmes logiciels ont été créés pour cibler particulièrement des applications de réseaux de capteurs.

Une étude comparative [44] entre eCos [53], un OS embarqué généraliste développé par RedHat, et TinyOS, décrit au paragraphe suivant, a montré que l'utilisation de ce dernier permet de réduire considérablement le surcoût qu'engendre l'OS. En effet, comme on peut le voir sur la figure 3.3, pour une application de réseau de capteurs sans fil traditionnelle, la part de l'OS dans le nombre d'instructions exécutées passe de 86 % du total des instructions exécutées pour eCos à 10 % pour TinyOS, soit une réduction du nombre d'instructions exécutées par l'OS d'un facteur 55. Bien que eCos supporte le temps réel et que TinyOS ne le supporte pas, la différence est énorme et montre l'importance d'utiliser un système d'exploitation dédié aux réseaux de capteurs sans fil.

3.2.2 TinyOS

Une manière intéressante pour réduire la consommation dans les réseaux de capteurs au niveau du système d'exploitation a été initialement proposée avec TinyOS [36] développé à l'université de Berkeley. TinyOS est un OS évènementiel qui n'a pas pour but de cibler un large panel d'applications, mais des applications spécifiques de réseaux de capteurs sans fil. Il ne possède pas d'espace kernel-user, ne permet ni allocation dynamique, ni mémoire virtuelle. TinyOS a donc une empreinte mémoire très faible puisqu'il ne prend que 300 à 400 octets dans le cadre d'une distribution minimale. Les zones de la mémoire se résument donc aux zones données initialisées (zone `data`) ou non initialisées (zone `bss`), et à la pile d'exécution.

Le système d'exploitation TinyOS s'appuie sur le langage NesC [31], une surcouche du langage C. Celui-ci propose une architecture basée sur des composants qui sont assemblés statiquement à la compilation, permettant de réduire considérablement la taille mémoire du système et de ses applications.

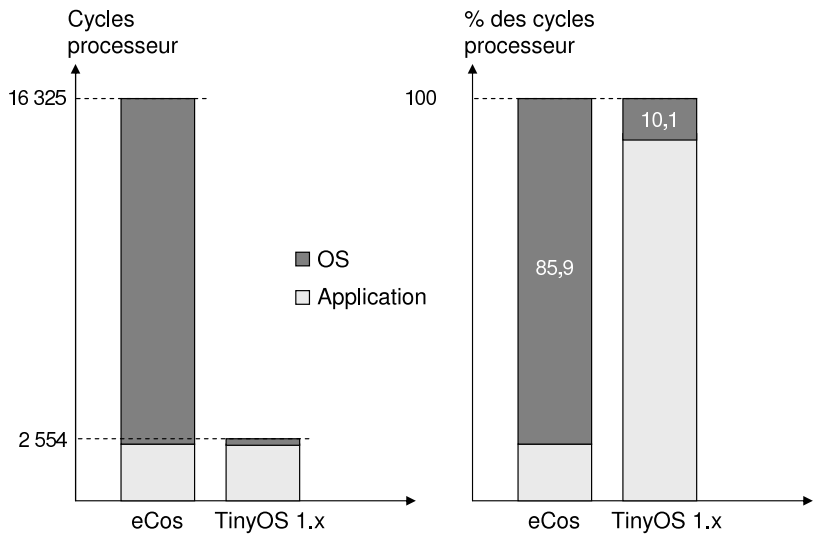


FIG. 3.3 – comparaison des surcoûts engendrés par eCos et TinyOS 1.x pour une application de réseau de capteurs sans fil donnée.

Chaque composant correspond à un élément matériel (LEDs, timer, ADC, radio) et peut être réutilisé dans différentes applications. Ces applications sont des ensembles de composants associés dans un but précis. Les composants peuvent être des concepts abstraits ou bien des interfaces logicielles aux entrées-sorties matérielles de la plateforme. L'implantation de composants s'effectue en déclarant des tâches, des commandes ou des évènements, qui communiquent entre eux au travers d'interfaces.

Les tâches sont utilisées pour effectuer la plupart des travaux de "longue durée" d'une application. A l'appel d'une tache, celle-ci va prendre place dans une file d'attente de type FIFO (*First In First Out*) pour y être exécutée dans l'ordre d'arrivée. Il n'y a pas de mécanisme de préemption entre les tâches et une tâche activée s'exécute en entier. Ce mode de fonctionnement permet de bannir les opérations pouvant bloquer le système (inter-blocage, famine, etc). Par ailleurs, lorsque la file d'attente des tâches est vide, le système d'exploitation met en veille le dispositif jusqu'au lancement de la prochaine interruption (on retrouve le fonctionnement évènementiel).

Les commandes permettent de lancer l'exécution d'une fonctionnalité implantée dans un autre composant. Les commandes sont des appels de fonctions. De même, les évènements sont aussi des appels de fonctions, mais celles-ci ont pour but de signaler qu'un évènement, tel que la fin d'un calcul, s'est produit. Lorsqu'il se produit une interruption matérielle, un évènement nesC est exécuté. Le code nesC ainsi exécuté est prioritaire par rapport aux tâches et peut interrompre la tâche en cours d'exécution.

NesC permet de déclarer deux types de composants : les modules et les configurations. Les modules constituent les briques élémentaires de code et implantent une ou plusieurs interfaces. À plus haut niveau, les configurations sont des composants permettant de faire le lien entre les différents composants (modules ou configurations), et de connecter les interfaces de ces composants entre elles.

Ainsi, comme on peut le voir figure 3.3, TinyOS permet de réduire considérablement l'énergie

consommée et la taille de la mémoire occupée. Pour autant, la bibliothèque de composants de TinyOS est particulièrement complète puisqu'on y retrouve des protocoles de communication, des pilotes de capteurs et des outils d'acquisition de données. L'ensemble de ces composants peut être utilisé tel quel, il peut aussi être adapté à une application précise.

Cependant, un certain nombre d'inconvénients existent, comme par exemple le fait que l'ordonnanceur soit figé ou encore qu'il n'est pas possible de modifier l'ordonnanceur pour qu'il supporte un système de priorités ou des contraintes temps réel.

Dans sa deuxième version, TinyOS essaie de résoudre certains de ces problèmes sans perdre en efficacité. Il permet par exemple de créer un ordonnanceur de tâches différent. Celui-ci est alors un composant nesC comme un autre. La notion de tâche peut aussi être redéfinie en modifiant les interfaces correspondantes, pour y ajouter des échéances de manière à pouvoir ordonnancer les tâches avec un ordonnanceur de type EDF par exemple. Le modèle d'exécution non préemptif reste le même.

3.2.3 Contiki

Contiki [25] est un système d'exploitation conçu pour les systèmes embarqués fortement contraints en mémoire. Celui-ci est composé d'un noyau évènementiel au-dessus duquel les tâches logicielles, implantées avec des *protothreads* [26], sont chargées et déchargées dynamiquement. L'utilisation des *protothreads* s'apparente à celle des "threads" classiques et permet de simplifier le développement des tâches concurrentes. Les communications entre les processus utilisent un système de messages en postant un évènement au noyau. Le noyau ne fournit pas de couche d'abstraction matérielle, mais laisse les processus accéder directement au matériel. Tous les processus partagent le même espace d'adressage et la même pile d'exécution ce qui permet de réduire l'encombrement mémoire. Contiki supporte l'allocation dynamique de la mémoire.

Contiki permet de réduire la consommation d'énergie par rapport à un système d'exploitation traditionnel, tout en gardant une approche de programmation système classique. Cependant, le manque d'étude comparative rend difficile l'évaluation et la comparaison des performances et de la consommation de cet OS vis-à-vis des autres OS.

3.2.4 SOS

SOS [32] est un OS dédié aux réseaux de capteurs sans fil qui se veut être, au contraire de TinyOS, généraliste, dynamique et modulaire. En effet, la motivation principale de SOS est la reconfiguration dynamique, c'est-à-dire la capacité de modifier individuellement le code présent sur chaque nœud après le déploiement et l'initialisation du réseau. Ceci permet de faire des mises à jour incrémentales du système et d'ajouter de nouveaux modules logiciels après le déploiement du réseau.

SOS autorise l'allocation dynamique de la mémoire, aussi bien pour le noyau que pour les modules

applicatifs, mais ne possède pas de mécanisme de protection de la mémoire. L'ordonnanceur non préemptif gère les priorités des tâches, afin d'éviter de traiter les tâches critiques dans un contexte d'interruption.

D'après [32], la consommation d'énergie du processeur est supérieure d'environ 8 % à 10 % par rapport à TinyOS. Cependant, le bilan présenté ne semble pas tenir compte de la taille de la mémoire nécessaire pour faire fonctionner ce système. En effet, si la taille du code en ROM est sensiblement la même par rapport à TinyOS, l'occupation de la mémoire RAM est entre 4 et 5 fois supérieure, ce qui engendre une consommation supplémentaire qui peut s'avérer importante.

Lorsqu'une mise à jour est nécessaire en revanche, la taille du code à modifier dans le cas de SOS, peut être très minime (quelques centaines d'octets). Au contraire, avec TinyOS, une mise à jour coûte cher, non seulement parce qu'il faut propager à tous les noeuds du réseaux la totalité du microcode (qui peut atteindre typiquement quelques dizaines de kilo octets), mais aussi parce qu'il est nécessaire d'écrire la totalité de ce microcode dans la mémoire flash, ce qui consomme beaucoup d'énergie.

SOS permet alors de réduire sensiblement la consommation d'énergie engendrée par une mise à jour du microcode présent sur les nœuds de réseaux de capteurs. Cependant, ce système consomme plus d'énergie en période de fonctionnement. Il n'est donc à recommander que si les mises à jours dans le réseaux sont sensées se produire fréquemment.

3.2.5 Think

Think ("*Think Is Not a Kernel*") [29] est un canevas logiciel dans lequel toutes les entités du système sont des composants. Un composant est une structure qui encapsule des données et un comportement. Ce concept permet une gestion uniforme des ressources (donnée, périphériques, connexions, etc) et des gestionnaires (ordonnanceurs, mémoire, protocoles, etc). Ce modèle est hiérarchique, c'est-à-dire qu'un composant peut être composé d'autres composants plus petits.

Les composants interagissent entre eux au moyen d'interfaces assez similaires à celle de TinyOS. Les interfaces des composants sont interconnectées avec des connexions qui peuvent être implantées de différentes sortes telles que des appels de méthodes classiques, un peu comme TinyOS, mais aussi avec des appels systèmes ou des appels de procédures distantes (RPC). Les liaisons ne relient pas forcément des composants présents physiquement sur une même machine. Dans le contexte qui nous intéresse, le réseau est un réseau sans fil et il convient donc de n'utiliser ces appels distants qu'à bon escient pour ne pas consommer trop d'énergie. Ces interfaces peuvent inclure du traitement, par exemple pour du "monitoring" ou du débogue.

Dans Think, la notion de composant est utilisée jusqu'au plus bas niveau. Autrement dit, même les éléments matériels sont définis sous forme de composants. Cependant, ceux-ci ne fournissent que les fonctionnalités proposées par le matériel afin de rester conforme à la philosophie de Think qui est de n'imposer aucune abstraction système. Ces composants étant fortement liés au matériel sous-jacent, les systèmes les utilisant ne seront donc portables que sur des machines strictement similaires.

Les composants peuvent aussi être gérés dynamiquement par des contrôleurs. En utilisant les interfaces de contrôle de Fractal [14], ceci permet par exemple de donner ou changer l'accès à l'architecture, reconfigurer les connexions, ou encore remplacer les composants. Ces contrôleurs sont optionnels et peuvent être supprimés quand ce contrôle n'est pas nécessaire pour des raisons de performance ou de coût énergétique.

Tout comme TinyOS, Think fournit un ensemble de composants systèmes fréquemment utilisés au travers d'une bibliothèque nommée Kortex. Parmi les services proposés, on peut trouver des ordonnancements, la gestion de la mémoire (plate ou paginée), la gestion réseau ou encore le chargement dynamique de composants.

Un travail de Master effectué à l'université de Pau par Séverine Sentilles montre que Think est une alternative sérieuse à considérer pour la programmation des nœuds de réseaux de capteurs. Malheureusement tout comme Contiki, peu d'études comparatives sont disponibles pour évaluer les performances et la consommation de l'OS.

3.3 Conclusion

Dans un réseau de capteurs, au niveau matériel, la radio est la partie qui consomme le plus d'énergie. Cependant, la consommation du microcontrôleur reste assez conséquente, surtout si le volume de données échangées sur le réseau est important. Il est donc important d'optimiser la consommation de la partie numérique de la plateforme.

Par ailleurs, l'utilisation d'un microcontrôleur asynchrone permet de réduire de manière importante la consommation du logiciel, ou bien d'augmenter nettement les performances du système. Ceci permet d'ouvrir de nouvelles perspectives applicatives aux réseaux de capteurs en permettant d'effectuer des traitements plus importants comme par exemple des traitement d'images en sortie d'un capteur photo.

Surdimensionner un sytème en choisissant un microcontrôleur trop puissant pour une application donnée coûte de l'énergie. Cependant, il est possible dans ce cas de faire fonctionner ce microcontrôleur à une vitesse plus faible, et de dimimuer sa tension d'alimentation. Si le coût en énergie d'une telle approche est faible – ce qui est le cas si on considère les microcontrôleurs asynchrones présentés précédemment – on peut alors utiliser un microcontrôleur assez puissant permettant de garder le même cœur numérique pour toutes les applications de réseaux de capteurs sans fil, ou au moins de limiter le nombre de plateformes. Ainsi, le temps et le coût de développement pour chaque application s'en trouvent réduits. Le coût de fabrication peut aussi être réduit puisque les nœuds sont fabriqués en plus grand nombre.

Au niveau logiciel, l'utilisation d'un système d'exploitation dédié aux réseaux de capteurs ou plus généralement aux systèmes très fortement contraints en mémoire permet de réduire la consommation d'énergie de manière significative. De nombreux systèmes pour réseaux de capteurs existent, parmi les-

quels ont peut citer TinyOS, Contiki, SOS ou Think. Chacun de ces OS a ses spécificités. TinyOS est un système à base de composants décrits en NesC et assemblés à la compilation. Il possède une empreinte mémoire très faible grâce à une gestion statique de celle-ci. Contiki, quant à lui, est modélisé à base de *protothreads* ce qui lui permet de simplifier le développement des tâches concurrentes. SOS a pour objectif de simplifier la maintenance du microcode en minimisant la taille du logiciel à mettre à jour lorsque c'est nécessaire. Quant à Think, il modélise tout le système, à la fois logiciel, matériel, et noyau, sous forme de composants. Il permet par exemple de changer l'ordonnanceur, ou la gestion de la mémoire ; il est donc extrêmement modulaire.

Chapitre 4

Partitionnement logiciel-matériel

4.1 Motivations

Le partitionnement logiciel-matériel est un moyen très efficace de réduire la consommation d'énergie dans les circuits. En effet, concevoir une partie matérielle dédiée à un traitement est toujours bien plus efficace qu'un programme s'exécutant sur un processeur généraliste, aussi bien en termes de nombre de transitions électriques, donc d'énergie consommée, qu'en termes de vitesse de calcul. Néanmoins, cette partie matérielle dédiée a un coût supplémentaire en silicium et il est alors nécessaire de trouver un compromis entre le prix, la performance et la durée de vie du nœud.

Dans un contexte de réseau de capteurs sans fil, on peut notamment considérer le partitionnement d'une éventuelle partie cryptographique, d'une partie dédiée à la localisation, de l'application de codage/décodage de trames, ou encore des protocoles réseau de bas niveaux tels que la couche physique (PHY) et une partie de la couche d'accès au médium (MAC).

4.1.1 Exemple de la radio

Dans les premières architectures de réseaux de capteurs sans fil, la radio fournit un signal binaire à faible débit. Ce signal est alors échantillonné pour définir sa valeur logique. Pour être plus robuste au bruit, il est nécessaire de prendre plusieurs échantillons de la valeur du signal pour filtrer les erreurs. Ainsi, si on considère des transmissions de l'ordre de 20 kbps et un microcontrôleur synchrone fonctionnant à 4 MHz, pour obtenir 10 échantillons par valeur binaire, il faut que le microcontrôleur se réveille tous les 20 cycles d'horloge, rendant inutile la mise en mode faible consommation (voir section 5.1.2). Le microcontrôleur doit alors fonctionner tout le temps de la réception, et consomme beaucoup d'énergie pour échantillonner 10 valeurs.

Il est facile de réduire la consommation d'énergie en implantant une petite partie numérique échantillonnant le signal de la radio à une fréquence de 200 kbps pour obtenir les 10 échantillons qui per-

mettront le filtrage des erreurs. Cette partie matérielle est aussi active tout le temps de la transmission, mais fonctionne à une fréquence moindre et est complètement dédiée, donc optimisée par rapport à un microcontrôleur généraliste. De plus, puisque les paquets reçus par la radio font très souvent plusieurs dizaines d'octets, il est intéressant de communiquer avec le microcontrôleur, non plus au niveau bit, mais au niveau octet, mot de 16 ou 32 bits, ou encore au niveau paquet. Beaucoup d'énergie peut ainsi être économisée.

D'ailleurs, de nombreux industriels proposent des composants radio de plus en plus complexes, implantant e matériel des parties des protocoles de communication bas niveaux tels que la couche physique ou la couche MAC, en permettant en particulier de vérifier s'il y a eu des erreurs de transmission, de les corriger si possible, et d'acquitter automatiquement les paquets qui ont été reçus correctement. Par exemple, comme on l'a vu à la section 3.1.4, le composant Chipcon CC2420, très utilisé dans les réseaux de capteurs sans fil, implante la couche physique et une partie de la couche MAC du standard 802.15.4 du protocole ZigBee. De cette manière, le traitement automatique des données reçues est effectué à moindre coût énergétique et permet au microcontrôleur de rester en mode faible consommation beaucoup plus longtemps.

De même, pour éviter de réveiller le microcontrôleur périodiquement pour sonder le canal et vérifier si une émission va se produire, de nombreux composants radio possèdent un timer configurable permettant de s'auto-réveiller. C'est alors le composant radio qui réveille le microcontrôleur lorsqu'une réception s'est produite.

4.1.2 Chiffrement des données : algorithme AES

Considérons maintenant l'exemple de l'algorithme AES (Advanced Encryption Standard) qui est un algorithme de chiffrement symétrique choisi en octobre 2000 par le NIST (National Institute of Standards and Technology) [55] pour être le nouveau standard cryptographique pour les organisations gouvernementales des États-Unis. Bien que cet algorithme ne soit peut-être pas le plus adapté aux réseaux de capteurs sans fil, il permet de mettre en évidence les problèmes et les bénéfices que l'on peut tirer du partitionnement.

4.1.2.1 Implantation logicielle

Prenons l'exemple de transmissions radio chiffrées par l'algorithme AES avec une clé de 128 bits et des données de 1 ko. Cet algorithme sépare la donnée de 1 ko en paquets de 128 bits, soit 64 paquets en tout.

Supposons que cet algorithme soit exécuté en logiciel sur un microcontrôleur msp430 de Texas Instrument, réputé pour sa faible consommation énergétique et largement répandu dans les architectures de réseaux de capteurs. D'après ses spécifications, à 4 Mhz (3 V), ce microcontrôleur consomme typiquement 0,9 nJ par cycle, une instruction prenant 1 à 6 cycles pour s'exécuter. Selon l'université de

Technologie de Graz qui vend une IP logicielle AES optimisée pour msp430 [56], le chiffrement d'un paquet de 128 bits prend 5432 cycles. Ainsi le chiffrement d'une donnée de 1 ko prendrait 347648 cycles et consommerait donc typiquement 313 μJ en 87 ms.

4.1.2.2 Implantation matérielle

Considérons maintenant un processeur cryptographique AES asynchrone réalisé au sein du laboratoire en technologie 0,13 μm [12] . À sa tension nominale de 1,2 V, avec une clé de 128 bits, celui-ci consomme 12,6 nJ par chiffrement de paquets de 128 bits, en 1 μs. Alimenté en 0,5 V, il consomme 2 nJ et prend 7,5 μs pour faire le même calcul. Par conséquent, le chiffrement d'une donnée de 1 ko coûte entre 132 nJ et 742 nJ en un temps compris entre 66 μs et 481 μs en fonction de la tension d'alimentation.

Bien que le msp430 soit, semble-t-il, réalisé dans une technologie plus ancienne, la différence en termes de consommation et de performance entre l'implantation logicielle et l'implantation matérielle est considérable : il y a un facteur compris entre 420 et 2370 pour la consommation d'énergie, et un facteur compris entre 180 et 1320 pour la vitesse. Ceci montre clairement l'importance du partitionnement dans la conception pour réduire au mieux la consommation des nœuds des réseaux de capteurs.

4.2 Partitionnement et génération d'ordonnancement statique du logiciel

D'une manière générale, il est intéressant, pour le partitionnement logiciel-matériel, de considérer le système dans son intégralité, c'est-à-dire de pouvoir simuler, à partir d'une description haut niveau, son fonctionnement et la consommation de tous les blocs qui le composent pour les optimiser et décider du type d'implantation, et générer un ordonnancement du code logiciel.

4.2.1 Outil de conception de matériel asynchrone : TAST

Afin de palier le manque d'outils de conception pour les circuits asynchrones de type QDI, le laboratoire TIMA a développé l'environnement de conception TAST [23, 22, 21] ("TIMA Aynchronous Synthesis Tool"). Cet environnement comporte un ensemble d'outils d'aide à la conception des circuits asynchrone QDI. L'objectif final de ces outils est de générer, à partir d'une description CHP, un réseau de portes logiques asynchrones.

La figure 4.1 présente le flot de conception des circuits asynchrones implantés dans TAST. Le circuit est décrit initialement en CHP qui représente à haut niveau les communications entre les différents processus. A partir de cette description, l'outil génère un ensemble de structures composé :

– de réseaux de Pétri et de graphes type flots de données. Les réseaux de Pétri permettent de modéliser les états fonctionnels d'un système et les relations qui les lient, avec une complexité algorithmique réduite. Ces graphes permettent de réaliser des vérifications formelles sur le circuit [10], mais également de simuler fonctionnellement le circuit très tôt dans la phase de conception.

– de diagrammes de décision multi-valués qui permettent de réaliser les étapes de synthèse des circuits asynchrones.

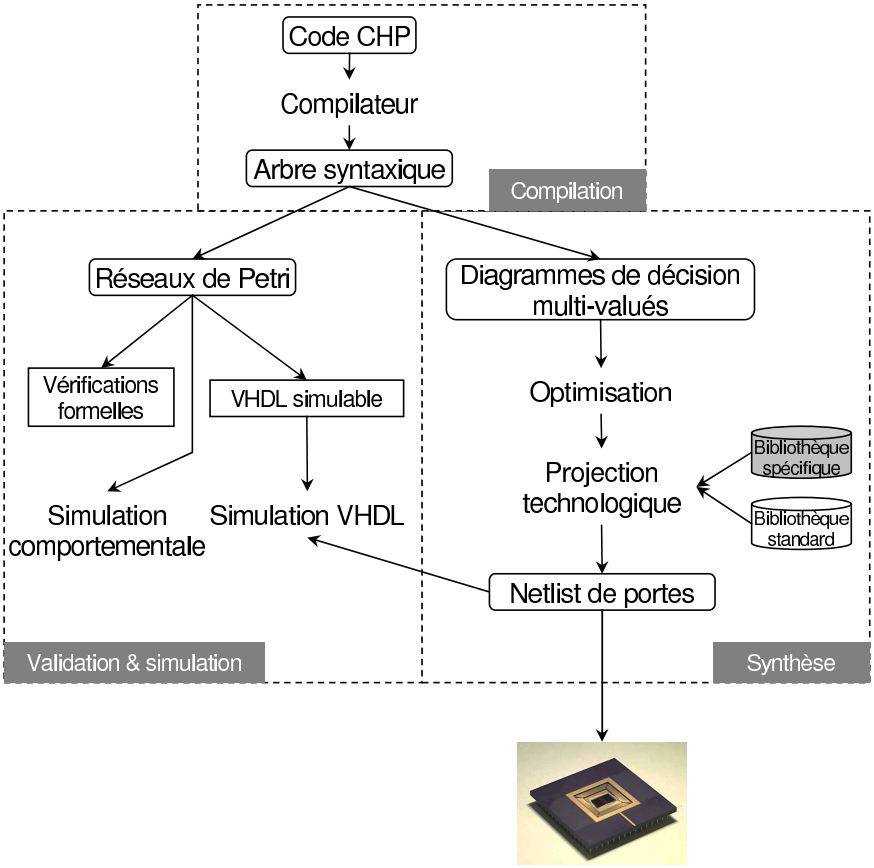


FIG. 4.1 – Flot de conception TAST.

Dans le cadre du partitionnement logiciel-matériel, l'idée est de décrire l'application dans sa globalité en CHP, et de simuler le système. On peut ensuite évaluer, à l'aide des résultats de simulation, l'activité des processus CHP. En fonction des activités, on peut sélectionner les processus qu'il est intéressant d'implanter en matériel du point de vue de la consommation énergétique. Le partitionnement final devra tenir compte, en plus du critère énergétique, du coût en surface du système final et de la capacité à pouvoir faire évoluer la plateforme.

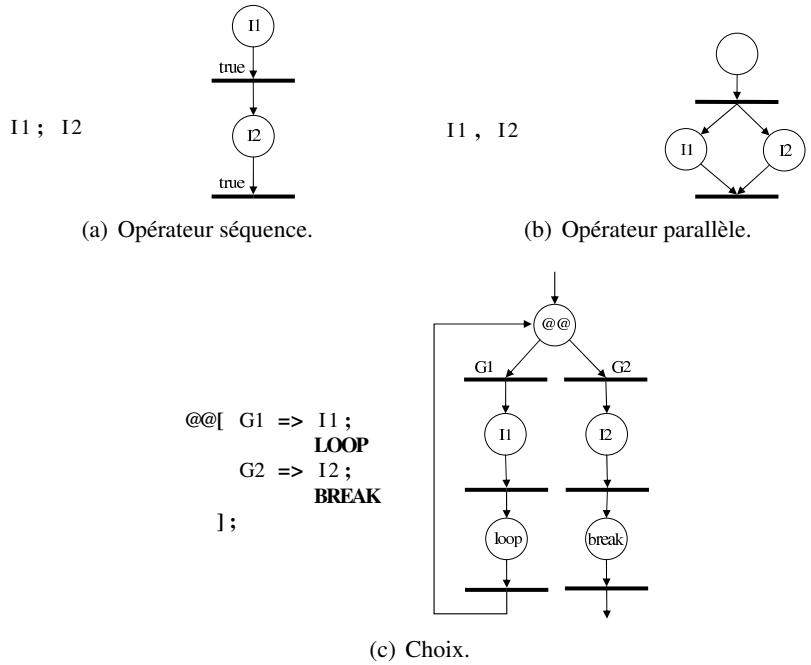
4.2.1.1 Du CHP aux réseaux de Pétri

Présentation générale des réseaux de Pétri Un réseau de Pétri est un graphe orienté bipartie comprenant deux sortes de nœuds : les places et les transitions. Un graphe bipartie se définit comme un graphe $G = (S, A)$ dans lequel S est l'ensemble des sommets (places et transitions) et A l'ensemble des arcs. On représente généralement les places par des cercles et les transitions par des barres. Les arcs d'un réseau de Pétri relient donc les transitions aux places ou les places aux transitions. Les places contiennent des jetons ou marques qui se déplacent de place en place en franchissant les transitions suivant une règle dite de franchissement.

Formellement, un réseau de Pétri est défini par $R = (P, T, W)$, où P est un ensemble fini de places $\{p_1, \ldots, p_m\}$, T est un ensemble fini $\{t_1, \ldots, t_m\}$ de transitions et $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ est la fonction de valuation, $\mathbb{N}$ représente l'ensemble des entiers naturels.

Un marquage est une fonction de $P \rightarrow \mathbb{N}$ qui définit la distribution des marques dans les places. A toute place $p_i \in P$ est associé le nombre de jetons qu'elle contient. On appelle pré-ensemble d'une transition t (*resp.* d'une place p) l'ensemble de places ${}^\bullet t$ (*resp.* l'ensemble des transitions ${}^\bullet p$) dont il existe un arc partant de ces places vers t (*resp.* de ces transitions vers p). De même, on appelle post-ensemble d'une transition t (*resp.* d'une place p) l'ensemble de places $t^\bullet$ (*resp.* l'ensemble des transitions $p^\bullet$) dont il existe un arc partant de t vers ces places (*resp.* de p vers ces transitions).

Le franchissement d'une transition t_j ne peut s'effectuer que si chacune des places de ${}^\bullet t_j$ contient au moins une marque. On dit alors que la transition est franchissable, ou validée. Le franchissement d'une transition t_j consiste à retirer une marque dans chacune des places ${}^\bullet t_j$ et à ajouter une marque dans chacune des places de $t_j^\bullet$.



(a) Opérateur séquence.

(b) Opérateur parallèle.

(c) Choix.

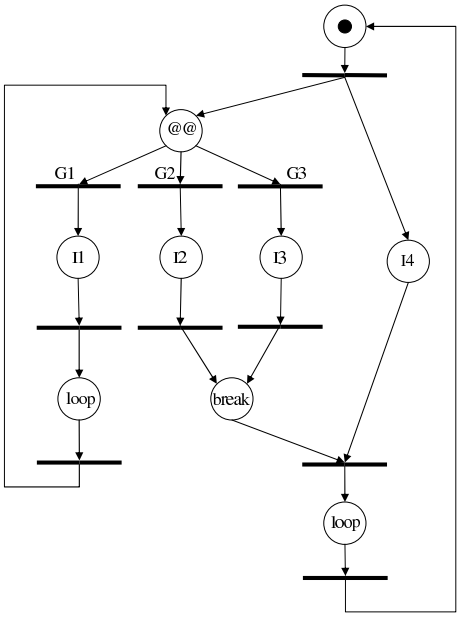
FIG. 4.2 – Génération du réseau de Pétri à partir du code CHP.

Génération des réseaux de Pétri depuis le CHP Dans les réseaux de Pétri générés à partir du CHP, les places correspondent à des instructions alors que les transitions correspondent à des gardes. Dans ce cas, en plus des règles de franchissement de transitions mentionnées plus haut, il faut tenir compte de l'évaluation de la condition présente sur les transitions pour pouvoir franchir celles-ci. La figure 4.2 montre les principales étapes de génération du réseau de Pétri. Ainsi, entre deux places qui représentent des instructions en séquence (figure 4.2(a)), la transition est toujours vraie, tout comme les transitions divergence et convergence de l'opérateur parallèle (figure 4.2(b)). Pour ce qui est du choix, qu'il soit déterministe ou indéterministe, sa représentation est la même (figure 4.2(c)). L'instruction de choix est

sur la place, alors que les gardes `G1` et `G2` sont sur les transitions. Un exemple complet est montré figure 4.3.

```
COMPONENT exemple
BEGIN
  PROCESS exemple
  BEGIN
  [
    @@[ G1 => I1 ; LOOP
        G2 => I2 ; BREAK
        G3 => I3 ; BREAK ] ,
     I4 ;
     LOOP
  ]
  END;
END;
```

(a) CHP code.



(b) Réseau de Pétri équivalent.

FIG. 4.3 – Exemple d'un code CHP et du réseau de Pétri généré.

4.2.2 Génération du code

4.2.2.1 Composition des réseaux de Pétri

Afin d'avoir une modélisation globale du système, il est souhaitable d'assembler les réseaux de Pétri représentant chaque processus communicant au niveau de leurs ports de communication pour former un seul réseau de Pétri.

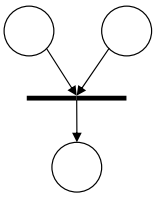


FIG. 4.4 – Représentation d'un rendez-vous à l'aide de réseau de Pétri.

Les communications utilisées intrinsèquement par le matériel asynchrone sont des communications point à point unidirectionnelles et de type synchrone, c'est-à-dire qu'un rendez-vous à lieu entre les processus communicants à chaque communication. Ce type de communication se représente très bien en réseau de Pétri, comme le montre la figure 4.4. Cependant, comme nous l'avons vu précédemment, les instructions sont représentées par des places dans le réseau de Pétri. Il serait alors préférable, pour

la composition, d'avoir justement les communications sur les transitions, afin de pouvoir fusionner les transitions correspondant au même canal de communication entre deux processus.

Il est donc nécessaire de traduire le réseau de Pétri initial en un réseau de Pétri dans lequel les instructions, à l'exception des choix, sont sur les transitions, les gardes restant sur les transitions. Ainsi,la structure du réseau de Pétri obtenue est similaire, comme on peut le voir sur la figure 4.5.

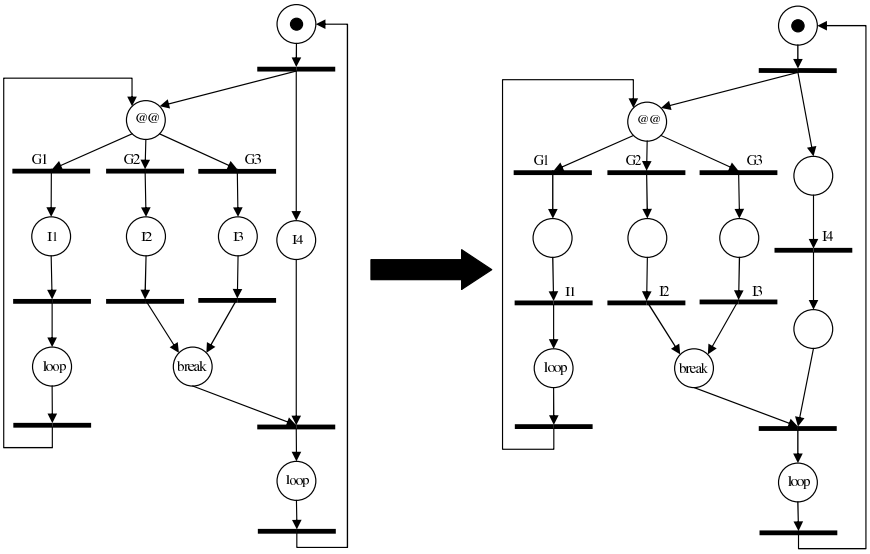


FIG. 4.5 – Modification d'un réseau de Pétri afin d'obtenir les instructions sur les transitions.

Ensuite, pour obtenir la description globale il faut composer les réseaux de Pétri modifiés au niveau des communications. Pour cela, pour chaque canal reliant deux processus, il est nécessaire de faire correspondre chaque émission d'un processus avec toutes les réceptions de l'autre processus connecté à ce canal. Ceci s'effectue en mettant en commun les transisions qui représentent les instructions de communication pour ce canal. La figure 4.6 montre le réseau de Pétri résultant.

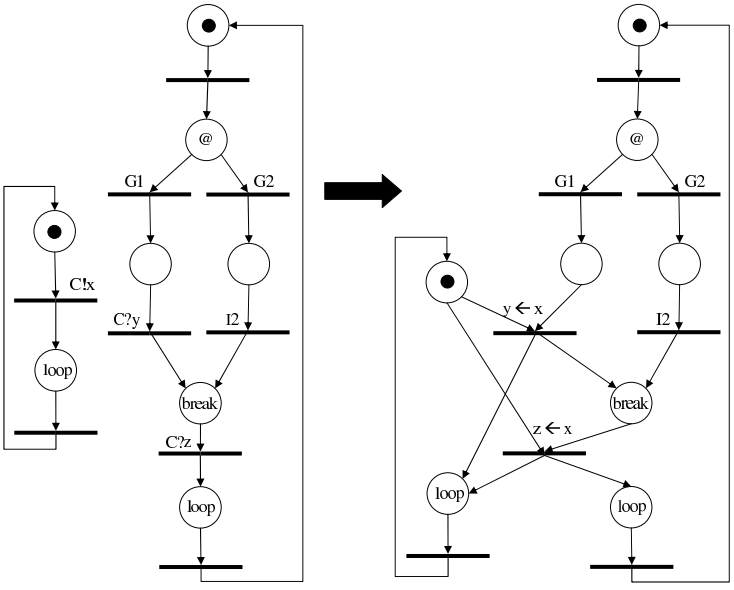


FIG. 4.6 – Composition des réseau de Pétri au niveau des communications.

Comme on l'a vu précédemment section 2.3.5, une "probe" est un booléen présent la plupart du temps dans les gardes, et peut donc être représentée par une condition présente uniquement sur une transition du réseau de Pétri. Néanmoins, l'évaluation de ce booléen ne dépend pas uniquement de l'exécution du processus qui exécute la "probe", mais aussi de l'état du processus connecté à ce canal sondé. Il est alors nécessaire de modéliser le comportement d'une "probe" en explicitant son modèle de fonctionnement en réseau de Pétri, montré figure 4.7.

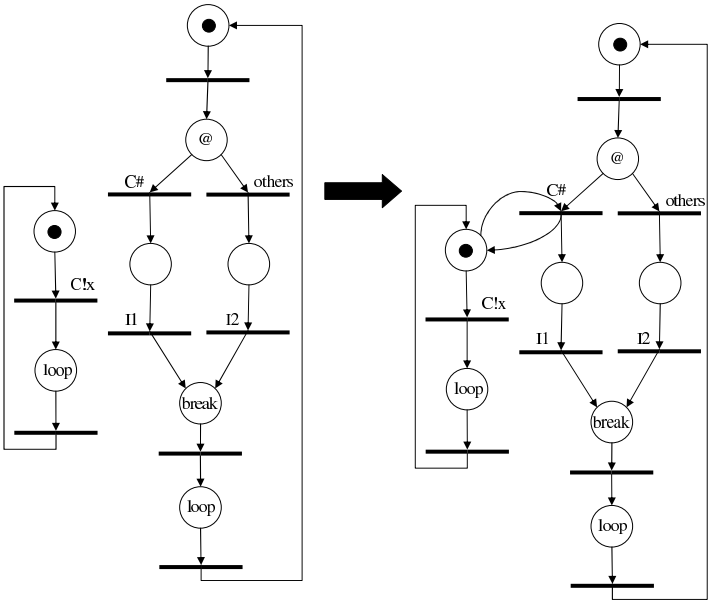


FIG. 4.7 – Modèle des "probes" en réseau de Pétri.

4.2.2.2 Ordonnancement statique du code

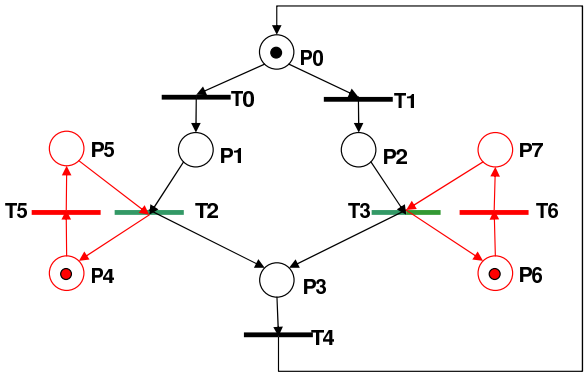
Maintenant que la description de l'application est réalisée de manière globale sous forme d'un seul réseau de Pétri, et que le partitionnement de ce système a été effectué pour définir les processus qui seront implantés en logiciel ou en matériel, il est nécessaire d'ordonnancer les processus logiciels.

Pour être très efficace du point de vue de la consommation, on cherche à effectuer un ordonnancement statique du code. Le problème est alors de transformer la description concurrente à base de réseau de Pétri en description séquentielle exécutée sur un processeur. Des travaux visant à résoudre un problème assez proche ont été réalisés [17]. La principale différence avec notre approche concerne le type de communication utilisé, qui est synchrone dans notre cas, alors qu'il est asynchrone et implanté par des FIFO dans cette étude. Le problème est transposable à notre cas, notamment grâce à l'utilisation d'un réseau de Pétri unique.

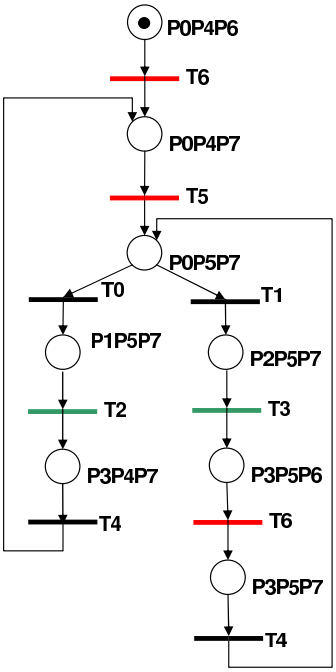
Nous nous plaçons dans un premier temps dans le cas où il n'y a pas de "probe" dans la description du système. La génération d'un ordonnancement statique suppose alors que le réseau de Pétri décrivant l'application ait la propriété "extended free choice", c'est à dire que

$$\forall t_1, t_2 \in T, {}^\bullet t_1 \bigcap {}^\bullet t_2 \neq \emptyset \Rightarrow {}^\bullet t_1 = {}^\bullet t_2.$$

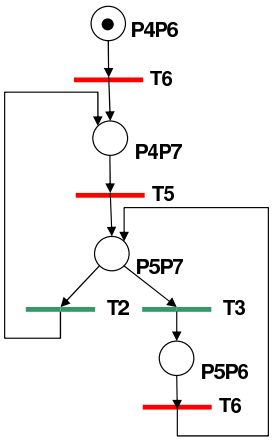
Ceci signifie intuitivement que lorsqu'il y a un choix, les transitions représentant les gardes de ce choix sont toutes sensibilisées en même temps, c'est-à-dire que, indépendamment des valeurs booléennes des gardes, soit toutes les transitions peuvent être franchies en même temps, soit aucune ne peut l'être. En l'absence de "probe", le réseau de Pétri composé de tous les processus est "extended free choice".



(a) réseau de Pétri de l'application



(b) sous-ensemble du graphe de marquage généré par l'algorithme



(c) ordonnancement statique

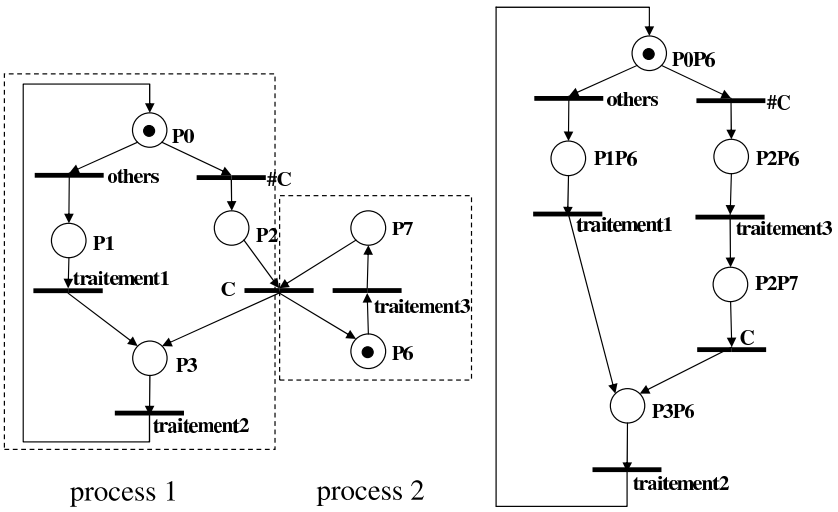
FIG. 4.8 – Génération de l'ordonnancement statique à partir de la description globale du système.

Sans entrer dans les détails — qui peuvent être consultés dans l'étude [17] — l'algorithme de génération de l'ordonnancement part du marquage initial du réseau de Pétri et crée un réseau de Pétri représentant un sous ensemble du graphe de marquage qui garantit l'absence d'interblocage. Dans l'exemple montré figure 4.8, on part du réseau de Pétri de la figure 4.8(a) avec pour marquage initial $P_0P_4P_6$, puis l'algorithme génère le sous ensemble du graphe de marquage (figure 4.8(b)). Ce dernier n'est qu'une exécution concurrente globale possible. Néanmoins, puisqu'on considère un réseau de Pétri "extended free choice" correspondant à une description insensible aux délais, cette approche n'est pas réductrice. Enfin, on crée l'ordonnancement logiciel statique en ne conservant que les transitions correspondant aux

processus logiciels — ici T_2, T_5, T_3 et T_6 — pour être exécutées sur le processeur (figure 4.8(c)). Il est donc possible de générer un ordonnancement statique à partir d'une description CHP d'une application sans "probe".

4.2.3 Le problème des "probes"

La "probe" est un mécanisme permettant au CHP d'être Turing-complet, c'est-à-dire de pouvoir décrire toutes les fonctionnalités de la plupart des langages de programmation usuels. Il peut donc être nécessaire d'utiliser ce mécanisme dans la description CHP d'une application. Cependant, comme on peut le remarquer figure 4.7, la composition au niveau des "probes" fait que le réseau de Pétri n'est plus "extended free choice". Alors, l'algorithme précédent ne permet plus de générer des ordonnancements statiques valides. Pour comprendre ce qui se passe, considérons l'exemple de la figure 4.9 dans lequel les "probes" sont traitées comme des booléens classiques, c'est-à-dire sans les composer (figure 4.9(a)). L'ordonnancement résultant, donné figure 4.9(b), génère un ordonnancement sans interblocage, mais qui exécute le `traitement1` en boucle. Ce comportement n'est pas conforme à la description concurrente de haut niveau.



(a) description d'une application avec des "probes" (b) ordonnancement généré

FIG. 4.9 – Ordonnancement statique non valide généré en présence de "probe".

Ce problème vient du fait que séquentialiser une description concurrente introduit du déterminisme dans l'exécution. Cette déterminisation, combinée au fait qu'il y ait une dépendance d'un processus vis-à-vis de l'autre dans leurs exécutions, engendre ce type de comportement.

4.2.4 Conclusion sur le partitionnement à partir d'une description globale du système

Comme on vient de le voir, la génération d'un ordonnancement statique à partir d'une description globale en CHP d'une application pouvant comporter des "probes", n'est pas possible. Une solution serait alors de ne garder que les processus logiciels et de faire un ordonnanceur dynamique qui, d'une

manière aléatoire, ordonnancerait les processus. Cette solution se rapproche des solutions traditionnelles à base de tâches.

De plus, le CHP est un langage de description matériel bas niveau. Il n'est pas adapté à la description de logiciel. Bien que des travaux de thèse soient en cours pour permettre la description haut niveau de circuits asynchrones à l'aide d'un sous-ensemble de SystemC [40], une description concurrente n'est pas toujours facile à mettre en œuvre.

Enfin, comme nous le verrons dans la partie 5.1.3, l'adaptation dynamique en tension est une technique très importante pour réduire la consommation des circuits mais serait très difficile à implanter avec une telle description haut niveau.

C'est pour toutes ces raisons que cette approche a été abandonnée au profit d'une description disjointe plus classique du logiciel et du matériel.

4.3 Interface logiciel-matériel et synchronisation

Implanter des parties consommatrices d'énergie en matériel permet de réduire l'énergie consommée par une tâche. Cependant, il est important de prendre en compte les interfaces entre le logiciel et le matériel pour réduire le surcoût des communications et ne pas perdre le bénéfice du partitionnement.

4.3.1 Attente active

Le moyen le plus simple pour synchroniser le logiciel et le matériel est de vérifier en boucle qu'une nouvelle donnée est arrivée. Cette solution consomme généralement beaucoup d'instructions inutilement est n'est donc pas souhaitable sur un plan énergétique.

4.3.2 Mécanisme d'interruption

La synchronisation par interruption est le mécanisme de synchronisation utilisé la plupart du temps entre le matériel et le logiciel.

Considérons l'exemple précédent de l'algorithme AES sur msp430. A chaque fois qu'un chiffrement de 128 bits se termine, le processeur cryptographique lève une interruption sur le processeur. Alors, lorsque cette interruption est prise en compte, le contexte d'exécution courant est sauvegardé, c'est-à-dire que les 16 registres du microcontrôleur sont sauvés, un traitant d'interruption est exécuté et les registres sont restaurés. Même s'il est possible d'effectuer des optimisations, le chiffrement d'une donnée de 1 ko représente 64 appels au traitant d'interruption, et à 2048 accès mémoire pour la sauvegarde et la restauration des registres. Rien que pour la sauvegarde et la restauration des registres, en considérant 5 cycles en moyenne pour une sauvegarde/restauration de registres, il faut environ 10240 cycles ce

qui représente une consommation typique de plus de 9 μJ. Ainsi, le mécanisme de synchronisation par interruption consomme, sans tenir compte de la consommation du traitant d'interruption appelé 64 fois, entre 12 et 68 fois plus que la tâche de chiffrement elle-même.

Toutefois, bien que les communications entre le logiciel et le matériel représentent la plupart de l'énergie consommée, il demeure très avantageux d'exécuter les chiffrements AES en matériel.

4.3.3 Communication synchrone

Comme nous l'avons vu dans le chapitre 2, le matériel asynchrone utilise naturellement des communications synchrones entre les différents blocs matériels asynchrones. Ces communications synchrones, ou rendez-vous, peuvent être utilisées avantageusement pour réduire le surcoût engendré par les communications logiciel-matériel.

En effet, grâce au rendez-vous, il est possible d'effectuer une lecture bloquante à une adresse mémoire correspondant à un registre de sortie d'un périphérique. En fait, le processeur demande la donnée avec une requête, et attend l'acquittement. Une fois le calcul terminé, le périphérique écrit le résultat dans son registre de sortie. L'acquittement étant codé avec les données, la communication est ainsi débloquée, et le logiciel continue son exécution. Un schéma de l'architecture est montré figure 4.10.

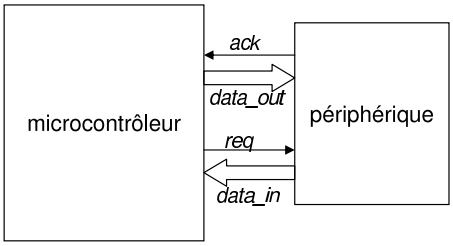


FIG. 4.10 – Architecture de communication synchrone entre le processeur et un périphérique.

Ce type de synchronisation permet d'annuler le surcoût temporel et énergétique des communications entre le logiciel et le matériel. Néanmoins, pendant que le périphérique effectue son calcul, le microcontrôleur attend le résultat (attente passive) et ne peut exécuter d'autres tâches qui seraient éventuellement prêtes à l'être. Ceci peut poser problème lorsqu'une interruption qu'il faut traiter rapidement – provenant de la radio par exemple – est levée. Dans ce cas, il faut attendre la fin du calcul du périphérique pour que le processeur se débloque et puisse traiter cette interruption.

Cette solution reste plus efficace en performance qu'une solution purement logicielle puisque du matériel dédié exécute une tâche beaucoup plus rapidement qu'avec une description logicielle sur un microcontrôleur généraliste. Néanmoins, ce mécanisme de communications synchrones doit être amélioré pour permettre la concurrence entre le processeur et le périphérique, notamment si on considère un système temps réel.

Une façon de résoudre ce problème est de définir une nouvelle instruction de lecture en mémoire qui

permet de prendre en compte les interruptions qui peuvent survenir lorsque le processeur est bloqué sur une lecture. Ceci nécessite aussi une petite modification de l'architecture au niveau des bus de données en insérant un module sur le canal de communication entre un périphérique et le processeur (figure 4.11).

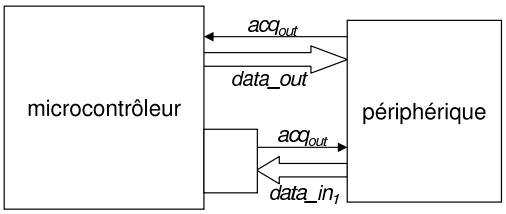


FIG. 4.11 – Architecture de communication synchrone interruptible entre le processeur et un périphérique.

Ce module permet au processeur de débloquer la communication synchrone sur laquelle il est bloqué lorsqu'un évènement tel qu'une interruption est levée. Il peut alors traiter l'évènement, avant d'effecuter de nouveau la lecture bloquante.

Cette solution supprime le coût des communications entre le logiciel et le matériel tout en permettant de traiter les interruptions. Ceci permet d'améliorer le niveau de concurrence entre le logiciel et le matériel. En particulier, ce type de module permet de réduire la consommation du système tout en permettant de respecter l'ordonnancement et les contraintes de tâches temps réel. Ce point sera détaillé dans le chapitre 7.

4.4 Conclusion

En permettant d'implanter des tâches très consommatrices d'énergie en matériel, le partitionnement logiciel-matériel est un moyen très efficace pour réduire la consommation du nœud. Néanmoins, il est important de prendre en compte les communications entre le logiciel et le matériel qui peuvent représenter une part très importante de la consommation.

Le matériel asynchrone permet d'avoir des communications par rendez-vous. Ce modèle basé sur un protocole de type "poignée de mains" permet de synchroniser le logiciel et le matériel à coût nul. Néanmoins, ce mécanisme n'autorise pas la concurrence entre le logiciel et le matériel. Il est alors nécessaire de modifier l'architecture pour autoriser les communications bloquantes à être interrompues. Le tableau 4.1 récapitule l'énergie consommée par une application de chiffrement AES en fonction du partitionnement et du type de communication choisi. Il montre en particulier qu'un partitionnement logiciel-matériel judicieux et un système de communication synchrone entre le processeur et ses périphériques permettent de gagner un facteur compris entre 421 et 2371 suivant la tension d'alimentation, tout en ayant des performances bien supérieures.

TAB. 4.1 – Récapitulatif de l'énergie consommée par un chiffrement AES d'un ko de donnée.

Implantation	**Chiffrement**		**Coût des comm.**	**Conso. totale**	**Temps de calcul total**
	128 bits	**1 ko**			
Logicielle	4 889 nJ	313 μJ	-	313 μJ	87 ms
Matérielle avec interruptions	2→12,6 nJ	132→742 nJ	9 μJ	9,1→9,7 μJ	66→481 μs + 2,5 μs (comm.)
Matérielle avec lectures bloquantes	2→12,6 nJ	132→742 nJ	0	132→742 nJ	66→481 μs

Chapitre 5

Gestion de l'énergie

Il existe de nombreux mécanismes pour réduire la consommation d'énergie dans les circuits, tels que le "clock gating", les modes faible consommation ou encore l'adaptation dynamique des tensions d'alimentation et de polarisation du substrat. Ces mécanismes sont, la plupart du temps, des mécanismes matériels gérés par le logiciel à plus haut niveau. Nous allons présenter, dans ce chapitre, les principes de ces mécanismes et des implantations matérielles pour les mettre en œuvre.

5.1 Techniques pour réduire la consommation d'énergie

5.1.1 Le "clock gating"

Dans un circuit synchrone, la synchronisation entre les blocs est réalisée à l'aide d'une horloge. Cette horloge permet de définir la validité des données en entrée de chaque bloc. Sans précaution particulière, ce signal est propagé dans tous les blocs du circuit, qu'ils soient en train d'effectuer un calcul valide ou non. Ainsi, lorsqu'il n'y a pas de donnée utile à l'entrée d'un bloc, celui-ci va commuter inutilement provoquant une consommation inutile.

Le "clock gating" est une technique permettant de couper les parties de l'arbre d'horloge reliées à des blocs n'effectuant pas de calcul. Ceci permet de réduire la consommation d'énergie dans les circuits synchrones, mais n'est pas toujours facile à mettre en œuvre.

5.1.2 Modes faible consommation

Lorsqu'un processeur synchrone n'a plus de tâche logicielle à exécuter, celui-ci peut se mettre en attente d'interruptions qui le réveilleront. Dans ce cas, bien qu'il n'y ait aucun calcul à réaliser, l'horloge interne continue à fonctionner, ce qui génère une consommation totalement inutile. C'est pour cette raison que les modes faible consommation ont été créés. Ces modes sont contrôlés par le logiciel et

permettent de couper les différentes horloges du circuit pendant que le processeur attend une interruption. L'économie d'énergie réalisée peut être très significative.

Par exemple, si on considère le msp430 de Texas Instrument et l'ATMega164 d'Atmel, d'après leurs fiches techniques, le passage de leurs modes actifs respectifs à leurs modes de veille les moins consommants permet de faire passer la consommation typique, à 3 V et 4 MHz, de 1200 μJ pour le msp430 et de 2100 μJ pour l'ATMega164, à respectivement 0,1 μJ et 0,2 μJ. Ces rapports en consommation, de 12 000 et 10 500 respectivement, sont très conséquents et justifient l'utilisation de ces modes (tableau 5.1). La différence de consommation entre ces deux microcontrôleurs est due au fait que l'ATMega164 est beaucoup plus évolué que le msp430. En particulier, l'ATMega164 exécute la plupart de ses instructions en un cycle, alors qu'une instruction peut prendre 6 cycles avec le msp430.

TAB. 5.1 – Latence et énergie consommée typiques pour retourner du mode de veille profonde vers le mode actif avec les microcontrôleur TI msp430 et Atmel ATMega164.

Microcontrôleur	**horloge**	**latence**	**consommation**	
			énergie	**#instr. éq.**
TI msp430x2xx	interne	4,9 μs	18 nJ	20
Atmel ATmega164P	interne	5,75 μs	36 nJ	23
	céramique	254 μs	1,6 μJ	1 019
	quartz	4 ms	25 μJ	15 923

Néanmoins, cette économie se fait aux dépends de la réactivité du système, puisque la redémarrage des horloges coupées lors des phases de réveil introduit des latences qui peuvent être importantes. Ces latences dépendent du type d'horloge utilisé. S'il s'agit de l'horloge interne au microcontrôleur, tel qu'un oscillateur RC, la latence se situe typiquement à 4,9 μs et 5,75 μs, suivant le microcontrôleur, pour revenir du mode de veille profonde en mode actif. Avec un résonateur céramique, la latence de l'ATMega164 passe à 254 μs, et à 4 ms avec un quartz haute fréquence. Le passage du mode actif en mode de veille profonde s'effectue, quant à lui, presque instantanément pour ces deux microcontrôleur (1 à 2 cycles). Attention cependant à la précision de ces horloge : bien que plus rapide à démarrer, l'oscillatur RC est très peut précis et varie grandement en fonction des conditions d'utilisation (tension, température).

À ces latences s'ajoute une consommation supplémentaire, due à la synchronisation des horloges, qui dépend encore du type d'horloge. Ainsi, avec l'oscillateur interne, le passage du mode de veille profonde en mode actif consomme typiquement 18 nJ pour le msp430 et 36 nJ pour l'ATMega164, soit l'équivalent de l'énergie consommée par environ une vingtaine d'instructions. Avec un résonateur céramique il faut compter l'équivalent d'un millier d'instructions, et un équivalent de presque 16 000 instructions avec un quartz haute fréquence (tableau 5.1).

Il est alors évident que l'utilisation de l'horloge interne du microcontrôleur permet de gagner énormément en énergie et en temps. Néanmoins, pour des raisons de performances, il peut être nécessaire d'augmenter la fréquence de fonctionnement. Par exemple, le msp430x2xx considéré ne peut pas fonctionner à plus de 16 MHz, ce qui correspond généralement à une vitesse moyenne de l'ordre de 6 Mips. Pour certaines applications, cette vitesse maximale peut s'avérer insuffisante. Si on considère l'ATMega164,

celui-ci peut fonctionner à 8 MHz avec son oscillateur interne, soit près de 8 Mips, mais peut monter jusqu'à 20 MHz grâce à un quartz externe. Dans ce cas, les temps de latence et l'énergie consommée pour synchroniser ce type d'horloge lors du réveil du microcontrôleur posent un véritable problème.

Par ailleurs, les chiffres reportés dans le tableau 5.1 ne prennent en compte que les modes de veille profonde les moins consommants de chaque microcontrôleur, par rapport à leurs modes actifs respectifs. Cependant, il existe généralement plusieurs modes faible consommation. Plus un mode permet d'arrêter des horloges dans le circuit, moins celui-ci consomme d'énergie dans ce mode, mais plus il va mettre de temps pour resynchroniser ces horloges, en entraînant une consommation supplémentaire ponctuelle au réveil. Il se pose donc le problème de la gestion de ces modes au niveau logiciel puisqu'on n'a, a priori, aucune information sur les dates d'arrivée des évènements. Il est par conséquent très difficile de prévoir la durée de la veille et donc le mode qui minimisera la quantité d'énergie consommée au total. C'est pourquoi seulement le mode actif et le mode de veille profonde sont généralement utilisés. De l'énergie et du temps sont donc gaspillés.

5.1.3 Adaptation dynamique de la tension d'alimentation

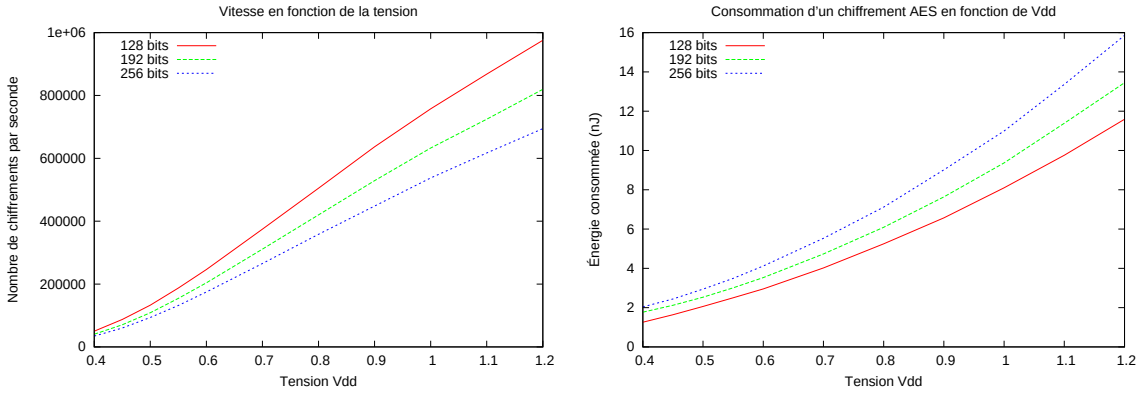
Un moyen efficace pour réduire la consommation énergétique est d'utiliser l'adaptation dynamique en tension (Dynamic Voltage Scaling ou DVS) pour contrôler dynamiquement l'énergie consommée. Dans un circuit CMOS, l'énergie consommée E est proportionnelle au carré de la tension V_{dd} et à la capacité totale commutée C (équation 5.1). La variation de la vitesse est donnée, quant à elle, équation 5.2. Les paramètres α et β sont des constantes technologiques positives. Dans la mesure où β est généralement proche de 1 dans les technologies assez récentes, on peut considérer que la vitesse S du processeur varie approximativement linéairement avec la tension d'alimentation.

$$E \propto C \times V_{dd}^2 \tag{5.1}$$

$$S \propto (V_{dd} - \alpha)^{\beta} \tag{5.2}$$

Ainsi, une faible diminution de la tension ralentit faiblement le circuit mais permet de réduire de manière importante la consommation d'énergie. Des résultats expérimentaux réalisés sur le processeur cryptographique AES [12] (0,13μm) mentionné plus haut sont montrés figure 5.1 ; ceux concernant un microcontrôleur asynchrone 8 bits CISC en technologie 0,25μm sont montrés figure 5.2.

Si on a le temps d'effectuer un calcul, on peut alors réduire considérablement la consommation d'énergie, et permettre ainsi de passer, dans l'exemple du chiffrement AES 128 bits, d'une consommation de 11,6 nJ par chiffrement à 1,2 V, à une consommation de 1,25nJ à 0,4 V, soit une réduction d'un facteur proche de 10.



(a) nombre de chiffrements par seconde (b) consommation énergétique

FIG. 5.1 – Vitesse et de consommation d'un coprocesseur cryptographique AES asynchrone.

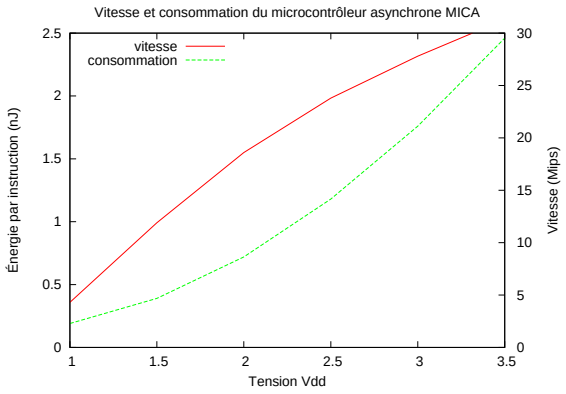


FIG. 5.2 – Vitesse et consommation en fonction de la tension d'alimentation pour le microcontrôleur asynchrone MICA.

5.1.4 Polarisation du substrat

Comme on vient de le voir, il est possible de diminuer la consommation dynamique du circuit en abaissant la tension d'alimentation V_{dd}. Les dernières technologies permettent d'utiliser des transistors de plus en plus petits, ce qui tend à faire diminuer la consommation dynamique. En revanche, la puissance statique devient une source de consommation importante, et il est primordial de pouvoir la diminuer, surtout dans un contexte de réseaux de capteurs.

Ceci est rendu possible grâce à la technique de polarisation du substrat ("Adaptative Body Biasing", ABB). Cette technique consiste à dissocier le substrat des transistors et leurs sources afin d'y appliquer une tension V_{bs} et réduire, à V_{dd} constant, le courant de fuite $I_{ds_{leak}}$ entre le drain et la source. On diminue ainsi l'énergie statique (équation 5.3).

$$E_{stat} = \int_t V_{dd} \times I_{ds_{leak}}.dt \tag{5.3}$$

D'après [41, 52], il est possible de définir un modèle du courant de fuite, donné dans l'équation 5.4.

Comme on peut le remarquer, le courant de fuite dépend aussi de V_{dd}. Les paramètres a, b et c sont positifs et dépendent de la technologie utilisée.

$$I_{ds_{leak}} = a.e^{b.V_{dd}}.e^{-c.V_{bs}} \tag{5.4}$$

Ainsi, à V_{dd} fixe, plus on augmente V_{bs}, plus les courants de fuite diminuent. Au contraire, à V_{bs} fixe, plus on augmente V_{dd}, plus les fuites augmentent. La figure 5.3 montre la valeur normalisée des courants de fuite en fonction de V_{bs} et V_{dd}.

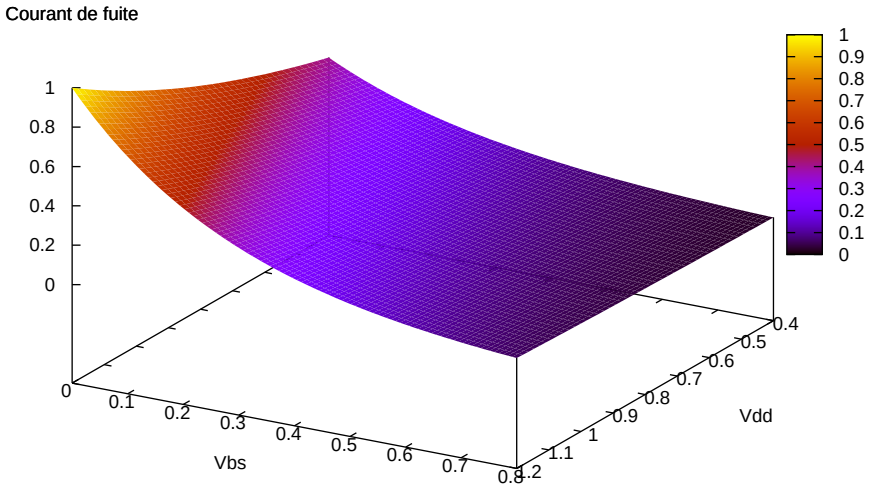


FIG. 5.3 – Courant de fuite $I_{ds_{leak}}$ normalisé en fonction de V_{dd} et V_{bs}.

Par ailleurs, polariser le substrat influence la vitesse de commutation des transistors. D'après [41, 52], il est possible de modéliser la vitesse de commutation des transistors par l'équation 5.5, représentée graphiquement figure 5.4.

$$S \propto (V_{dd} - \alpha_1 - \alpha_2.V_{bs})^{\beta} \tag{5.5}$$

Les paramètres α_1, α_2 et β sont des constantes technologiques positives. On retrouve bien, à $V_{bs} = 0$, l'équation 5.2. À V_{dd} constant, la vitesse du circuit diminue lorsque l'on augmente V_{bs}. On peut remarquer que le fait de diminuer la tension d'alimentation V_{dd} par trois, de 1,2 V à 0,4 V, et d'augmenter V_{bs} de 0 V à 0,8 V permet de diminuer par 40 le courant de fuite, et diminue par 120 la puissance statique pour la technologie considérée. Dans le même temps, la vitesse du transistor est divisée par 2, ce qui fait que l'énergie statique consommée pour un calcul donné est divisée par 60.

L'optimisation dynamique conjointe des techniques du DVS et de l'ABB est complexe et laissée pour une étude future. Toutefois, il est possible d'utiliser la technique du DVS dynamiquement, et de ne polariser le substrat que pendant les périodes de veille, qui sont importantes dans les applications

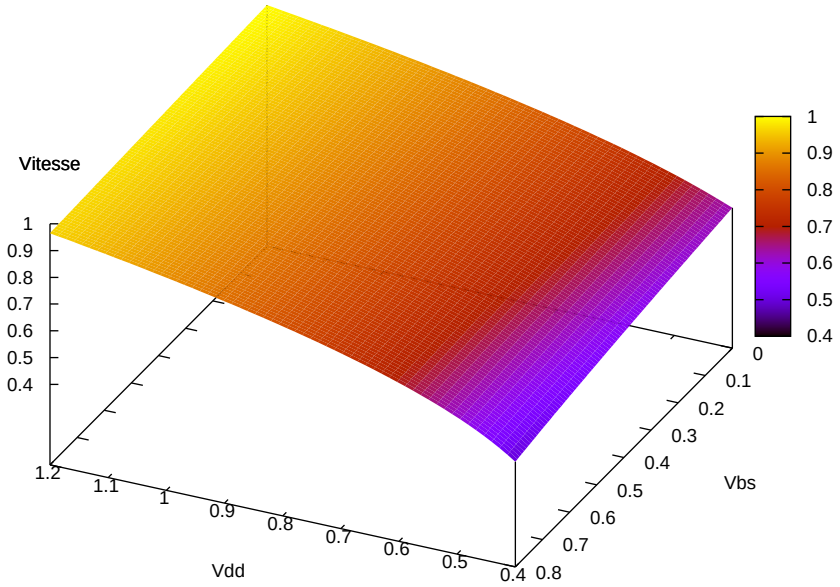


FIG. 5.4 – Vitesse normalisée d'un circuit en fonction de V_{dd} et V_{bs}.

considérées, pour diminuer la consommation d'énergie statique. C'est ce que nous considérerons dans la suite du document, et c'est pourquoi nous ne nous préoccuperons plus de la technique de l'ABB.

5.2 Intérêt de l'asynchrone pour la gestion de l'énergie

La technique du "clock gating" ou les modes de faible consommation permettent de réduire localement ou globalement l'activité dans les circuits synchrones. Dans les circuits asynchrones, du fait qu'il n'y a pas d'horloge, ces techniques sont inutiles. L'implantation locale du protocole de communication entre les blocs permet de réduire intrinsèquement l'activité inutile du circuit, et ceci à coût temporel et énergétique nul.

En ce qui concerne les techniques d'adaptation des tensions d'alimentation et de substrat, celles-ci sont valables aussi bien pour les circuits synchrones que les circuits asynchrones. Cependant, lorsque l'on change la tension d'alimentation d'un circuit synchrone, il est nécessaire de changer conjointement la fréquence de fonctionnement. Ainsi, si on veut abaisser la tension, il est nécessaire de diminuer au préalable la fréquence du circuit et attendre que l'horloge se stabilise. Ceci est une contrainte qui peut coûter cher en temps et en énergie selon les architectures des circuits considérés. Avec les circuits asynchrones en revanche, puisqu'il n'y a pas d'horloge globale, il n'y a aucun surcoût engendré suite à un changement de tension. La vitesse globale du circuit s'adapte en fonction du temps d'établissement des signaux internes.

Par ailleurs, les circuits asynchrones QDI sont, par définition, insensibles aux délais d'établissement des signaux au sein du circuit. Il ne sont donc pas sensibles aux variations de délais dans les chemins

de données, contrairement aux circuits synchrones. Ainsi, les circuits asynchrones sont beaucoup plus robustes aux variations de process et au bruit sur l'alimentation que les circuits synchrones et permettent donc de descendre plus bas en tension. Le tableau 5.2 montre les tensions d'alimentation de trois processeurs synchrones. On peut remarquer que la tension minimale d'alimentation pour de tels processeurs est généralement assez élevée, notamment si on la compare à la tension minimale du processeur cryptographique AES présenté figure 5.1 qui est de 400 mV, c'est-à-dire en dessous de la tension de seuil des transistors en technologie HCMOS9 0,13 μm de ST Microelectronics, qui est d'environ 450 mV.

TAB. 5.2 – Exemple de variation de tension et de vitesse de 3 processeurs synchrones.

Processeur	**Plage de tension**	**Plage de fréquence**
IBM PowerPC405LP	1,0 V - 1,8 V	152 MHz - 380 MHz
TransMeta Crusoe TM58000	0,8 V - 1,3 V	300 MHz - 1 GHz
Intel XScale PXA261	1,1 V - 1,43 V	200 MHz - 400 MHz

5.3 Coprocesseur DVS

5.3.1 Motivation

Du fait qu'ils sont souvent inactifs, les processeurs sont des cibles idéales pour utiliser la technique du DVS. Avec un processeur synchrone, il est nécessaire de connaître la correspondance entre la tension d'alimentation et les vitesses de fonctionnement du circuit pour que le circuit puisse fonctionner lorsque la tension est abaissée. Ceci oblige donc à caractériser les circuits en fonction du pire cas temporel, et en gardant une marge pour faire face au problème de variation technologique.

Pour un processeur asynchrone, le problème est un peu différent. En effet, il n'est pas nécessaire d'avoir une correspondance entre la tension et la vitesse pour que le circuit soit fonctionnel lorsque la tension est abaissée. Cependant, au niveau logiciel, on aimerait savoir quelle tension appliquer pour garantir une vitesse donnée. On peut alors, comme pour le synchrone, caractériser le processeur en fonction des niveaux de tensions. Cette caractérisation doit tenir compte de pires cas au niveau des instructions, mais aussi au niveau des données puisque les temps d'exécution dépendent aussi des données d'entrée. Ceci revient à considérer que toutes les instructions prennent le même temps que le pire cas. On perd alors un des intérêts des circuits asynchrones puisque l'on considère alors un comportement temporel en pire cas, et non plus un comportement moyen. Cette caractérisation pire cas ne serait jamais atteinte, si bien que ceci reviendrait à définir une vitesse d'exécution trop élevée, et donc à exécuter une application logicielle toujours plus rapidement que ce qu'elle pourrait être idéalement. De l'énergie serait donc gaspillée.

Une autre solution, que nous explorons dans la suite de ce chapitre, consiste à asservir la tension d'alimentation en fonction des besoins du logiciel. En effet, une application, ou plus généralement le système d'exploitation, définit une consigne de vitesse, exprimée par exemple en milliers ou millions d'instructions par secondes (kips ou Mips), et la donne en paramètre à un coprocesseur qui sera alors

chargé d'asservir la tension d'alimentation pour que le processeur fonctionne à cette vitesse. Ainsi, en mesurant la vitesse réelle du processeur, on a un comportement en vitesse moyenne pendant l'exécution du logiciel.

5.3.2 Principe de la régulation

La figure 5.5 montre le schéma d'asservissement d'un tel coprocesseur DVS qui contrôle la tension d'alimentation du processeur en fonction d'une consigne logicielle. Lorsque le processeur termine l'exécution d'une instruction, il en informe le coprocesseur, via le signal de fin d'instruction, qui mesure la vitesse réelle du processeur. Cette vitesse est alors comparée à la consigne générée par le logiciel, permettant d'asservir la tension d'alimentation du processeur en formant un système de régulation en boucle fermée. Les problématiques liées à la génération de la consigne logicielle seront abordées dans le chapitre 6.

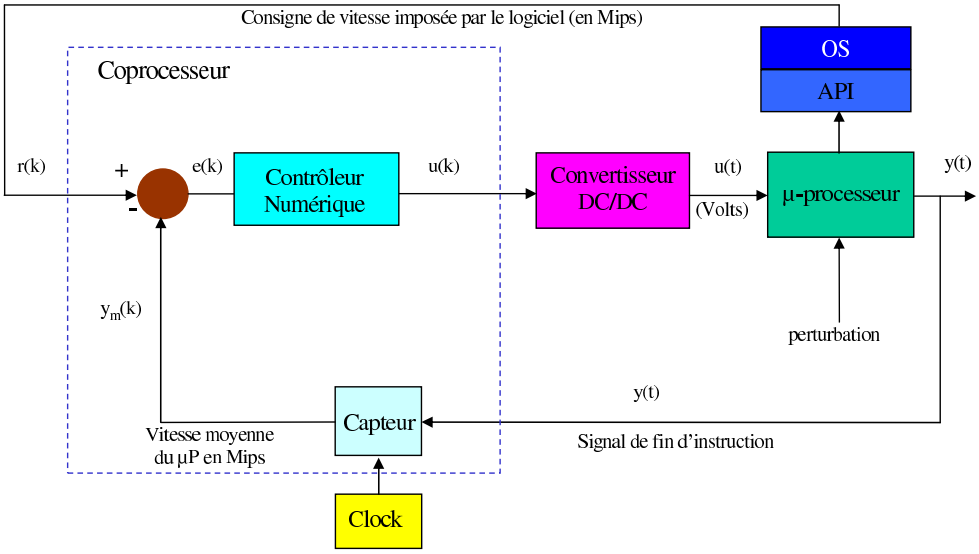


FIG. 5.5 – Schéma bloc du coprocesseur DVS.

Ce système, particulièrement intéressant pour les circuits asynchrones, peut aussi s'appliquer à des circuits synchrones dans la mesure où les instructions peuvent prendre un temps variable pour s'exécuter sur des processeurs complexes (défauts de cache, instructions multi-cycles, ...).

5.3.3 Mesure de la vitesse

Chaque fois que le processeur finit d'exécuter une instruction, un signal est envoyé au capteur qui le comptabilise. Puis, à partir de l'horloge externe qui fournit une base de temps, il est possible de connaître la vitesse réelle du processeur, moyennée sur la période de cette horloge. Il est alors nécessaire de réaliser une division. Pour réduire la complexité d'un tel traitement, il est souhaitable de considérer des périodes d'horloge qui sont des puissances de 2. On peut, par exemple, considérer une horloge à 31,25 kHz, donc de période 32 μs. Ainsi, il suffit de convertir la vitesse en Mips définie par le système d'exploitation en

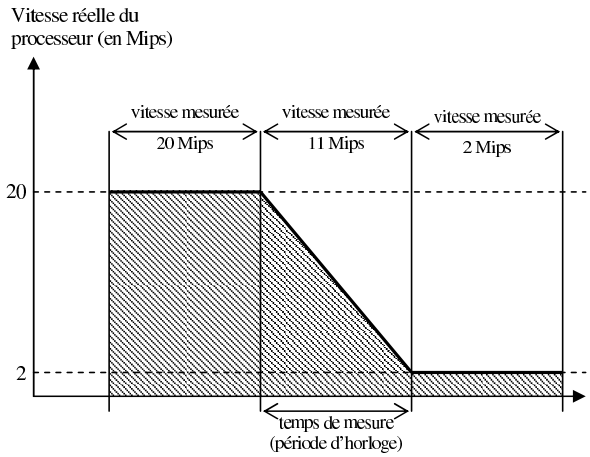


FIG. 5.6 – Vitesse instantanée et vitesse moyenne calculée.

nombre d'instructions à exécuter pendant cette période de 32 μs. Ceci peut même être effectué avec un décalage de 5 bits par le processeur lui-même. Ainsi, le capteur ne se résume plus qu'à un point mémoire qui conserve le résultat en sortie entre deux mesures et un compteur qui intègre les instructions exécutées, et qui se remet à zéro à chaque top d'horloge de 31,25 kHz.

Il est important de choisir la fréquence d'horloge de la base de temps avec précaution. En effet, dans la mesure où l'on souhaite réaliser un système faible consommation, il est souhaitable de choisir une fréquence d'horloge assez basse, surtout si cette horloge est fournie par un oscillateur externe. Dans ce cas, un oscillateur interne paraît difficielent utilisable, car trop imprécis. De plus, si on choisit une fréquence trop élevée, on risque d'avoir une précision sur la mesure de la vitesse du processeur assez mauvaise. Par exemple, si on choisit une fréquence d'horloge de 1 MHz, et que le logiciel souhaite que le processeur fonctionne aux alentours de 2 Mips, on va alors mesurer la vitesse du processeur avec en moyenne 2 échantillons. C'est à dire que le moindre retard sur une instruction peut provoquer une erreur d'un échantillon sur le décompte des instructions exécutées, soit une erreur de 50 % sur la mesure de la vitesse.

En revanche, choisir une vitesse trop faible réduit la réactivité du système. En effet, si on prend une fréquence de 31,25 kHz, soit une période d'environ 32 μs, et que l'on souhaite que le processeur fonctionne à 20 Mips, soit un temps moyen entre deux instructions de 50 ns, la vitesse est réévaluée toutes les 640 instructions. Bien qu'élevée, cette valeur pourrait éventuellement convenir. Cependant, il faut aussi tenir compte du comportement intégrateur de la régulation qui peut avoir besoin de plusieurs échantillons avant de réagir (voir section 5.3.4). De plus, la mesure effectuée par le capteur est une moyenne sur la période d'horloge. Ainsi, si on considère l'exemple de la figure 5.6 où il y a une variation linéaire de la tension et de la vitesse instantanée du processeur de 20 Mips à 2 Mips entre deux tops d'horloge, en supposant que les instructions exécutées sont toujours les mêmes, la vitesse mesurée en sortie du capteur sera de $\frac{20+2}{2} = 11$ Mips, alors que la vitesse instantanée du processeur est de 2 Mips. Cette valeur ne sera prise en compte qu'à la prochaine mesure, ce qui réduit encore la réactivité du système.

L'idéal serait alors d'avoir une faible fréquence quand le processeur fonctionne à faible vitesse et d'augmenter la fréquence d'horloge du capteur avec la vitesse du processeur. Ceci est une solution envisagée dans une étude d'automatique réalisée sur ce système dans le cadre d'une thèse qui vient de débuter, mais ne sera pas considérée ici.

5.3.4 Contrôle du régulateur DC/DC

Le contrôleur numérique doit ajuster la tension du régulateur en fonction de l'erreur entre la consigne définie par le logiciel et la vitesse mesurée. Il est possible dans un premier temps de considérer un contrôle uniquement proportionnel. Dans ce cas, la commande pour contrôler le DC/DC est proportionnelle à l'erreur entre la consigne et la mesure. La variation de la tension d'alimentation étant très rapide par rapport à la fréquence des mesures de la vitesse du processeur, ce type de contrôle provoque des oscillations de la tension d'alimentation, ce qui n'est pas souhaitable du point de vue de la consommation.

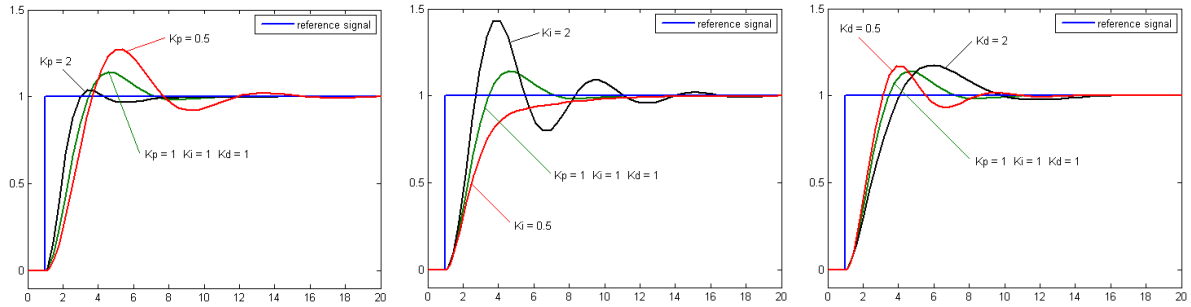
Il est alors possible d'ajouter un facteur dérivateur pour réduire le phénomène d'oscillation. En effet, en calculant la commande du contrôle du DC/DC en fonction de la dérivée de l'erreur dans le temps, il est possible de freiner ces oscillations, de telle sorte que le niveau de tension d'alimentation du processeur mette plus de temps à s'établir vers le niveau requis. Le contrôle dérivateur ralentit donc la vitesse de changement du contrôleur.

On peut alors ajouter un terme intégral au contrôle. Celui-ci prend en compte l'erreur accumulée du système. Il permet alors d'accélérer la vitesse d'établissement de la sortie. Cependant, si le facteur intégral est trop important, il se produit des phénomènes de dépassement ("overshoot") qui peuvent s'avérer problématiques.

Si on ajoute toutes les contributions proportionnelle, intégrale et dérivée, on obtient un contrôle PID. Si on appelle $e(t)$ l'erreur commise, la sortie de ce contrôleur a alors la forme :

$$S(t) = K_p.e(t) + K_i.\int_0^t e(\tau)d\tau + K_d.\frac{de(t)}{dt}$$

La figure 5.7 montre l'influence des paramètres K_p, K_i et K_d sur la sortie de contrôleur.



(a) paramètre proportionnel K_p (b) paramètre intégral K_i (c) paramètre dérivé K_d

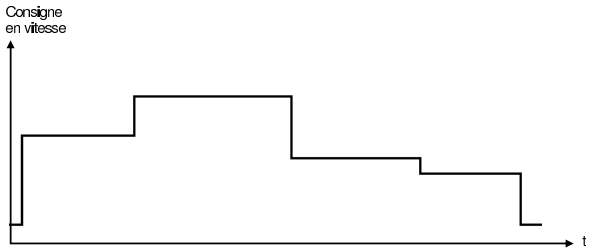
FIG. 5.7 – Influence des paramètres K_p, K_i et K_d sur la sortie d'un contrôle PID.

Définir ces paramètres est une tâche compliquée lorsque les blocs ne sont pas linéraires, comme c'est le cas ici. Une étude approfondie est alors nécessaire pour modéliser précisément le comportement des différents blocs d'automatique qui composent ce système. Une telle étude n'a pas été réalisée pour le moment. Les paramètres PID utilisés ici ont donc été réglés "à la main" en essayant d'obtenir un résultat satisfaisant.

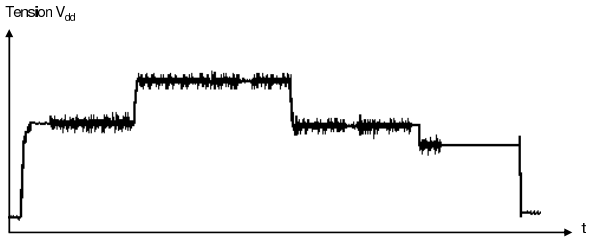
5.3.5 Exemple de régulation

Afin de montrer la faisabilité de la régulation, le système a été modélisé et simulé. Ainsi, le capteur et le contrôleur numérique ont été implantés en VHDL-AMS. Le processeur a été modélisé également en VHDL-AMS fonctionnel à partir des caractéristiques du processeur asynchrone Lutonium dont les caractéristiques correspondent aux modèles présentés précédemment équations 5.1 et 5.2. Les régulateurs ont été décrits en netlist de transistors ou en VHDL-AMS fonctionnel.

La figure 5.8 montre un exemple de régulation de la tension du processeur avec un régulateur à sortie continue, pour un profil logiciel donné. La consigne générée par le logiciel est donnée figure 5.8(a), la valeur de la tension d'alimentation en sortie du DC/DC est donnée figure 5.8(b), et la vitesse mesurée figure 5.8(c).



(a) Consigne logicielle en entrée du coprocesseur



(b) Tension d'alimentation aux bornes du processeur

(c) Vitesse mesurée par le capteur

FIG. 5.8 – Exemple de régulation avec un régulateur continu.

Comme on peut le constater, la vitesse du processeur mesurée est très proche de la consigne. De plus, la tension d'alimentation possède une forme assez similaire à cette consigne. On peut toutefois constater de petites oscillations de la vitesse mesurée autour de celle-ci. Ceci est dû au fait que les paramètres PID ont été définis manuellement. Une étude complète d'automatique pourrait diminuer ou supprimer ce phénomène.

5.3.6 Choix du régulateur DC/DC

Dans ce système, le choix du type de convertisseur DC/DC a une grande importance. En effet, si le convertisseur possède peu de niveaux de sortie, la tension d'alimentation sera en moyenne assez supérieure à la tension idéale. Au contraire, plus le niveau de sortie peut être proche de ce qu'il faudrait pour que la vitesse réelle du processeur soit la même que la consigne, et plus l'énergie consommée sera faible. Dans [64], nous avons réalisé une étude pour évaluer l'impact du type de régulateur utilisé. Dans cette étude, nous avons considéré deux catégories de régulateurs : les régulateurs à sortie continue, et les régulateurs à paliers, commandés par des commandes binaires.

La figure 5.9 montre des courbes de consommation du processeur, pour un profil applicatif identique donné :

- sans régulation de tension, à vitesse maximale (a)
- régulé avec un régulateur commandé sur 2 bits, (4 niveaux de tension)
- régulé avec un régulateur 2 bits (4 niveaux de tension) et une politique logicielle de gestion de la vitesse idéale qui prend en compte le surplus de vitesse pour ralentir le plus tôt possible le processeur (c)
- régulé avec un régulateur 4 bits (16 niveaux de tension) (d)
- régulé avec un régulateur à sortie continue (e)

Ces courbes ne considèrent que l'énergie consommée par le processeur et ne tiennent pas compte de l'énergie dissipée dans le régulateur et dans le coprocesseur. Il faut donc en plus considérer le rendement des régulateurs qui dépend d'un grand nombre de paramètres tels que la plage de tension, l'intensité débitée, la stabilité de l'alimentation, ...

Si le processeur sans régulation consomme à vitesse maximum 33 μJ pour le profil d'application donné sur la période considérée, on peut voir que l'utilisation d'un régulateur à paliers permet de faire chuter la consommation d'énergie à 15,5 μJ, soit 53 % d'économie.

On peut également remarquer que plus il y a de niveaux en sortie, plus il est possible de réaliser des économies d'énergie. En particulier, passer de 4 à 16 niveaux permet de faire passer la consommation d'énergie de 15,5 μJ à 10,9 μJ et d'économiser ainsi près de 30% d'énergie supplémentaire pour ce profil applicatif donné. Le gain est encore meilleur (10 μJ) si on utilise un régulateur à sortie continue. Cependant, la différence avec un régulateur à 16 niveaux est faible, et le choix devra donc se faire principalement sur le rendement des régulateurs pour les profils de courant considérés.

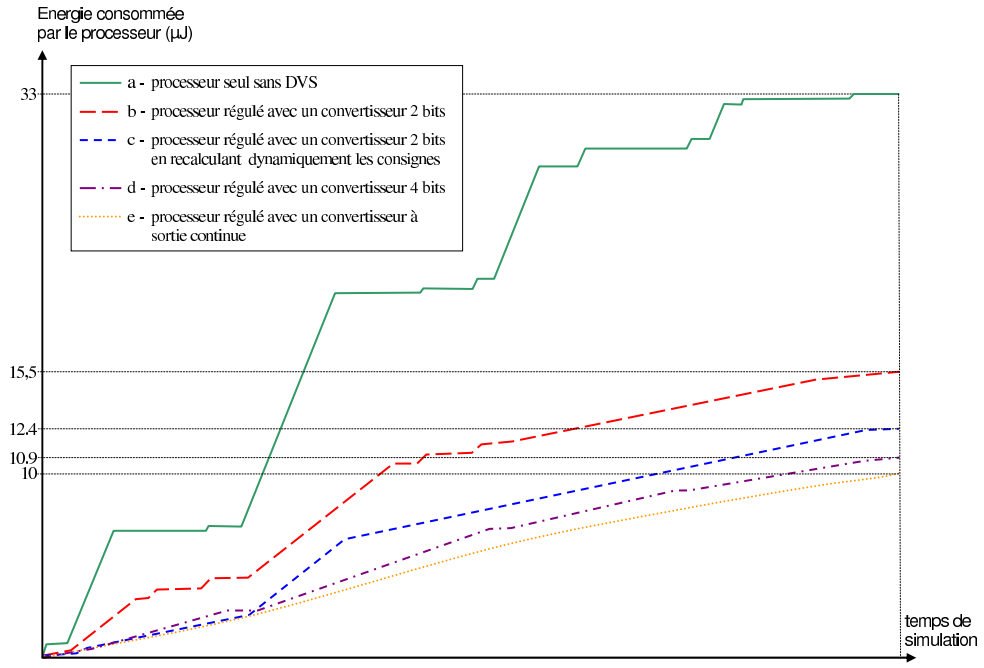


FIG. 5.9 – Énergie consommée par la processeur avec différentes régulations, pour un profil d'application donné.

Enfin, on peut voir un gain de 25% entre la consommation de la courbe (b) et celle de la courbe (c). Ce gain est uniquement dû à la politique de gestion de la consommation mise en œuvre au niveau logiciel. La politique logicielle de gestion des tensions est donc un aspect très important dans la régulation qui sera traité dans les chapitres 6 et 7.

5.4 Conclusion

L'utilisation de circuits asynchrones permet de réduire naturellement l'activité électrique au sein du circuit. Ainsi, les techniques de réduction de l'activité des circuits synchrones n'ont pas lieu d'être, ce qui permet d'éviter une gestion logicielle et une mise en œuvre matérielle potentiellement compliquée.

De plus, les techniques d'adaptation des tensions du circuit permettent de réduire considérablement l'énergie consommée statiquement et dynamiquement. En effet, la technique de l'ABB permet de polariser le substrat des transistors afin de réduire fortement le courant de fuite. La technique du DVS permet, quant à elle, de jouer sur la tension d'alimentation du circuit pour modifier la vitesse des circuits. Ainsi, beaucoup d'énergie peut être économisée si on a le temps d'effectuer un traitement. Les circuits asynchrones n'ayant pas d'horloge, il n'y a pas de fréquence d'horloge à changer conjointement à la tension. Le contrôle de cette technique s'en trouve donc facilité.

Néanmoins, l'absence d'horloge dans un processeur asynchrone empêche de connaître simplement la vitesse du circuit au niveau logiciel. Pour éviter de caractériser les temps d'exécution pire cas en fonction des tensions d'alimentation, nous avons spécifié un coprocesseur. Il permet d'asservir la vitesse

du processeur en fonction d'une consigne définie par le logiciel. Dans ce cas, la vitesse du processeur est mesurée ce qui permet de conserver un comportement en moyenne et, par conséquent, d'économiser de l'énergie.

Chapitre 6

Ordonnancement et gestion du DVS

6.1 Problématique dans un contexte de réseau de capteurs sans fil

Pour contrôler le coprocesseur DVS, il est nécessaire de générer une consigne logicielle. Cette consigne en vitesse doit être calculée en fonction des tâches qui s'exécutent sur le système, c'est-à-dire en fonction de leur complexité en termes de temps d'exécution ou de nombre d'instructions à exécuter, ainsi qu'en termes d'échéances. Les notions de contraintes temps réel et d'ordonnancement de tâches interviennent donc naturellement pour contrôler la vitesse du processeur.

Dans un réseau de capteurs sans fil, les tâches ont une faible complexité et il est donc très important de prendre cet aspect en considération pour la programmation et l'ordonnancement de celles-ci. Comme nous l'avons vu à la section 3.2.1, l'utilisation de système d'exploitation dédié au réseaux de capteurs permet d'économiser beaucoup d'énergie par rapport aux OS généralistes.

TinyOS est un OS pour réseaux de capteurs très faible consommation et avec une très faible empreinte mémoire. De plus, dans la version 2.x, il est possible de redéfinir la notion de tâche pour y ajouter les contraintes temps réel et de créer un ordonnanceur permettant de gérer ces contraintes. Nous avons choisi TinyOS 2.0 comme base pour développer et comparer les algorithmes d'ordonnancement temps réel et de DVS, en raison principalement de sa légèreté.

Au niveau algorithmique, la gestion de l'énergie se fait à plusieurs niveaux. Il y a tout d'abord le type d'ordonnancement de tâches, préemptif ou non-préemptif. Les ordonnancements non préemptifs sont très peu couteux à la fois en ressources mémoire et en ressources processeur. De plus, ils sont supportés par TinyOS. Quant aux ordonnancements préemptifs, bien que plus coûteux, ils permettent un contrôle plus précis des tâches et de la vitesse. Ils permettent donc généralement d'économiser plus d'énergie. Néanmoins, il faut faire attention à ce que le surcoût qu'engendre l'implantation d'un ordonnancement préemptif ne coûte pas plus cher en énergie que les optimisations en vitesse que permettent les préemptions, surtout dans un contexte de réseaux de capteurs sans fil où les tâches sont généralement

petites comparativement aux tâches rencontrées habituellement dans les systèmes temps réel telles que du décodage MPEG.

Il y a de plus le type de tâches : périodique ou sporadique. Les tâches périodiques ont des dates d'occurrence connues ce qui autorise généralement des politiques plus agressives. D'un autre côté, dans un contexte de réseau de capteurs sans fil, certaines tâches peuvent être considérées comme sporadiques, comme par exemple, une tâche créée pour recevoir un message provenant de la radio suite à une interruption. Considérer ce type de tâches peut s'avérer assez coûteux dans certains cas s'il est mal géré.

Enfin, il existe différents algorithmes permettant de contrôler la vitesse du processeur en fonction des tâches présentes sur le système et de leurs contraintes temps réel. Ces algorithmes ont une grande influence sur la consommation d'énergie.

Sans vouloir se montrer exhaustive, cette partie a pour but de mettre en évidence les mécanismes de gestion du temps réel et du DVS qui ont un impact, qu'il soit positif ou négatif, sur la consommation d'énergie dans un contexte de réseau de capteurs sans fil.

6.2 Ordonnancement de tâches

Il existe principalement deux types d'ordonnancement utilisés dans les systèmes temps réel : les ordonnancements "à priorité dynamique" tels que l'EDF (Earliest Deadline First) et les ordonnancements "à priorité constante" comme l'algorithme RM (Rate-Monotonic).

6.2.1 Définitions

Nous supposons dans cette section un ordonnancement de tâches indépendantes, sur un système uniprocesseur.

Définition 6.1 *Un ensemble de tâches est dit **faisable** s'il existe un ordonnancement de ces tâches qui respecte les contraintes temporelles associées à ces tâches.*

Définition 6.2 *Un ordonnanceur est dit **optimal** s'il peut produire un ordonnancement pour tout ensemble faisable de tâches.*

Définition 6.3 ***Test d'ordonnançabilité** : un test d'ordonnançabilité est utilisé pour valider qu'une application donnée peut satisfaire ses contraintes temporelles quand elle est ordonnancée par un algorithme d'ordonnancement donné.*

Ce test d'ordonnancement est souvent réalisé à la compilation, avant que le système et ses tâches ne démarrent. Si ce test peut être effectué efficacement, alors il peut aussi être réalisé à l'exécution comme un test en ligne.

Définition 6.4 *On dit qu'un système de tâches est à* ***échéance sur requête*** *si les périodes des tâches sont égales à leurs échéances relatives.*

6.2.2 Notations

Soit T_i la $i^{ième}$ tâche du système. On appellera N_i le nombre d'instructions à exécuter dans le pire cas de la tâche T_i, t_i sa date de création et t_{d_i} son échéance absolue. On peut alors définir son échéance relative $\Delta_i = t_{d_i} - t_i$.

Appelons $S_i = \frac{N_i}{\Delta_i}$ la vitesse intrinsèque de la tâche T_i. Cette vitesse correspond à la vitesse moyenne minimale que le processeur doit avoir pour exécuter la tâche T_i, supposée seule sur le système, pour respecter son échéance. Cette vitesse est à mettre en relation avec le taux utilisation U_i de la tâche, utilisé généralement dans les systèmes temps réel. En effet, si on appelle W_i le temps d'exécution pire cas de la tâche, on a $U_i = \frac{W_i}{\Delta_i}$. Si on appelle $S_{proc_{max}}$ la vitesse maximale du processeur, puisque $W_i = \frac{N_i}{S_{proc_{max}}}$, on a $S_i = U_i \times S_{proc_{max}}$. Les notions de vitesse et de taux d'utilisation sont donc équivalentes.

Généralement exprimé en nombre de cycles, le temps d'exécution n'est pas facile à connaître avec un processeur asynchrone. De plus, dans l'approche utilisant le taux d'utilisation, il est nécessaire de faire des considérations dans le pire cas. C'est pourquoi il est préférable d'utiliser la notion de vitesse et de nombre d'instructions permettant d'avoir un comportement en moyenne grâce à l'utilisation du coprocesseur du chapitre 5, plutôt que celle de taux d'utilisation.

On appellera de plus n_i le nombre d'instructions exécutées par la tâche T_i depuis son occurrence (donc $0 \leq n_i \leq N_i$), et $n_{restant_i} = N_i - n_i$ le nombre d'instructions de T_i qu'il reste à exécuter dans le pire des cas. De la même manière on appellera w_i le temps passé à exécuter la tâche T_i et $w_{restant_i} = W_i - w_i$. Pour une tâche périodique T_i, on notera P_i sa période.

Enfin, on appellera c l'indice de la tâche qui s'exécute actuellement sur le processeur.

6.2.3 Ordonnancement Rate-monotonic

L'algorithme rate-monotonic (RM) [45] est un algorithme d'ordonnancement temps réel en ligne à priorité constante. Il attribue la priorité la plus forte à la tâche qui possède la plus petite période. Il n'est donc utilisable qu'avec des tâches périodiques. RM est optimal dans le cadre d'un système de tâches indépendantes et à échéance sur requête, avec un ordonnanceur préemptif. De ce fait, il n'est généralement utilisé que pour ordonnancer des tâches vérifiant ces propriétés.

Test 6.1 *Soit $\{T_1, \ldots, T_n\}$, un ensemble de n tâches indépendantes, préemptibles et périodiques, à échéance sur requête sur un système uniprocesseur. Une condition nécessaire pour un ordonnancement*

faisable de cet ensemble de tâche par l'ordonnancement RM est :

$$U = \sum_{i=1}^{n} \frac{W_i}{P_i} < 1 \qquad ou\ encore \qquad S_{proc} = \sum_{i=1}^{n} \frac{N_i}{P_i} < S_{proc_{max}}$$

et une condition suffisante est :

$$U = \sum_{i=1}^{n} \frac{W_i}{P_i} < n \times (2^{\frac{1}{n}} - 1) \qquad ou\ encore \qquad S_{proc} = \sum_{i=1}^{n} \frac{N_i}{P_i} < S_{proc_{max}} \times n \times (2^{\frac{1}{n}} - 1)$$

6.2.4 Algorithme EDF

L'algorithme earliest deadline first (EDF) [45] est un algorithme d'ordonnancement préemptif temps réel à priorité dynamique. Il attribue une priorité à chaque tâche en fonction de l'échéance de cette dernière. Plus l'échéance absolue t_{d_i} d'une tâche T_i est proche, plus sa priorité est grande. Cet algorithme est optimal pour tous les types de systèmes de tâches. En particulier, il peut aussi bien ordonnancer des tâches périodiques que des tâches sporadiques.

Test 6.2 *Soient n tâches indépendantes et préemptibles sur un système uniprocesseur. Une condition suffisante pour un ordonnancement faisable est :*

$$U = \sum_{i=1}^{n} \frac{W_i}{\Delta_i} < 1 \qquad ou\ encore \qquad S_{proc} = \sum_{i=1}^{n} \frac{N_i}{\Delta_i} < S_{proc_{max}}$$

Le test d'ordonnançabilité 6.2 est valable à la fois pour des tâches sporadiques et des tâches périodiques ordonnancées par l'algorithme EDF. Dans le dernier cas, la seule restriction est que les échéances des tâches périodiques doivent être plus petites que leurs périodes respectives, c'est-à-dire que $\Delta_i \leq P_i, \forall i \in [1; n]$.

6.2.5 Algorithme EDF non préemptif

Du fait qu'il puisse être inclus dans le calcul du temps d'exécution pire cas, le surcoût engendré par l'OS, et par l'ordonnanceur en particulier, est souvent négligé. Pourtant, dans le contexte des réseaux de capteurs sans fil, les algorithmes d'ordonnancement peuvent avoir un impact important aussi bien sur la charge du processeur que sur la quantité de mémoire requise, notamment s'il y a des préemptions. C'est pourquoi il est intéressant de considérer les ordonnancements non préemptifs, généralement supportés par les systèmes d'exploitation dédiés aux réseaux de capteurs sans fil comme TinyOS. Nous avons pour cela réalisé une version non préemptive de l'ordonnancement EDF, que l'on nommera NP-EDF.

Puisqu'il s'agit d'un algorithme de type EDF, il faut garantir que la tâche en cours d'exécution possède la plus petite échéance absolue. Ainsi, si une tâche T_i est créée avec une échéance absolue plus

petite que celle de la tâche T_c en train de s'exécuter, puisqu'il n'y a pas de préemption, il faut rendre la tâche T_c aussi "prioritaire" que T_i en modifiant l'échéance absolue t_{d_c} de telle sorte que l'on ait $t_{d_c} = t_{d_i}$.

Le changement des contraintes temps réel pose le problème de l'ordonnançabilité d'un ensemble de tâches par l'ordonnanceur NP-EDF. En effet, le fait de changer l'échéance de la tâche T_c en cours d'exécution change par la même occasion le taux d'utilisation de cette tâche sur le processeur. Cette augmentation dépend du temps d'exécution restant dans le pire cas $w_{restant_c}$ de la tâche T_c et de l'échéance relative de la tâche qui vient d'être créée Δ_i. L'utilisation de la tâche courante passe donc de $\frac{W_c}{\Delta_c}$ à $\frac{w_{restant_c}}{\Delta_i}$. Ainsi, pour garantir qu'un ensemble de tâches $(T_k)_{k=1}^n$ puisse être ordonnancé par l'algorithme NP-EDF, il suffit de vérifier que l'utilisation totale du processeur soit toujours inférieure à 1, ce qui peut se traduire par le test 6.3.

Test 6.3 *Soient $(T_k)_{k=1}^n$ n tâches indépendantes et non préemptibles sur un système uniprocesseur. Une condition suffisante pour un ordonnancement faisable est :*

$$\forall p, q \in [1, n], \sum_{\substack{k=1,\\ k\neq p}}^{n} \frac{W_k}{\Delta_k} + \frac{W_p}{\Delta_q} < 1 \qquad ou\ encore \qquad \forall p, q \in [1, n], \sum_{\substack{k=1,\\ k\neq p}}^{n} \frac{N_k}{\Delta_k} + \frac{N_p}{\Delta_q} < S_{proc_{max}}$$

Ce test peut s'avérer assez coûteux à calculer en ligne. Il pourra donc être avantageusement effectué statiquement hors ligne si l'on connait les caractéristiques de toutes les tâches qui sont amenées à s'exécuter sur le système.

6.3 Modèle de tâche

6.3.1 Tâches sporadiques

Une tâche sporadique, ou apériodique, est par définition une tâche qui n'est pas périodique, c'est-à-dire que ses occurrences peuvent arriver à n'importe quel moment, ou bien même ne pas arriver du tout. La gestion du temps est généralement décorrélée de la notion de tâche. Elle peut être laissée aux soins des tâches elles-mêmes qui peuvent définir, via une interface permettant de masquer la gestion du timer, des traitements pouvant être ponctuels ou périodiques. Ces traitements peuvent s'exécuter dans un contexte d'interruption s'ils sont de très faible complexité, ou plus généralement au sein d'une tâche.

De plus, les occurrences de tâches ne sont pas forcément dépendantes du timer. Une tâche peut ainsi être créée par une interruption provenant de la radio par exemple, ou par une autre tâche. On peut d'ailleurs considérer que le comportement de ce type de tâche est évènementiel, c'est-à-dire que la création des tâches sporadiques fait suite à un évènement.

6.3.2 Tâches périodiques

Un grand nombre d'algorithmes de gestion du DVS supposent des tâches périodiques. Comme les dates d'occurrence des tâches sont prévues, il est possible de savoir quelle sera la charge future du processeur et par conséquent quelle tension appliquer pour satisfaire toutes les futures échéances.

De plus, les dates d'arrivée des tâches périodiques sont régies par des interruptions timer. Ainsi, il est intéressant de pouvoir gérer le temps au sein même de l'ordonnanceur pour centraliser sa gestion et s'affranchir de la couche de virtualisation qu'il est commode d'avoir pour gérer le temps avec des tâches sporadiques. Cette gestion centralisée est alors plus optimisée et permet de réduire le coût du traitement des interruptions provenant du timer, et donc de réduire la consommation d'énergie.

Cependant, il est parfois nécessaire d'avoir des tâches sporadiques dans un système. De plus, le comportement évènementiel des tâches sporadiques permet généralement d'optimiser les traitements dans le système. Celles-ci doivent donc pouvoir être gérées dans l'ordonnanceur de tâches périodiques. Ainsi, une tâche sporadique pourra facilement être ordonnancée en la considérant comme une tâche périodique dont la période est le temps minimum séparant chaque occurrence de cette tâche ; l'échéance et le temps d'exécution restent identiques.

Cette transformation est valable pour tout ordonnanceur de tâches périodiques, incluant les ordonnancements à priorités fixes tels que l'ordonnancement rate-monotonic. Cependant, si on se place dans le cadre plus restreint de l'EDF qui gère les tâches sporadiques, cette transformation est inutile. En revanche, il est important de considérer la gestion du temps, pour ne pas perdre l'avantage que permettent les tâches périodiques, et laisser ainsi l'ordonnanceur gérer le temps et les interruptions timer pour toutes les tâches.

Enfin, si une tâche sporadique est périodisée, il est important de pouvoir l'arrêter lorsque l'on sait que celle-ci n'a pas à s'exécuter à la prochaine période. Ceci permet donc d'éviter d'avoir des interruptions timer inutiles, et économiser de l'énergie.

6.4 Algorithme de gestion du DVS

Nous ne nous intéresserons dans la suite de ce chapitre qu'aux algorithmes EDF et NP-EDF qui possèdent une bonne ordonnançabilité tout en permettant des algorithmes de gestion d'énergie plus simples et plus efficaces. Ils permettent de plus de montrer l'impact des préemptions et du modèle de tâches sur la consommation.

6.4.1 Gestion statique de la vitesse

Les tests d'ordonnançabilité 6.2 et 6.3 fournissent des conditions suffisantes pour ordonnancer un ensemble de tâches selon un ordonnancement EDF ou NP-EDF, et fournissent en même temps un moyen

simple de gérer statiquement la vitesse. En effet, ces tests vérifient que le taux d'utilisation du processeur U ne pourra jamais excéder 1 pour un ensemble donné de tâches. C'est-à-dire que, dans le pire cas d'exécution, si on diminue la vitesse d'exécution du processeur, et qu'on augmente le temps d'exécution de toutes les tâches du système pour arriver à un temps compris entre W_i et $\frac{W_i}{U}$, le taux d'utilisation sera alors entre U et 1 et l'ensemble des tâches est toujours ordonnançable. On peut donc diminuer statiquement la vitesse du processeur, de $S_{proc_{max}}$ à $U \times S_{proc_{max}}$, sans violer les contraintes temps réel. D'ailleurs, la vitesse $U \times S_{proc_{max}}$ correspond exactement à la vitesse S_{proc} définie dans ces tests à l'aide des notions de vitesse. Cette vitesse est la vitesse minimale qui garantit le respect des échéances de toutes les tâches dans le pire des cas, quelles que soient leurs dates d'occurrence.

Cette gestion statique permet donc de diminuer la vitesse du processeur pour réduire la consommation d'énergie sans ajouter le moindre surcoût à l'exécution puisque la vitesse peut être calculée hors ligne. Néanmoins, dans la mesure où il se place dans des conditions pire cas, à la fois en termes de date de création de tâches, mais aussi en termes de temps d'exécution, on peut penser que cette solution peut manquer d'efficacité, comparativement à une gestion dynamique de la vitesse.

6.4.2 Gestion simple de la vitesse

Gérer statiquement la vitesse du processeur nécessite de considérer des pires cas qui peuvent ne jamais arriver et même se trouver très éloignés du cas moyen. Il peut donc être avantageux de considérer des algorithmes dynamiques qui appliquent un couple tension/vitesse au processeur en fonction des caractéristiques des tâches présentes dans le système à chaque instant.

Considérons dans un premier temps des tâches sporadiques ordonnancées par les algorithmes EDF. L'algorithme de gestion de la vitesse est donné algorithme 6.1. À chaque création d'une tâche T_i, on peut calculer la vitesse intrinsèque S_i de cette tâche à partir de l'échéance relative Δ_i et du nombre d'instructions à exécuter dans le pire cas, N_i. Cette vitesse est alors ajoutée à la vitesse courante du processeur pour définir la nouvelle vitesse du processeur et sera retranchée une fois l'échéance absolue passée. Le test 6.2 garantit que la vitesse maximale du processeur ne sera pas atteinte.

Considérons l'exemple de trois tâches montré figure 6.1. On peut remarquer que les surfaces correspondent à des nombres d'instructions. En additionnant les courbes de vitesse en fonction du temps de toutes les tâches pour obtenir la vitesse du processeur, on obtient bien un nombre d'instructions exécutées total, égal à la somme de toutes les instructions à exécuter.

Le principe est assez similaire avec un ordonnancement NP-EDF. En effet, comme on peut le voir dans l'algorithme 6.2, la seule différence réside dans le fait qu'à chaque création de tâche T_i, il faut non seulement ajouter la vitesse intrinsèque S_i de la tâche qui vient d'être créée, mais il faut aussi modifier l'échéance t_{d_c} de la tâche qui est en train de s'exécuter sur le processeur si $t_{d_i} < t_{d_c}$. Cette modification de l'échéance conduit à faire passer la vitesse intrinsèque S_c de la tâche T_c de $\frac{N_c}{\Delta_c}$ à $\frac{n_{restant_c}}{\Delta_i}$. Le test 6.3 garantit que la vitesse maximale du processeur ne sera pas atteinte.

Algorithme 6.1 : Gestion simple de la vitesse avec un ordonnancement EDF.

`maj_vitesse()`
début
$Vitesse_{coprocesseur} \leftarrow \sum_{i=0}^{n} S_i$
fin

`occurrence_tâche(`T_i`)`
début
$S_i \leftarrow \frac{N_i}{\Delta_i}$
`maj_vitesse()`
fin

`échéance_passée(`T_i`)`
début
$S_i \leftarrow 0$
`maj_vitesse()`
fin

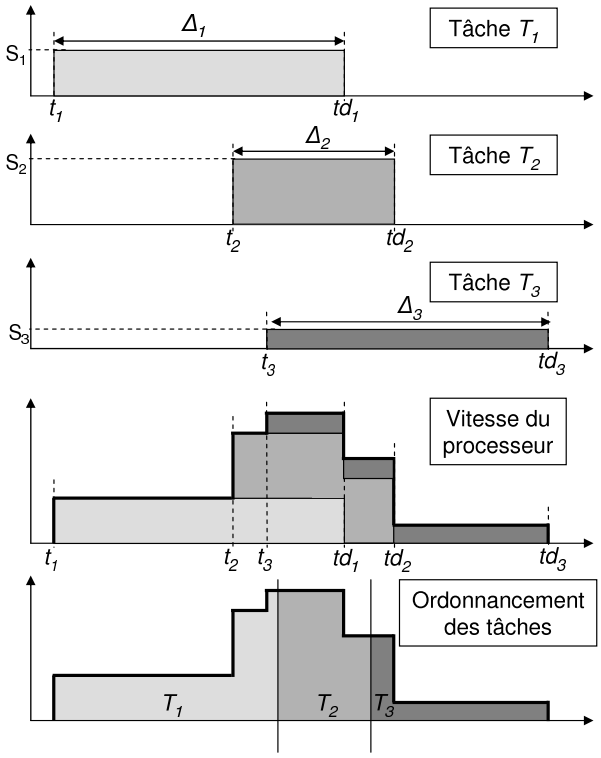


FIG. 6.1 – Principe de gestion simple de la vitesse avec ordonnancement EDF de tâches sporadiques.

Comparativement à l'EDF, comme on peut le voir figure 6.2, l'absence de préemption de l'algorithme NP-EDF cause une élévation de la vitesse due au fait que l'échéance de la tâche qui ne peut pas être préemptée est raccourcie. Cette augmentation de la vitesse entraîne une augmentation de la consommation.

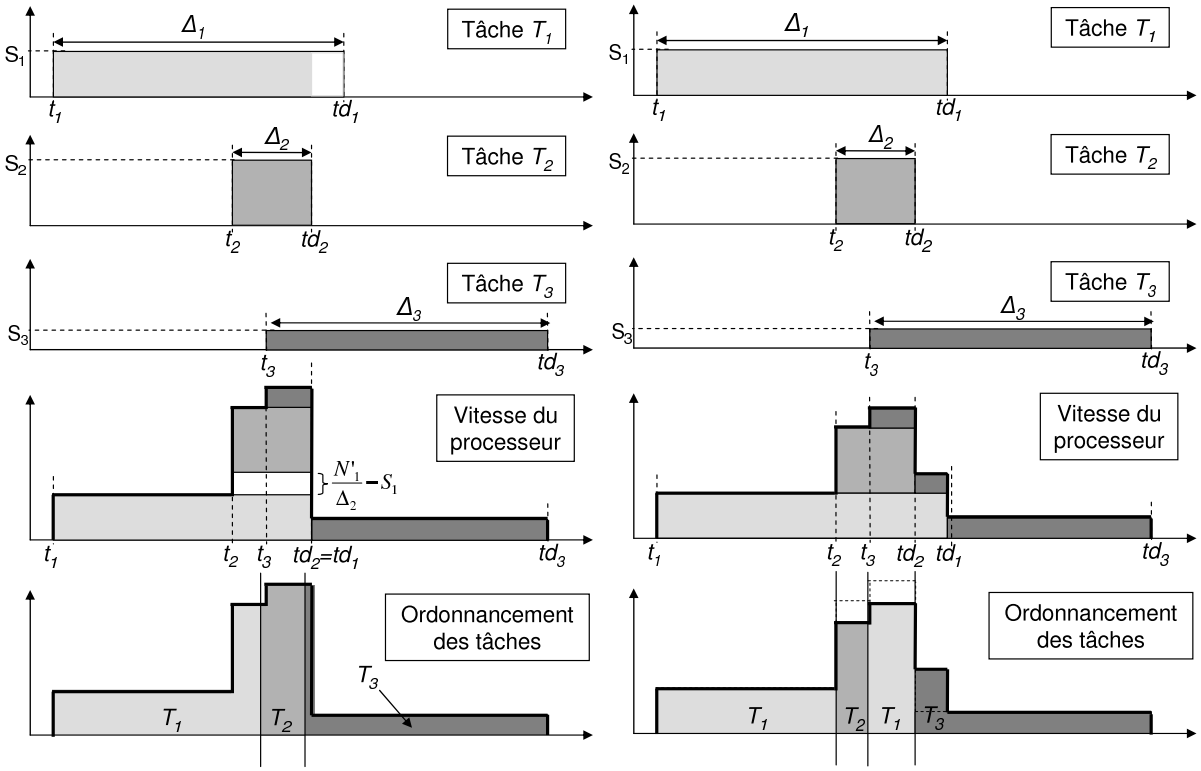
Enfin, dans la mesure où il est possible de les considérer comme des tâches sporadiques, les tâches périodiques peuvent être utilisées conjointement avec une telle politique de gestion de la vitesse.

Algorithme 6.2 : Gestion simple de la vitesse avec un ordonnancement NP-EDF.

`maj_vitesse()`
début
  $Vitesse_{coprocesseur} \leftarrow \sum_{i=0}^{n} S_i$
fin

`occurrence_tâche(`T_i`)`
début
  $S_i \leftarrow \frac{N_i}{\Delta_i}$
  si $(t_{d_i} < t_{d_c})$ **alors** $S_c \leftarrow \frac{n_c}{\Delta_i}$
  `maj_vitesse()`
fin

`échéance_passée(`T_i`)`
début
  $S_i \leftarrow 0$
  `maj_vitesse()`
fin



(a) ordonnancement EDF non-préemptif (b) ordonnancement EDF préemptif

FIG. 6.2 – Comparaison entre l'ordonnancement EDF non-préemptif (a) et préemptif (b).

6.4.3 Algorithme "Cycle-conserving RT-DVS"

Il est généralement rare que le pire cas d'exécution soit atteint. Les tâches temps réel utilisent donc généralement beaucoup moins le processeur que leur invocation pire cas. L'algorithme CC-RTDVS [57] propose de tirer profit de ceci pour réduire la vitesse du processeur. Cet algorithme suppose à la base l'utilisation des tâches périodiques et un ordonnancement préemptif EDF ou RM. Nous ne considèrerons ici qu'EDF pour les raisons déjà évoquées au début de ce chapitre 6.

Lorsqu'une tâche périodique est prête pour sa prochaine invocation, il n'est pas possible de connaître à l'avance le nombre d'instructions qui seront nécessaires pour exécuter cette tâche. Il faut donc supposer a priori qu'elle va utiliser ses contraintes pire cas. Lorsque cette tâche se termine, le nombre d'instructions exécutées est alors connu et comparé au nombre d'instructions pire cas. Toutes les instructions "non-utilisées" seraient alors gâchées et conduiraient le processeur à se mettre dans un mode d'attente prématurément, que ce soit juste après la fin de la tâche, ou plus généralement avant la prochaine occurrence de la tâche périodique.

Par conséquent, plutôt que de passer du temps à ne rien faire entre la fin de la tâche et sa prochaine occurrence, il est possible de réduire les vitesses des autres tâches pendant cet intervalle de temps, tout en garantissant qu'aucune contrainte ne soit violée.

L'algorithme 6.3 peut être décrit comme suit. Supposons qu'une tâche T_i se termine en ayant exécuté n_i instructions, qui est la plupart du temps éloigné de son pire cas N_i. On recalcule la vitesse intrinsèque de la tâche T_i en tenant compte du nombre d'instructions qui ont été effectivement exécutées, n_i, et on met à jour la vitesse de la tâche $S_i = \frac{n_i}{P_i}$. Puis on applique la vitesse du processeur à $S = \sum S_i$. À la prochaine occurrence de la tâche T_i, on recalcule la vitesse de la tâche $S_i = N_i/P_i$ en faisant donc la supposition que la prochaine exécution de la tâche T_i sera son exécution pire cas.

Algorithme 6.3 : Cycle-conserving RT-DVS avec des tâches périodiques.

`maj_vitesse()`
début
  $Vitesse_{coprocesseur} \leftarrow \sum_{i=0}^{n} S_i$
fin

`occurrence_tâche(`T_i`)`
début
  $S_i \leftarrow \frac{N_i}{P_i}$
  `maj_vitesse()`
fin

`fin_tâche(`T_i`)`
début
  $S_i \leftarrow \frac{n_i}{P_i}$
  `maj_vitesse()`
fin

On peut remarquer que cet algorithme de gestion de la vitesse est très similaire à celui qui a été exposé dans la partie précédente. La seule différence réside dans le fait qu'il permet de réduire la vitesse du processeur lorsque les tâches utilisent moins d'instructions que leurs invocations pire cas. D'ailleurs, cet algorithme pourrait être facilement adapté pour fonctionner avec des tâches sporadiques, comme on peut le voir sur l'algorithme 6.4. En effet, contrairement aux tâches périodiques, il n'y a pas automatiquement de nouvelle occurrence d'une tâche sporadique à chaque fois que son échéance est passée. Ainsi, pour supporter le modèle de tâche sporadique, il suffit de remettre à zéro la vitesse intrinsèque d'une tâche sporadique dont l'échéance vient de passer. De plus, il est possible d'utiliser cette gestion de la vitesse avec un ordonnancement NP-EDF. Il suffit de modifier la fonction `occurrence_tâche` pour modifier la vitesse de la tâche courante si nécessaire.

Algorithme 6.4 : Cycle-conserving RT-DVS avec des tâches sporadiques.

`maj_vitesse()`
début
  $Vitesse_{coprocesseur} \leftarrow \sum_{i=0}^{n} S_i$
fin

`occurrence_tâche(`T_i`)`
début
  $S_i \leftarrow \frac{N_i}{\Delta_i}$
  `maj_vitesse()`
fin

`fin_tâche(`T_i`)`
début
  $S_i \leftarrow \frac{n_i}{\Delta_i}$
  `maj_vitesse()`
fin

`échéance_passée(`T_i`)`
début
  $S_i \leftarrow 0$
  `maj_vitesse()`
fin

6.4.4 Algorithme "Look-ahead RT-DVS"

L'algorithme LA-RTDVS [57] suivant est réputé pour être encore plus efficace dans les systèmes temps réel traditionnels pour économiser de l'énergie. L'algorithme précédent permet d'exécuter les tâches à une vitesse élevée pour ensuite la diminuer lorsque ces dernières se terminent. À l'inverse, cet algorithme essaie de différer le travail à effectuer au maximum, et définit la vitesse du processeur au minimum pour garantir les échéances de toutes les tâches du système.

D'un côté ceci peut obliger le système à choisir une vitesse future très élevée, voir même la vitesse maximale. D'un autre côté, si les tâches prennent beaucoup moins d'instructions que leurs pires cas, le

futur pic de vitesse peut ne jamais avoir lieu et cette heuristique permet alors de travailler la plupart du temps à faible vitesse.

On peut donc noter que l'efficacité de cette heuristique dépend grandement du fait que les tâches se terminent bien avant leur pire cas.

Algorithme 6.5 : Look-ahead RT-DVS.

`occurrence_tâche(`T_i`)`
début
  $n_i \leftarrow N_i$
  `différer()`
fin

`fin_tâche(`T_i`)`
début
  $n_{restant_i} \leftarrow 0$
  `différer()`
fin

`pendant_execution_tâche(`T_i`)`
début
  décrémenter $n_{restant_i}$
fin

`différer()`
début
  $S \leftarrow S_{statique} = \sum_{i=0}^{n} S_i$
  $min_{instr} \leftarrow 0$
  pour *i=1* **à** *n*, $T_i \in \{T_0, \ldots, T_n / t_{d_0} > t_{d_1} > \cdots > t_{d_n}\}$ **faire**
    $S \leftarrow S - \frac{N_i}{P_i}$
    $x \leftarrow \max(0, n_{restant_i} - (S_{proc_{max}} - S) \times (t_{d_i} - t_{d_n}))$
    si $i \neq n$ **alors**
      $S \leftarrow S + \frac{n_i - x}{t_{d_i} - t_{d_n}}$
    $min_{instr} = min_{instr} + x$
  $Vitesse_{coprocesseur} \leftarrow \frac{min_{instr}}{t_{d_n} - t_{courant}}$
fin

Cette heuristique est montrée algorithme 6.5 et peut être décrite ainsi. Appelons $n_{restant_i}$ le nombre d'instructions restant à exécuter dans le pire cas. Cette variable est affectée à N_i à chaque nouvelle occurrence de la tâche, décrémentée lors de l'exécution de la tâche et mise à zéro lorsque la tâche est terminée. La partie principale de cet algorithme concerne la fonction `differer()`. Dans cette fonction, on regarde l'intervalle entre maintenant et la prochaine échéance parmi les échéances de toutes les tâches classées dans l'ordre de l'EDF inversé (i.e. $t_{d_n} \leq \ldots \leq t_{d_1}$), et on essaie de différer le maximum de travail après cette échéance t_{d_n}. On calcule alors le nombre minimal d'instructions min_{instr} qui doivent être exécutées avant la prochaine échéance t_{d_n} pour que toutes les échéances puissent être respectées.

La vitesse est ensuite définie pour que le processeur exécute les min_{instr} instructions avant cette plus proche échéance t_{d_n}.

Pour calculer min_{instr}, il faut considérer les tâches dans l'ordre inverse de l'EDF, c'est-à-dire en commençant par la tâche dont l'échéance est la plus éloignée. On suppose alors une exécution pire cas pour les tâches qui ont une échéance plus rapprochée dans le temps, c'est-à-dire plus prioritaires au sens de l'EDF. Puis on calcule le nombre minimum d'instructions, x, que la tâche T_i doit exécuter avant l'échéance la plus proche t_{d_n}, pour que l'échéance de T_i soit respectée. Une fois parcourues toutes les tâches dans l'ordre EDF inversé, la vitesse du processeur est définie.

On ne considère, dans cet algorithme, que les tâches qui sont amenées à s'exécuter, et leur échéance absolue. Hormis le fait que l'on interdise à une tâche d'arriver avant que son échéance précédente ne soit passée – ce qui est le cas par définition avec les tâches périodiques – on ne fait aucune supposition sur les prochaines occurrences des tâches. On peut donc utiliser cet algorithme de gestion des tensions aussi bien avec des tâches périodiques qu'avec des tâches sporadiques.

De même, si l'ensemble des tâches du système passe le test 6.3, il est possible d'ordonnancer les tâches à l'aide de l'algorithme NP-EDF.

6.5 Implantation et méthodologie de simulation

6.5.1 Simulateur

Pour évaluer précisément les différents algorithmes logiciels, nous avons porté TinyOS 2.0 vers une architecture MIPS puis simulé sur un simulateur de jeu d'instructions (ISS), VMIPS. Pour cela nous avons enrichi l'ISS d'un timer indépendant du processeur et d'un coprocesseur DVS. Ce dernier permet de changer la vitesse du processeur par rapport au temps de simulation, et de mesurer la consommation du processeur. Les modèles de tension, de vitesse et d'énergie utilisés sont les mêmes que ceux décrits section 5.1.3, équations 5.1 et 5.2. Ne ciblant pas de technologie en particulier, nous avons choisi dans l'équation 5.2 de prendre $\beta = 1$ pour simplifier, mais ce paramètre serait facilement modifiable si on considérait une technologie particulière.

Dans la réalité, le temps d'exécution, ainsi que l'énergie consommée peuvent dépendre des données traitées et de l'instruction considérée. Afin de simplifier le problème, on considère ici que le temps d'exécution et la vitesse ne dépendent que de la tension d'alimentation fournie par le coprocesseur DVS. Il ne s'agit pas d'une limitation dans la mesure où les applications et les algorithmes considérés sont très semblables globalement et où les données traitées sont généralement très similaires. Ainsi, pour l'énergie, on accumule le carré de la valeur de la tension à chaque instruction alors que la vitesse varie linéairement en fonction de la tension entre 1 Mips à 0,6V et 20 Mips à 1,2V.

Par ailleurs, un périphérique de traçage a été ajouté à l'ISS. Il permet de tracer l'exécution du code

pour établir le profil de l'application, c'est-à-dire le nombre d'instructions exécutées dans les différentes parties du code telles que l'initialisation, l'ordonnanceur, les traitants d'interruption ou l'application.

6.5.2 Implantation de l'EDF dans TinyOS

6.5.2.1 TinyOS et les préemptions

Comme nous l'avons vu précédemment, du fait qu'il n'y ait pas de préemption, l'ordonnancement NP-EDF décrit dans la section 6.2.5 peut être amené à changer les échéances des tâches en cours d'exécution, ce qui augmente la vitesse du processeur. Cette augmentation de vitesse introduit une consommation supplémentaire et oblige à surdimensionner le processeur. C'est pourquoi l'ordonnancement EDF préemptif a été considéré, bien que plus coûteux en mémoire et en utilisation processeur.

Le modèle d'exécution de TinyOS/nesC interdit normalement la préemption entre les tâches. Ceci permet par exemple d'avoir des vérifications statiques sur les possibles variables qui seraient partagées entre du code synchrone, c'est-à-dire les tâches nesC, et le code asynchrone, c'est-à-dire le code des traitants d'interruptions.

Passée cette limitation, rien dans le modèle de programmation par composant nesC n'empêche l'utilisation d'un ordonnancement préemptif, même si une partie de cet ordonnanceur doit être décrit en assembleur et en C. Bien que le fonctionnement du système résultant soit assez éloigné du modèle fonctionnel de TinyOS 2, ce dernier permet une comparaison assez précise des différents algorithmes de gestion de la consommation.

6.5.2.2 Implantation des préemptions

Les préemptions entre tâches ne pouvant s'effectuer que par des appels systèmes ou par des interruptions, il est possible de masquer les interruptions pour avoir des sections atomiques. Il est donc possible de garantir une cohérence des données dans le programme. Toutefois, le compilateur nesC n'est plus en mesure de détecter les accès concurrents à des variables comme il le fait normalement, et il est donc du ressort du programmeur de s'assujétir cet aspect de la programmation.

Dans un OS multi-tâche traditionnel, on définit une pile par processus et la mémoire est généralement allouée dynamiquement. Si le système supporte la gestion de la mémoire virtuelle, les processus peuvent alors avoir une taille de pile par défaut assez petite, et demander à l'OS d'allouer de la mémoire virtuelle supplémentaire si besoin.

Dans TinyOS, il n'y a pas de gestion de la mémoire virtuelle, et la mémoire est allouée statiquement. Une solution serait alors d'allouer statiquement un espace en mémoire pour la pile de chaque processus présent sur le système. Cet espace mémoire doit être dimensionné pour une utilisation pire cas de la mémoire. Ceci nécessiterait donc une taille de RAM très importante, et donc une consommation électrique

relativement importante de cette mémoire.

Dans la mesure où il n'existe aucune protection des espaces mémoire dans TinyOS, il est possible de réduire l'impact de l'implantation de l'ordonnancement préemptif sur l'occupation mémoire, en n'utilisant qu'une seule pile pour toutes les tâches et l'ordonnanceur. Les contextes des tâches sont alors sauvegardés sur une même et unique pile. Ainsi, on retrouve au sommet de la pile le contexte de la tâche qui est en train de s'exécuter. Lorsqu'un appel système ou une interruption survient, le contexte de la tâche courante (T_1) est sauvegardé sur le sommet de la pile. Le traitant est donc appelé, puis lorsqu'il a terminé, c'est l'ordonnanceur, programmé en assembleur, qui s'exécute. Comme l'ordonnanceur n'utilise que des registres et des variables globales, il n'y a que le contexte de la tâche interrompue sur le sommet de la pile. Deux cas se présentent alors :

- Soit il n'y a aucune tâche plus prioritaire qui a été créée lors de cette interruption ou de cet appel système, et l'ordonnanceur retourne de l'interruption en restaurant le contexte en sommet de pile pour rendre la main à la tâche T_1 qui était en train de s'exécuter. Si une tâche qui ne requiert pas de préemption avait été créée pendant cette interruption (qui est donc moins prioritaire que T_1 au sens de l'EDF), l'ordonnanceur conserve les informations temps réel de cette nouvelle tâche dans des variables statiques (en zone *data* ou *bss*) sans créer de contexte sur la pile en mémoire.
- Soit une tâche plus prioritaire(T_2) a été créée. Dans ce cas, une préemption doit avoir lieu. Le contexte de la tâche interrompue, T_1, demeure conservé sur la pile et ne sera pas restauré au retour de cette interruption. On modifie alors l'adresse de retour de l'interruption pour que la nouvelle tâche T_2 soit exécutée. Cette adresse modifiée correspond à l'adresse d'une fonction C (ou nesC), qui doit donc retourner à l'appelant lorsqu'elle se termine, ce qui est fait généralement en dépilant l'adresse de retour qui est présente en sommet de pile. Donc pour que la tâche rende la main à l'ordonnanceur lorsqu'elle terminera, il faut préparer le sommet de la pile pour y mettre une adresse de retour de la tâche T_2 pointant vers un appel système qui exécute l'ordonnanceur. Enfin on retourne de l'interruption, sans restaurer le contexte de la tâche interrompue T_1 pour exécuter la nouvelle tâche T_2.

Lorsqu'une tâche se termine, celle-ci effectue donc un appel système pour appeler l'ordonnanceur. À ce moment là, il n'y a que le contexte d'une tâche interrompue sur le sommet de la pile. L'ordonnanceur décide ensuite s'il exécute une nouvelle tâche (en préparant la pile), ou restaure le contexte d'une tâche préemptée. S'il ne reste plus aucune tâche nesC à exécuter ou à restaurer, le contexte de la tâche initiale qui effectue en boucle des instructions "`wait_for_interrupts`" est restauré et le système se met en veille.

Prenons l'exemple de cinq tâches, montré figure 6.3. On suppose dans cet exemple que les dates de création sont $t_1 < t_2 < t_3 < t_4 < t_5$ et que les échéances sont $t_{d_5} < t_{d_2} < t_{d_3} < t_{d_1} < t_{d_4}$. Á $t = t_5$, les tâches T_1, T_2, T_3, et T_4 sont présentes dans l'ordonnanceur. Seules les tâches T_1 et T_2 possèdent un contexte sur la pile. T_2 est en train de s'exécuter. Arrive alors une interruption ou un appel système. Le contexte de la tâche T_2 est alors sauvegardé sur la pile, et la tâche T_5 est créée. L'ordonnanceur est ensuite exécuté et décide de préempter la tâche en cours pour exécuter la tâche la plus prioritaire, T_5. Le

contexte de T_2 reste donc sauvé sur la pile. Le contexte de T_5 est créé sur le sommet de la pile, et on retourne de l'interruption/appel système sans restaurer de contexte et on exécute T_5.

Lorsque T_5 termine, il rend la main à l'ordonnanceur qui restaurera le contexte de la tâche T_2 qui, lorsqu'elle terminera, rendra à son tour la main à l'ordonnanceur. Celui-ci exécutera, comme toujours, la tâche la plus prioritaire, T_3, mais sans restaurer de contexte puisqu'il s'agit d'une nouvelle tâche. Puis T_1 sera exécuter, et ensuite T_4. Enfin, comme il ne reste plus de tâche sur le système, on restaurera le contexte de la boucle principale pour que le système se mette en veille.

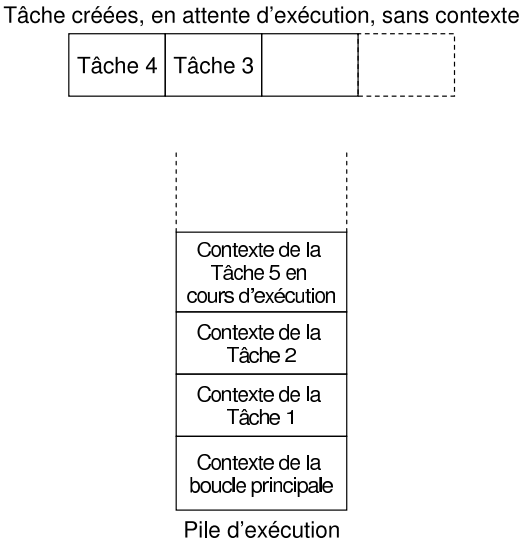


FIG. 6.3 – Pile d'exécution d'un système exécutant 5 tâches avec pour dates de création $t_1 < t_2 < t_3 < t_4 < t_5$ et pour échéances $t_{d_5} < t_{d_2} < t_{d_3} < t_{d_1} < t_{d_4}$.

Cette solution permet donc d'implanter un ordonnanceur préemptif, et en particulier ici une ordonnanceur EDF, avec un faible impact sur l'occupation mémoire. La seule limitation de cette solution réside dans le fait qu'il n'est pas possible, dans le cas général, de changer l'échéance d'une tâche qui a déjà commencé son exécution et qui a été préemptée. Ceci n'est pas un réel problème dans les cas qui nous intéressent.

6.6 Résultats

6.6.1 Intérêt du DVS

Pour évaluer l'intérêt du DVS, prenons l'exemple d'une application simplifiée de réseau de capteurs sans fil (application 1). Cette application reçoit toutes les secondes une trame qui est découpée en trois paquets. Ces paquets sont reçus à des intervalles de 50 ms. Une tâche modélise le décodage/correction de ces trames. Le temps de simulation dure 20 secondes, soit 19 réceptions de trames, soient 57 paquets.

Cette application est utilisée avec la version d'origine de TinyOS ne gérant pas le DVS, et avec

l'ordonnancement non-préemptif NP-EDF décrit précédemment gérant les tâches sporadiques. Dans cet exemple, les contraintes temporelles sont assez élevées, de telle sorte que le processeur ne reste pas trivialement à une vitesse faible, mais peut fonctionner à une vitesse supérieure à 14,5 Mips. D'ailleurs, lors de l'exécution de cette application, 20 changements d'échéances dus à des créations de tâches prioritaires se sont produits, entrainant des augmentations soudaines de la vitesse du processeur. Les résultats de simulation sont reportés dans le tableau 6.1.

TAB. 6.1 – Résultats de simulation montrant l'intérêt de la technique du DVS.

Application 1	**TinyOS 2.x "standard"**	**Gestion du DVS sans préemption**	**Différence**
Énergie consommée	5 543 652	2 356 030	-57,5 %
Instructions totales	3 725 642	3 933 673	5,6 %
initialisation	458	916	100,0 %
traitant d'IT	20 898	148 130	608,8 %
ordonnanceur	931 504	1 129 505	21,3 %
application	2 772 782	2 655 122	-4,2 %

L'intérêt de la technique du DVS est évidente : bien qu'ayant un surcoût de plus de 5% en termes d'instructions exécutées, l'utilisation de cette technique permet de réduire l'énergie consommée de près de 60%. Cette augmentation du nombre d'instructions est due à deux choses. D'une part, l'ordonnanceur NP-EDF est plus complexe que l'ordonnanceur FIFO par défaut de TinyOS, et doit gérer en même temps la vitesse du processeur en fonction des tâches s'exécutant sur le système. D'autre part, dans la mesure où il faut gérer les contraintes temps réel des tâches qui sont créées suite à une interruption timer dans le cas NP-EDF, les gestions du timer et des interruptions sont assez différentes dans les deux versions.

6.6.2 Ordonnancement préemptifs et non préemptifs

Pour comparer les ordonnancements préemptif et non préemptif, il est nécessaire de se placer dans un contexte où il y a un grand nombre de préemptions, les deux algorithmes étant similaires en dehors de ce contexte. Lors de l'exécution de l'application 1 avec les deux politiques d'ordonnancement, dans chacun des deux cas, il y a eu 20 créations de tâches prioritaires qui ont entrainé soit des préemptions dans le cas de l'EDF, soit des changements d'échéance et des modifications de la vitesse intrinsèque de la tâche en cours d'exécution dans le cas du NP-EDF. Les résultats de simulation sont reportés dans le tableau 6.2.

Malgré un paramétrage des tâches temps réel un peu moins favorable en termes de consommation (nombre d'instructions pire cas plus grand), on peut voir que l'ordonnancement avec préemptions est bien meilleur et permet de gagner près de 30% d'énergie par rapport à l'ordonnancement non préemptif, soit 70% par rapport à la version ne gérant pas le DVS.

Alors que l'on aurait pu s'attendre à avoir un nombre d'instructions exécutées bien supérieur dans le cas de la version préemptive, on remarque que les chiffres sont comparables dans les deux versions. Ceci est dû à une bonne implantation des préemptions permettant de réaliser la sauvegarde des contextes

TAB. 6.2 – Résultats de simulation montrant l'intérêt de la préemption (tâches sporadiques).

Application 1	**Gestion du DVS sans préemption**	**Gestion du DVS avec préemption**	**Différence**
Énergie consommée	2 356 030	1 701 245	-27,8 %
Instructions totales	3 933 673	3 940 187	0,2 %
initialisation	916	820	-10,5 %
traitant d'IT	148 130	166 978	12,7 %
ordonnanceur	1 129 505	1 209 090	7,0 %
application	2 655 122	2 563 299	-3,5 %

des tâches préemptées avec un coût supplémentaire très faible (dans les deux cas, il y a sauvegarde du contexte lors de la prise en compte d'appels système ou d'interruptions).

Lorsqu'il n'y a pas de préemption, les deux ordonnancements se comportent de manière semblable. La différence réside dans le fait que l'ordonnancement préemptif est un petit peu plus complexe. La différence en consommation est alors généralement la même que la différence du nombre d'instructions exécutées et ne dépasse jamais 3%.

6.6.3 Modèles de tâches périodiques et sporadiques

Pour tester l'influence du type de tâches sur la consommation, considérons une nouvelle application (application 2). Cette application modélise le même comportement que l'application 1, mais elle est découpée avec un nombre de tâches inférieur. Il n'y a aucune préemption dans cet exemple.

Deux versions de cette application ont été réalisées pour l'ordonnancement de tâches périodiques. Une version naïve (version 1), dans laquelle aucune des tâches périodiques présentes dans le système ne peut être arrêtée. Ceci génère donc des interruptions inutiles lorsque l'on sait d'avance qu'il n'y a aucun traitement à réaliser. Dans l'exemple de l'application de réseaux de capteurs sans fil choisi, s'il est nécessaire de vérifier toutes les secondes si un message a été envoyé, il est inutile de vérifier la présence, toutes les 50 ms, d'un paquet du message qui n'a pas été envoyé, ou qui a déjà été reçu complètement.

La version 2 au contraire, permet d'interrompre les occurrences périodiques d'une tâche lorsque son exécution n'est pas nécessaire. Cette implantation permet donc de modéliser un comportement plus "sporadique" que la version 1, en arrêtant la tâche périodique de période 50 ms qui permet de recevoir un paquet pour reconstituer le message complet. Ainsi, toutes les secondes cette tâche est démarrée, et est arrêtée quand il n'y a pas de transmission radio ou quand tous les paquets sont reçus. Ceci permet d'économiser un grand nombre d'interruptions timer inutiles sur le système, mais nécessite de pouvoir être géré correctement par le système.

La politique de gestion de la vitesse utilisée pour l'ordonnancement périodique est l'algorithme "cycle-conserving DVS" présenté plus haut. Le tableau 6.3 présente les résultats.

On remarque alors que la deuxième version de l'application pour l'ordonnancement périodique per-

TAB. 6.3 – Influence du modèle de tâche sur la consommation.

Application 2					
Ordonnanceur	*NP-EDF*	*EDF*	*EDF*	*EDF*	*EDF*
Modèle de tâche	*sporadique*	*sporadique*	*sporadique*	*périodique (version 1)*	*périodique (version 2)*
Gestion du DVS	*Simple*	*Simple*	*CC-RTDVS*	*CC-RTDVS*	*CC-RTDVS*
Énergie consommée	560 841	576 554	580 339	739 681	516 821
Instructions totales	1 522 767	1 564 022	1 574 334	1 998 750	1 404 458
initialisation	952	818	814	910	910
traitant d'IT	87 137	88 194	87 811	182 149	33 336
ordonnanceur	199 941	239 640	250 507	545 616	122 420
application	1 234 737	1 235 370	1 235 202	1 270 075	1 247 792

met de réduire de 30 % le nombre d'instructions exécutées et d'économiser 31 % de l'énergie par rapport à la première version.

Pour ce qui est de la comparaison avec les tâches sporadiques, on peut remarquer une économie d'énergie de près de 11 %. Ces 11 % correspondent approximativement à la différence du nombre d'instructions exécutées. Cet écart est notamment dû à la gestion des timers et du temps qui peut être centralisée au sein de l'ordonnanceur dans le cas de la version périodique, et peut donc être plus optimisée, comparativement à la gestion du temps des tâches sporadiques qui passe par une couche de virtualisation. L'ordonnancement des tâches périodiques est, de plus, moins complexe à mettre en œuvre, ce qui permet de réduire de presque moitié le nombre d'instructions exécutées pour l'ordonnanceur, à fréquence d'apparition des tâches identique.

Enfin, pour compléter cette comparaison, considérons l'application 3 suivante. Celle-ci est composée de tâches périodiques (ininterrompues). Il n'y a aucune préemption. L'une d'entre elles a une complexité variable de sorte que l'exécution réelle peut être éloignée de l'exécution pire-cas. Le résultat des simulations est montré dans le tableau 6.4.

TAB. 6.4 – Évaluation de l'ordonnancement EDF de tâches périodiques dont la charge varie.

Application 3				
Ordonnanceur	*NP-EDF*	*EDF*	*EDF*	*EDF*
Modèle de tâche	*sporadique*	*sporadique*	*sporadique*	*périodique*
Gestion du DVS	*Simple*	*Simple*	*CC-RTDVS*	*CC-RTDVS*
Énergie consommée	10 521 613	10 684 389	10 498 153	9 917 094
Instructions totales	18 234 390	18 435 974	18 494 107	17 771 724
initialisation	952	815	814	907
traitant d'IT	272 864	277 003	277 003	181 479
ordonnanceur	1 021 752	1 218 384	1 276 402	546 209
application	16 938 822	16 939 772	16 939 888	17 043 129

On remarque que, avec une gestion de la vitesse CC-RTDVS, l'ordonnancement EDF de tâches périodiques permet de gagner près de 5,5% en consommation par rapport à l'ordonnancement préemptif de tâches sporadiques, alors que le nombre d'instructions exécutées n'est inférieur que de 4%. La différence

d'énergie par instruction entre la version périodique et la version sporadique est donc encore négligeable.

En résumé, si le système supporte sporadiquement l'arrêt et le redémarrage des tâches périodiques, l'utilisation de l'ordonnancement EDF préemptif de tâches périodiques permet de réduire la consommation d'énergie grâce à une gestion plus optimisée des timers et du temps. Par contre, l'énergie par instruction étant sensiblement la même dans les deux cas, les tâches périodiques ne permettent aucune optimisation supplémentaire si on considère un ordonnancement de type EDF.

Si par contre, l'OS ne supporte pas de mettre en suspens une tâche périodique, alors l'utilisation de l'ordonnancement EDF préemptif avec des tâches périodiques est moins efficace qu'avec des tâches sporadiques, notamment à cause des nombreuses interruptions inutiles et de la vitesse accrue du processeur.

6.6.4 Algorithmes de gestion du DVS

Considérons l'ordonnancement préemptif EDF de tâches périodiques et trois politiques de gestion des tensions évoquées précédemment, c'est-à-dire la gestion statique de la vitesse, la gestion du DVS "cycle conserving" (CC-RTDVS) et la gestion "look-ahead" (LA-RTDVS). Les résultats de simulation sont décrits dans le tableau 6.5.

TAB. 6.5 – Comparatif des algorithmes de gestion du DVS.

Application 3	**Gestion statique de la vitesse**	**Algorithme "Cycle conserving"**	**Algorithme "Look-ahead"**
Énergie consommée	23 914 948	9 917 094	10 326 160
Instructions totales	17 578 145	17 771 724	18 264 054
initialisation	779	907	905
traitant d'IT	172 262	181 479	182 149
ordonnanceur	434 191	546 209	1 037 617
application	16 970 913	17 043 129	17 043 383

Dans la mesure où le nombre d'instructions exécutées peut être assez éloigné du pire cas, l'application 3 considérée est favorable à l'algorithme LA-RTDVS. On remarque néanmoins que l'énergie consommée est supérieure de 4% à l'énergie consommée avec le CC-RTDVS, alors que le nombre d'instructions exécutées n'est supérieur que de 2,5%. Bien que très efficace pour des applications plus complexes comme le décodage d'images ou de vidéos, le LA-RTDVS possède une énergie moyenne par instruction supérieure au CC-RTDVS. Cet algorithme n'est donc pas intéressant dans les applications de réseaux de capteurs sans fil. La différence est due au fait que les périodes où le processeur fonctionne à faible vitesse (suite à une fin de tâche prématurée) n'arrivent pas à compenser les moments où l'exécution est proche du pire cas et où il faut alors fonctionner à une vitesse très élevée.

La gestion statique définit une valeur fixe de la vitesse du processeur de sorte que toutes les exécutions respectent les contraintes temps réel du système. Cette valeur est donc calculée pour l'exécution pire cas en termes de temps de calcul des tâches, mais aussi en termes de dates de création des tâches. L'énergie consommée est donc nettement supérieure dans ce cas.

6.7 Conclusion

Pour calculer la consigne passée au coprocesseur DVS, il est nécessaire de doter les tâches du système de contraintes temps réel. Le problème est alors de savoir, dans un contexte de réseaux de capteurs sans fil, quels sont les algorithmes de gestion de l'énergie permettant d'optimiser la consommation d'énergie, avec quel type d'ordonnanceur, et quel modèle de tâche.

Pour pouvoir évaluer la consommation du logiciel, TinyOS a été porté et compilé vers une architecture MIPS, puis exécuté sur un simulateur de jeu d'instructions supportant le DVS afin d'évaluer précisément la consommation du logiciel. TinyOS a de plus été modifié pour permettre le support de l'ordonnancement EDF préemptif.

Au terme de ce comparatif, on peut retenir que dans la quasi-totalité des cas, l'ordonnancement EDF préemptif autorise une meilleure gestion de la vitesse du processeur. Dans les rares exemples où ce n'est pas le cas, c'est-à-dire lorsque l'ordonnancement EDF non préemptif (NP-EDF) consomme moins que l'EDF, l'énergie gaspillée ne dépasse jamais les 3%.

En ce qui concerne le type de tâche, périodique ou sporadique, on peut noter que, la plupart du temps, l'énergie moyenne par instruction est très semblable dans les deux cas lorsqu'elles sont ordonnancées par un algorithme à priorité dynamique tel que l'EDF (ou NP-EDF). Toutefois, le temps étant mieux géré avec les tâches périodiques, l'application consomme moins d'instructions. Ainsi, l'utilisation de tâches périodiques, si elle est bien gérée par l'OS, permet de gagner en consommation.

Enfin, la politique de DVS a une grande importance. Des trois algorithmes testés (gestion statique de la vitesse, CC-RTDVS, LA-RTDVS), CC-RTDVS est celui qui permet d'économiser le plus d'énergie avec toutes les applications testées, dans le cadre d'une application de réseau de capteurs.

Chapitre 7

Communications synchrones et gestion du DVS

Nous avons vu, au chapitre 4, comment le partitionnement logiciel-matériel permet de réduire considérablement l'énergie consommée par une application, et comment les communications synchrones entre le logiciel et le matériel asynchrone permettent de s'affranchir du coût de synchronisation entre le processeur et ses périphériques. Les chapitres 5 et 6 ont montré, à la fois au niveau logiciel et au niveau matériel, comment contrôler la vitesse du processeur pour réduire dynamiquement sa consommation.

Le principal problème est alors de pouvoir concilier les deux approches. En effet, il faut prendre garde à ce que le temps où le logiciel est bloqué sur une communication synchrone n'entraîne pas une violation des contraintes temps réel utilisées pour calculer la vitesse du processeur. C'est ce que propose de résoudre ce chapitre, en supposant l'utilisation d'un ordonnanceur EDF préemptif.

7.1 Principe

D'une manière générale, pour définir les paramètres temps réel d'une tâche partitionnée, il est nécessaire de considérer les temps d'exécution dans le pire cas, à la fois du logiciel (W_{log}) et du matériel (W_{mat}), pour calculer le temps d'exécution pire cas de la tâche entière W. On a alors $W = W_{mat} + W_{log}$. Dans ce cas, dans la mesure où l'on utilise des communications synchrones interruptibles, toute la théorie développée au chapitre 6 s'applique. En appelant Δ_i l'échéance relative de la tâche T_i, on peut définir le taux d'utilisation U_i et une vitesse intrinsèque S_i donnés par l'équation 7.1.

$$U_i = \frac{W_{mat_i} + W_{log_i}}{\Delta_i} \qquad et \qquad S_i = S_{max} \times U_i \tag{7.1}$$

Si on considère un processeur asynchrone, il est difficile d'avoir accès aux temps que mettent les instructions pour s'exécuter. Il est alors préférable de travailler avec les notions de vitesse et de nombre

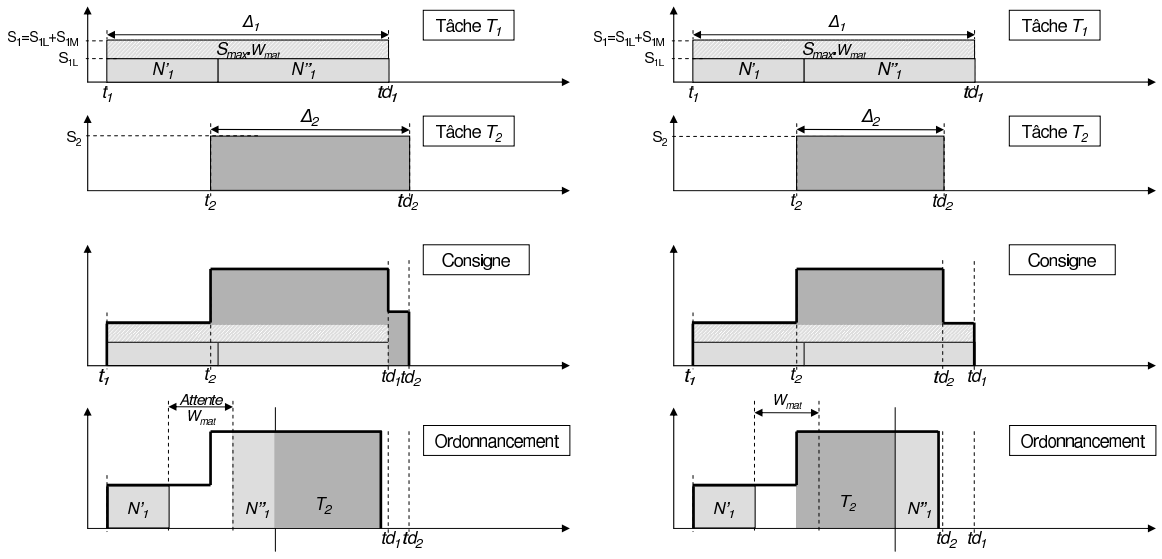
d'instructions, et d'utiliser le coprocesseur DVS décrit précédemment. On peut alors exprimer la vitesse intrinsèque S_i à l'aide des formules de l'équation 7.2.

$$S_i = \frac{N_i}{\Delta_i} + \frac{S_{max} \times W_{mat_i}}{\Delta_i} \tag{7.2}$$

Dans ce cas il est nécessaire de connaître le nombre d'instructions à exécuter dans le pire cas pour le logiciel, la durée d'exécution pire cas du matériel, ainsi que la vitesse moyenne maximale du processeur.

Prenons l'exemple de la figure 7.1 présentant deux tâches. L'une d'entre elle, T_1, est implantée en logiciel et en matériel, alors que l'autre, T_2, est purement logicielle. D'après l'équation 7.2, il y a deux contributions au calcul de la vitesse de la tâche T_1 : il y a la contribution du logiciel $S_{1_L} = \frac{N_1}{\Delta_1}$, et la contribution du matériel $S_{1_M} = \frac{S_{max} \times W_{mat_1}}{\Delta_1}$.

Dans cet exemple, la tâche T_2 est créée pendant que le processeur attend le résultat de la partie matérielle de la tâche T_1. Dans le cas de la figure 7.1(a), T_2 ne préempte pas le processeur. En revanche, dans le second cas (figure 7.1(b)), l'échéance absolue de T_2 étant la plus petite, il est nécessaire d'effectuer une préemption. Il faut donc que le processeur puisse débloquer sa communication. Comme la création de la tâche T_2 est due à une interruption, provenant du timer par exemple, il suffit d'implanter des communications synchrones interruptibles développées section 4.3.3. L'utilisation de ce type d'interface est donc nécessaire pour la gestion du temps réel avec des communications synchrones. Ceci permet, de plus, d'augmenter le parallélisme.



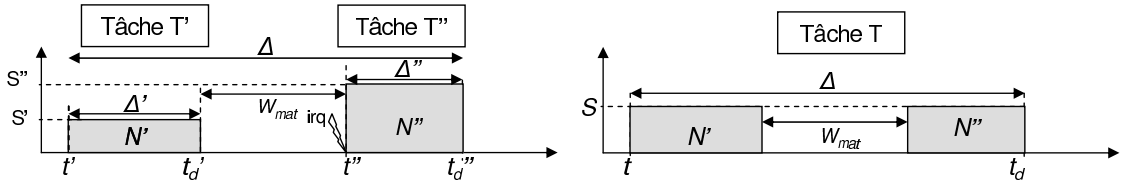
(a) sans préemption (b) avec préemption

FIG. 7.1 – Exemple de création de tâche lorsque le processeur est bloqué sur une communication synchrone interruptible.

7.2 Comparaison du profil de vitesse avec une approche traditionnelle

7.2.1 Solution traditionnelle

Avec une approche traditionnelle, un calcul, qui doit se faire à la fois en logiciel et en matériel, peut généralement se décomposer en deux tâches logicielles et un calcul matériel réalisé sur le périphérique. La figure 7.2(a) montre l'exemple d'un tel calcul avec les deux tâches logicielles T' et T'', et la tâche matérielle effectuant le calcul en un temps W_{mat}.



(a) exécution avec interruptions (b) exécution avec des communications synchrones

FIG. 7.2 – Exemple de découpage en tâche d'un traitement effectué en logiciel et en matériel, en fonction du type de synchronisation entre le processeur et son périphérique.

Si on appelle Δ l'échéance relative de la fin du calcul logiciel et matériel, Δ' (resp. Δ'') l'échéance relative de T' (resp. T''), et si on considère le modèle d'énergie variant quadratiquement avec la tension décrit au chapitre 5, équation 5.1, on peut alors exprimer l'énergie consommée pour le calcul global en logiciel et en matériel par l'équation 7.3. E_{mat} représente alors l'énergie dépensée par le calcul du périphérique, α une constante positive et S' et S'' les vitesses respectives des tâches logicielles T' et T''.

$$\begin{cases} E = \alpha.S'^2.\Delta' + \alpha.S''^2.\Delta'' + E_{mat} \\ \Delta' + \Delta'' = \Delta - W_{mat} \end{cases} \tag{7.3}$$

Il est alors facile de montrer, lorsque l'énergie, E_{mat}, dépensée par le périphérique, ne dépend pas des échéances des tâches logicielles, que l'énergie consommée par cette application atteint un minimum pour :

$$S' = S'' = \frac{N' + N''}{\Delta' + \Delta''} = \frac{N_{log}}{\Delta - W_{mat}} \tag{7.4}$$

7.2.2 Solution à base de communications synchrones

On peut remarquer dans les deux cas de la figure 7.1 que les exécutions des tâches se terminent avant leurs échéances respectives. Pourtant, dans cet exemple, les exécutions du logiciel et du matériel sont considérées comme étant les pires cas. Pour comprendre le phénomène, considérons la tâche partitionnée de la figure 7.2(b) fonctionnant à $S = \frac{N}{\Delta} + \frac{S_{max} \times W_{mat}}{\Delta}$ (équation 7.2). Dans sa pire invocation, le logiciel

possède N instructions que le processeur exécute à cette vitesse en un temps T_{exec_L} donné équation 7.5.

$$\begin{aligned} T_{exec_L} &= \frac{N}{S} \\ &= \Delta \times \frac{N}{N + S_{max} \times W_{mat}} \\ &= \Delta \times \frac{S_{max} \times W_{log}}{S_{max} \times (W_{log} + W_{mat})} \\ T_{exec_L} &= \Delta \times \frac{W_{log}}{W_{log} + W_{mat}} \end{aligned} \tag{7.5}$$

Si le matériel effectue le calcul en un temps W_{mat}, le temps d'exécution total T_{exec}, peut alors être exprimé à l'aide de la formule de l'équation 7.6.

$$\begin{aligned} T_{exec} &= T_{exec_L} + W_{mat} \\ &= \Delta \times \frac{W_{log}}{W_{log} + W_{mat}} + W_{mat} \\ T_{exec} &= \Delta \times \left(\frac{W_{log} + W_{mat} \times \frac{(W_{log} + W_{mat})}{\Delta}}{W_{log} + W_{mat}} \right) \end{aligned} \tag{7.6}$$

Or, comme il est nécessaire que $W_{log} + W_{mat} < \Delta$ pour que les contraintes temps réel puissent être respectées, on sait que le temps d'exécution T_{exec} est toujours plus petit que Δ si le temps de calcul matériel est fixe et égal à W_{mat}.

Par ailleurs, si on considère l'équation 7.2, on remarque que la vitesse intrinsèque dépend de S_{max}. On peut interpréter le terme $\frac{S_{max} \times W_{mat_i}}{\Delta_i}$ comme étant l'incrément de vitesse permettant de garantir que, même si les contraintes temps réel des tâches logicielles présentes sur le système obligent le processeur à fonctionner à S_{max}, il y aura un temps W_{mat} réservé pour le calcul matériel.

Cette vitesse S_{max} est forcément toujours inférieure à la vitesse maximale $S_{max_{techno}}$ permise par la technologie. De plus, pour qu'un ensemble de tâches puisse être ordonnançable, il faut et il suffit que la vitesse maximale du processeur S_{max} soit supérieure à la vitesse maximale $S_{max_{tâches}}$ qui peut être exigée par les tâches temps réel, et calculée par exemple grâce au test d'ordonnançabilité 6.2. On peut donc choisir une valeur de S_{max} pour laquelle on ait :

$$S_{max_{tâches}} \leqslant S_{max} \leqslant S_{max_{techno}}$$

Choisir la vitesse $S_{max} = S_{max_{tâches}}$ permet de minimiser les vitesses intrinsèques des tâches partitionnées, et donc d'économiser de l'énergie.

Cependant, malgré cette optimisation, si le processeur ne fonctionne pas à sa vitesse maximale S_{max}, celui-ci peut rester inactif pendant une certaine période de temps. Bien que des algorithmes tels que CC-RTDVS présenté à la section 6.4.3, permettent d'exploiter l'arrêt prématuré des tâches pour ralentir les autres tâches du système, les périodes d'inactvité peuvent engendrer une surconsommation d'énergie. Alors, pour réduire celle-ci, il peut être intéressant de ralentir le périphérique matériel en lui appliquant la technique du DVS.

7.3 DVS sur les périphériques

Le taux d'utilisation $U(t)$, calculé à partir des tâches prêtes à être exécutées sur le système à l'instant t, représente le coefficient permettant de réduire la vitesse du processeur tout en garantissant le respect des contraintes temps réel. Ce constat vaut aussi en particulier si on considère des tâches partitionnées. D'après l'équation 7.2, ceci signifie qu'il est possible de diminuer la vitesse de calcul du périphérique pour passer d'un temps de calcul maximum de W_{mat} à $\frac{W_{mat}}{U(t)}$. Le rapport entre les temps de calcul du matériel et celui du logiciel sont donc les mêmes, et correspondent au taux d'utilisation $U(t)$. Ainsi, comme on peut le voir sur l'exemple 7.3, diminuer la vitesse des périphériques permet de maximiser le temps de calcul tout en garantissant le respect des contraintes temps réel. En particulier, si une seule tâche est présente sur le système et si l'invocation de la tâche considérée est son pire cas, augmenter de cette manière le temps de calcul du périphérique permet d'étendre l'exécution sur toute la période correspondant à l'échéance relative Δ.

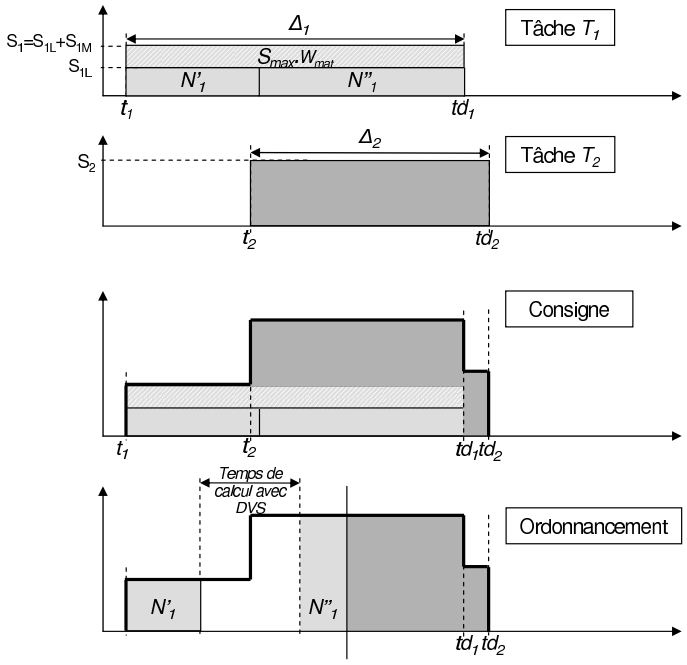


FIG. 7.3 – Exemple d'une tâche partitionnée en appliquant la technique du DVS à la partie matérielle.

Par ailleurs, on peut noter que ce taux d'utilisation dépend du temps. En effet, ce coefficient $U(t)$ peut changer dans le temps, par exemple, si une tâche est créée pendant que le processeur attend le résultat du matériel, ou si l'échéance d'une tâche est dépassée. Alors, il faut modifier la vitesse du périphérique en fonction de la variation du taux d'utilisation du système.

Faire du DVS sur un périphérique pose le problème de savoir comment contrôler son temps d'exécution ou sa vitesse, et à quel coût. En effet, si le périphérique consomme très peu d'énergie, il est sans doute préférable, du point de vue économique, de ne pas ajouter de régulateur et de laisser les algorithmes présentés au chapitre 6 tirer parti de la fin prématurée des tâches.

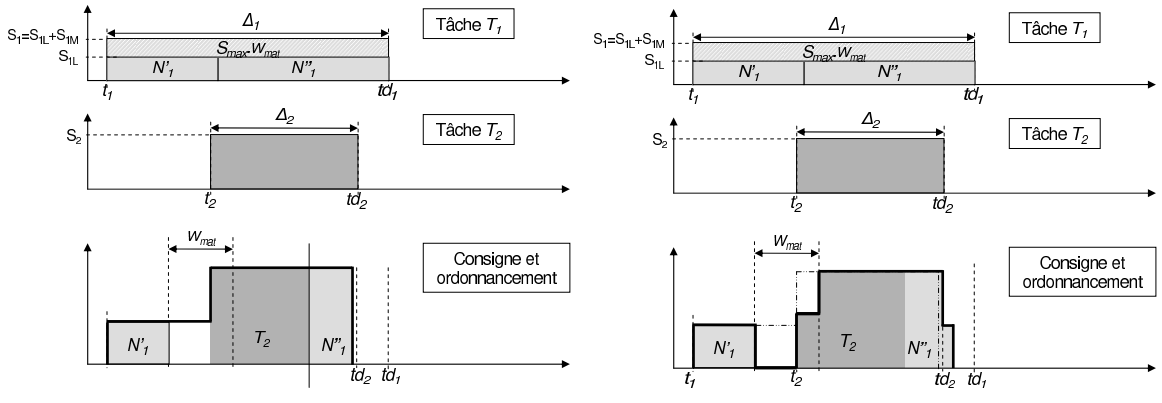
En revanche, si le périphérique consomme beaucoup, il peut être intéressant de contrôler sa vitesse

et sa tension. Dans ce cas, si la nature du calcul le permet, c'est-à-dire si on peut avoir accès à des informations permettant de mesurer la vitesse du périphérique, il est possible d'utiliser un coprocesseur comme celui présenté section 5.3. En particulier, pour le processeur cryptographique AES asynchrone du chapitre 4, dans la mesure où un chiffrement s'effectue en 16 passes, il est possible de mesurer la vitesse en cours de calcul et d'asservir cette vitesse pour avoir le temps d'exécution souhaité. Cependant, un tel coprocesseur coûte cher. Une solution bien plus économique consisterait alors à caractériser les temps d'exécution en fonction des différents niveaux de tension d'alimentation, et d'appliquer le niveau de tension adéquat en fonction des besoins, comme on le ferait typiquement en synchrone.

7.4 Optimisation de la vitesse

7.4.1 Principe

Dans certains cas, il est possible d'optimiser encore un peu plus la consommation d'énergie du système. Il est en effet possible de soustraire la contribution en vitesse des tâches qui sont bloquées sur une communication synchrone avec un périphérique. En effet, lorsqu'une tâche est partitionnée, il est nécessaire de considérer le temps d'exécution pire cas de la partie matérielle. Comme on l'a vu précédemment, au niveau du profil de vitesse des tâches, tout se passe comme si la tâche partitionnée était purement logicielle et devait exécuter $W_{mat} \times S_{max}$ instructions en plus de celles du logiciel. Cependant, il ne s'agit que d'instructions "virtuelles" permettant de réserver un créneau temporel et ne seront jamais exécutées par le processeur. On peut donc retirer la contribution de cette vitesse au niveau du processeur lorsque le périphérique travaille. Ceci revient à réduire la vitesse globale du processeur. Ainsi, il est possible d'optimiser le profil de vitesse du processeur de manière importante lorsqu'il se produit une préemption, comme on peut le voir figure 7.4, ce qui permet de maximiser le temps d'exécution dans le pire cas d'exécution.



(a) sans optimisation (b) avec optimisation

FIG. 7.4 – Profil de vitesse d'un système contenant une tâche partitionnée avec et sans optimisation pour un processeur.

Cependant, sur la figure 7.4(b), on peut remarquer qu'il est nécessaire d'augmenter la vitesse du pro-

cesseur lorsque le périphérique termine son calcul. Pour cela, comme une autre tâche peut avoir préempté le processeur pendant le calcul du périphérique, il faut lever une interruption. Or, le périphérique s'attend à ce que le processeur demande le résultat. Il est donc nécessaire d'enrichir l'interface de communication pour permettre au périphérique de choisir entre une synchronisation par interruption ou par lecture bloquante.

7.4.2 Enrichir l'interface logiciel-matériel

Afin d'enrichir l'interface entre le logiciel et le matériel présentée au chapitre 4, figure 4.11, il est possible de placer un module, que l'on nommera `sync`, permettant de choisir le mode de synchronisation adéquat. Le schéma d'une telle interface est montré figure 7.5.

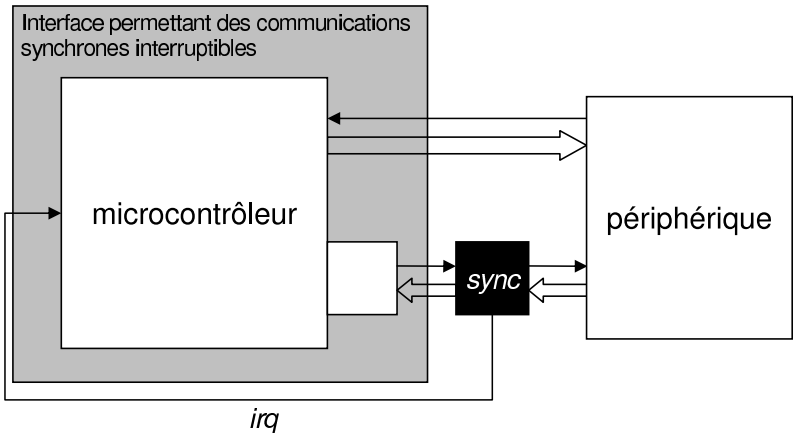


FIG. 7.5 – Schéma de l'architecture de communication

Ainsi, lorsque le périphérique termine un calcul, il positionne les données résultant du calcul sur le bus. Dans ce cas, si le processeur est bloqué en attente de ces données, il a déjà positionné une requête sur le canal de communication, et les données lui sont alors transmises par intermédiaire du module `sync`. Si au contraire le processeur n'a pas positionné son signal de requête, ce module `sync` lève une interruption lorsque le calcul du périphérique termine.

Il serait encore possible de complexifier cette interface pour permettre de la configurer. On pourrait, en particulier, interdire les interruptions et exiger de passer par des communications synchrones, même si le processeur n'est pas en attente. Une telle interface ne sera pas décrite ici.

7.4.3 Intérêt de cette modification

Ce type d'interface permet une gestion beaucoup plus souple et évolutive des communications entre le processeur et ses périphériques. De plus, il permet d'optimiser le profil de vitesse du processeur. Néanmoins, cette optimisation peut avoir un coût énergétique qui est le coût d'une interruption. Il est important d'effectuer une analyse énergétique pour évaluer l'intérêt d'une telle optimisation.

Il faut noter, tout d'abord, que cette optimisation n'a lieu que lorsque le processeur est bloqué sur une communication et qu'il se produit une interruption entraînant l'exécution de code. En dehors de ce contexte, cette optimisation n'économise pas d'énergie, mais n'en consomme pas plus non plus. Pour évaluer l'intérêt de cette optimisation, on se placera donc dans le cas où le processeur est interrompu lorsqu'il effectue une communication bloquante.

Il faut alors distinguer deux cas. D'une part, il se produit une interruption et une exécution de code, mais l'exécution de ce code se termine avant le périphérique. Dans ce cas, l'ordonnanceur va redonner la main à la tâche bloquée, et il ne se produira aucune interruption. Ainsi, cette optimisation permet d'économiser de l'énergie en faisant fonctionner le processeur moins vite.

D'autre part, le processeur est interrompu pendant qu'il est en attente de la fin d'un calcul d'un périphérique, du code est exécuté et son exécution se termine après le calcul du périphérique. Il doit alors se produire une interruption pour mettre à jour la vitesse du processeur. Dans ce cas, on gagne en vitesse, mais on perd l'énergie nécessaire pour traiter une interruption. Cette optimisation n'est alors valable que si l'énergie économisée grâce à la diminution de vitesse est plus élevée que le coût d'une interruption. En d'autres termes, plus le périphérique met de temps pour effectuer son calcul, ou plus l'abaissement de la vitesse est important, et plus il y a de chance que cette modification de l'architecture économise de l'énergie. Cette étude doit être réalisée au cas par cas, en fonction des diverses tâches du système et de leurs contraintes temps réel.

7.5 Améliorer les performances

Le partitionnement logiciel-matériel peut avoir deux objectifs. Le premier est de diminuer la consommation énergétique de l'application. C'est le cas qui nous intéresse principalement ici et pour lequel les communications synchrones ont été définies. Cependant, on peut aussi avoir besoin de partitionner le système afin de disposer de plus de performance pour permettre d'implanter une fonctionnalité qui ne pourrait pas être implantée uniquement en logiciel.

Grâce à l'utilisation des communications synchrones interruptibles, il est possible d'interrompre le processeur effectuant une lecture bloquante sur un registre de sortie d'un de ces périphériques. Ainsi, lorsqu'il se produit une interruption, du code est exécuté en parallèle du périphérique, et si une tâche est créée avec l'échéance absolue la plus petite, alors une préemption doit avoir lieu pour pouvoir l'exécuter. Ce type de mécanisme permet donc d'augmenter le parallélisme.

Cependant, si le processeur est bloqué, et que des tâches moins prioritaires sont présentes dans l'ordonnanceur, il est souhaitable, du point de vue des performances, de les exécuter le temps que le calcul matériel se termine. Ceci permet par exemple d'augmenter la réactivité du système, et par conséquent la qualité de service, notamment lorsque les temps de calcul du périphérique sont importants.

Dans ce cas, il faut que l'ordonnanceur puisse savoir quand le calcul sur le périphérique dédié se

termine afin de rendre la main à la tâche la plus prioritaire pour respecter l'ordonnancement EDF. Cette synchronisation peut s'effectuer efficacement grâce à l'utilisation du module `sync` défini précédemment. Dans ce cas, une interruption doit se produire si la quantité de code à exécuter présente dans l'ordonnanceur est suffisante. Alors, l'optimisation décrite précédemment permet de réduire la vitesse du processeur sans ajouter le moindre surcoût. En revanche, s'il n'y a pas d'autre tâche à ordonnancer, il n'y a aucun coût de synchronisation entre le processeur et le périphérique, puisque celle-ci s'effectue avec une lecture bloquante.

L'augmentation de performance ne se fait cependant pas sans un certain coût énergétique. En effet, d'une part, il doit se produire une interruption qui coûte de l'énergie dans la plupart des cas mais permet d'optimiser le profil en vitesse. De plus, cette interruption s'accompagne du coût des changements de contexte à opérer lorsque la tâche bloquée passe la main à l'ordonnanceur pour traiter une autre tâche, et lorsque le calcul matériel se termine.

D'autre part, dans le cas présent, l'amélioration du parallélisme oblige à pouvoir exécuter les tâches dans un ordre qui ne respecte pas strictement l'ordre de l'EDF. Ceci pose un problème en particulier avec notre implantation de l'ordonnanceur EDF qui ne possède qu'une seule pile pour toutes les tâches et qui suppose que la tâche qui s'exécute est la tâche qui est en sommet de pile (voir paragraphe 6.5.2.2). Cette implantation n'est plus possible avec ce type de parallélisme puisque la tâche qui s'exécute n'est pas forcément la plus prioritaire au sens de l'EDF. Pour résoudre ce problème, il est donc nécessaire de créer une pile par tâche, ce qui peut augmenter considérablement la taille de la mémoire, et la consommation associée à celle-ci.

En conclusion, que ce soit à cause de l'implantation logicielle choisie ou du surcoût dû aux synchronisations supplémentaires, l'amélioration des performances coûte généralement en énergie. Il est donc important de prendre cet aspect en considération lors de la spécification de l'architecture. Par exemple, s'il y a un important besoin de performances pour une application donnée, il peut être plus efficace en énergie d'utiliser un processeur légèrement plus puissant, quitte à le sous-exploiter en vitesse, que d'augmenter, d'une part, la taille de la mémoire nécessaire de manière significative, et la charge processeur à cause des interruptions et des changements de contextes, d'autre part.

7.6 Bilan énergétique

Pour évaluer les différentes solutions, reprenons l'exemple présenté au chapitre 4 du chiffrement de 1 ko de donnée par l'algorithme AES 128 bits. Pour avoir une évaluation précise, il faut appliquer les même contraintes temps réel à chacune des solutions envisagées. Comme un calcul AES matériel est beaucoup plus rapide que son équivalent logiciel, il est nécessaire d'utiliser un microcontrôleur suffisamment rapide pour avoir une comparaison réaliste. La vitesse du msp430 n'étant pas suffisante, nous considérerons l'utilisation du microcontrôleur asynchrone MICA (surcadencé) décrit à la section 3.1.1.2 et dont les courbes sont présentées figure 5.2.

7.6.1 Solution purement logicielle

L'algorithme AES n'étant pas disponible pour le microcontrôleur MICA, nous transposerons les données disponibles pour le msp430 à MICA pour permettre la comparaison de l'énergie consommée entre une solution purement logicielle avec DVS et les solutions où le calcul de l'AES s'effectue en matériel. En particulier, pour un chiffrement AES 128 bits, le msp430 prend 5432 cycles. Dans la mesure où ce microcontrôleur met 1 à 6 cycles pour exécuter une instruction, on considérera que sa vitesse est de 1,5 Mips à 4 MHz, c'est-à-dire une instruction pour 2,6 cycles en moyenne. On peut donc estimer le nombre d'instructions nécessaire pour effectuer une telle opération à environ 2000 instructions.

Supposons que ce chiffrement doive s'effectuer avec une échéance relative Δ de 5 ms. Le microcontrôleur MICA fonctionne, à la tension de 3,5 V, à une vitesse de S_{max}=31,3 Mips pour une consommation de 2,46 nJ par instruction (voir figure 5.2). La tension d'alimentation du microcontrôleur est volontairement choisie supérieure à sa tension nominale, qui est de 2,5 V, afin de permetttre une comparaison aisée entre la solution logicielle et les solutions matérielles. Avec l'implantation purement logicielle, les 2000 instructions d'un chiffrement 128 bits prennent donc 63,9 μs pour s'exécuter et consomment 4,92 μJ. Il faut donc 4 ms pour effectuer les 64 calculs correspondant au chiffrement de la donnée totale de 1 ko, pour une consommation de 315 μJ.

L'utilisation de cette tâche sur le processeur est donc de $U = \frac{4}{5} = 0,8$. Ainsi, en appliquant la technique du DVS sur cette tâche, il est possible de faire fonctionner le processeur à une vitesse de $U \times S_{max} = 25$ Mips tout en respectant les échéances. Pour cette vitesse, la tension d'alimentation qu'il faut appliquer est de 2,7 V. Comme le processeur consomme en moyenne 1,3 nJ par instruction à cette tension, la consommation totale est de 166 μJ pour le chiffrement de toute la donnée. Avec une utilisation de 0,8, cette tâche logicielle est très contrainte en temps d'exécution. Malgré cela, la technique du DVS permet de réduire par deux l'énergie consommée par cette tâche.

7.6.2 Partitionnement avec synchronisation systématique par interruption

Considérons maintenant le calcul de l'AES réalisé par le périphérique AES asynchrone présenté section 5.1.3 et dont les courbes sont montrées figure 5.1. Au début de chaque calcul, il est nécessaire que le logiciel renseigne la donnée à chiffrer. De même, à la fin des calculs, il faut lire la donnée cryptée. Cette donnée étant sur 128 bits, il faut chaque fois 16 opérations pour écrire cette donnée avec un microcontrôleur 8 bits, et le même nombre d'opérations pour lire le résultat. Ainsi, à chaque chiffrement de 128 bits, on peut considérer qu'il y a environ une cinquantaine d'instructions qui sont exécutées, soit un nombre d'instructions $N_{app} = 3200$ pour chiffrer toute la donnée. En plus de ces N_{app} instructions, il faut tenir compte du coût de chaque interruption. Alors, pour la sauvegarde des registres, la lecture du registre de statut, l'appel au bon traitant d'interruption, le retour du traitant, la restauration du contexte, et le retour de l'interruption, on peut considérer le coût d'une interruption à $N_{IT} = 60$ instructions. Le coût logiciel total de la tâche est donc $N = N_{app} + 64 \times N_{IT} = 7040$ instructions.

En ce qui concerne le traitement matériel, à vitesse maximale, un chiffrement AES 128 bits consomme 11,6 nJ et effectue le calcul en un temps $W_{mat} = 1$ μs. En supposant une échéance relative de $\Delta = 5$ ms, et en utilisant la formule 7.4, on peut calculer la vitesse requise par le logiciel pour respecter les échéances, $S = \frac{N}{\Delta - 64 \times W_{mat}} = 1,43$ Mips. Cette vitesse est en-dessous de la vitesse minimale du microcontrôleur MICA, qui est de 4,3 Mips. Ainsi, avec ces contraintes temps réel, le microcontrôleur fonctionne à 4,3 Mips. À cette vitesse, le microcontrôleur consomme 0,19 nJ par instruction. La consommation totale du logiciel est donc de 1,34 μJ. La consommation totale de l'application est donc de 2,08 μJ. On peut noter la réduction de la consommation d'un facteur 80, à contraintes temporelles identiques, que permet cette solution par rapport à une solution purement logicielle. Ce rapport pourrait même être encore plus grand si le microcontrôleur pouvait encore abaisser un peu plus sa vitesse de fonctionnement pour fonctionner à 1,43 Mips.

Si on effectue le même calcul avec une échéance $\Delta = 750$ μs, on obtient une vitesse du microcontrôleur de 10,3 Mips. À cette vitesse, celui-ci consomme 0,33 nJ par instruction. Ainsi, la consommation du logiciel passe à 2,32 μJ, et l'application complète consomme un total de 3,07 μJ. Dans ce cas, par rapport à la solution purement logicielle, le temps de calcul est divisé par 6,6 et la consommation par 54.

7.6.3 Partitionnement avec communication synchrone uniquement

L'utilisation du mécanisme de communication bloquante permet de s'affranchir du coût de la synchronisation logiciel-matériel. Ainsi, le nombre d'instructions à exécuter est $N = N_{app} = 3200$ instructions. D'après l'équation 7.2, lorsque le périphérique AES fonctionne à vitesse maximale, et si l'échéance relative est $\Delta = 5$ ms, la vitesse du microcontrôleur doit être de 0,65 Mips pour minimiser l'énergie consommée par la partie logicielle. Dans ce cas, le microcontrôleur fonctionne à sa vitesse minimale de 4,3 Mips en moyenne, la consommation du logiciel est de 608 nJ et la consommation totale est de 1,35 μJ.

Avec une échéance relative $\Delta = 750$ μs, la vitesse requise par le microcontrôleur est de 4,3 Mips. Cette vitesse correspond à la vitesse minimale. La consommation du logiciel est donc la même que précédemment, et la consommation totale est aussi identique lorsque le périphérique fonctionne à vitesse maximale.

La différence de consommation avec la solution utilisant systématiquement les interruptions est alors considérable. En effet, pour la tâche considérée, avec des contraintes temps réel assez lâches, l'économie réalisée s'élève à 35 %. Celle-ci passe à 56 % avec des temps d'exécution plus courts.

Par ailleurs, comme on l'a vu à la section 7.3, en utilisant des communications synchrones, il est possible d'appliquer la technique du DVS sur le périphérique. Pour cela, il est nécessaire de calculer l'utilisation de la tâche logicielle sur le microcontrôleur pour appliquer la réduction de vitesse au périphérique. Ainsi, dans le cas où $\Delta = 5$ ms, l'utilisation du logiciel est $U = \frac{S}{S_{max}} = \frac{0,65}{31,3} = 2$ %. Il est alors possible de réduire la vitesse du périphérique à 2 % de la vitesse maximale du périphérique.

Cette vitesse est en-dessous de la vitesse du périphérique à la tension minimale. En appliquant la vitesse minimale, l'énergie par chiffrement passe à 11,6 nJ à 1,26 nJ par chiffrement de 128 bits. Ainsi, la consommation totale de l'application est de 688 nJ. Dans ce cas, la technique du DVS appliquée au périphérique permet de diminuer de presque 10 l'énergie consommée par le calcul matériel de l'AES, et par deux l'énergie totale consommée par l'application.

En contraignant un peu plus le temps d'exécution, en choisissant $\Delta = 750\ \mu$s, le taux d'utilisation du microcontrôleur pour la tâche est de $U = \frac{S}{S_{max}} = \frac{4,3}{31,3} = 13,7\ \%$. Ceci signifie que l'on peut diminuer la vitesse de calcul du coprocesseur AES pour le faire consommer 2,1 nJ par chiffrement de 128 bits au lieu de 11,6 nJ à vitesse maximale. L'énergie consommée par l'application au total passe donc de 1,35 μJ à 742 nJ. L'utilisation du DVS sur le périphérique permet donc de réduire de 45 % l'énergie consommée.

7.6.4 Récapitulatif

Le tableau 7.1 récapitule les consommations. Comme on peut le voir, l'implantation en matériel de la tâche de chiffrement AES permet de réduire considérablement la consommation. De plus, l'utilisation de communications synchrones avec le périphérique matériel sur lequel on applique la technique du DVS permet de réduire la consommation d'énergie d'un facteur 3 pour $\Delta = 5$ ms et 4,1 pour $\Delta = 750\ \mu$s.

TAB. 7.1 – Consommation des différentes implantations du chiffrement AES d'une donnée de 1 ko.

Solution	$\mathbf{\Delta}$ (μs)	**Vitesse** (**Mips**)	$\mathbf{E_{log}}$ (**nJ**)	$\mathbf{E_{mat}}$ (**nJ**)	$\mathbf{E_{tot}}$ (**nJ**)
logicielle pure	5 000	25	166 400	0	166 400
partitionnée – avec interruptions	5 000	1,43 → 4,3	1 338	742	2 080
	750	10,3	2 323	742	3 065
partitionnée – avec communications synchrones	5 000	0,65 → 4,3	608	742	1 350
	750	4,3	608	742	1 350
partitionnée – avec communications synchrones et DVS matériel	5 000	0,65 → 4,3	608	80	688
	750	4,3	608	134	742

7.7 Conclusion

En incluant le temps de calcul du matériel, il est possible d'étendre les théories présentées au chapitre 6 pour calculer la vitesse intrinsèque de la tâche partitionnée. En général, la vitesse ainsi calculée est alors légèrement plus élevée que celle calculée avec une approche classique où les synchronisations entre le logiciel et le matériel ont lieu avec des interruptions. Cependant, cette méthode permet de s'affranchir du coût de synchronisation entre le logiciel et le matériel qui peut s'avérer très important. De plus, cette approche fournit un moyen simple de diminuer la vitesse des périphériques, en leur appliquant la technique du DVS. Dans ce cas, il est possible de maximiser le temps d'exécution, en réduisant la consommation à la fois du processeur et des périphériques.

En prenant l'exemple d'une application de chiffrements AES sur le microcontrôleur asynchrone

MICA, nous avons pu montrer que cette technique, avec gestion de la vitesse du périphérique, permet de réduire d'un facteur supérieur à 4 la consommation d'énergie de l'application entière par rapport à une approche classique où la synchronisation entre le processeur et les périphériques se fait à l'aide d'interruptions.

Par ailleurs, il est intéressant de modifier l'interface entre le logiciel et le matériel afin de permettre la commutation automatique entre le mode de synchronisation par interruption, ou le mode de communication synchrone. En effet, ce type d'interface permet une gestion beaucoup plus souple et évolutive des communications entre le processeur et ses périphériques. De plus, il permet d'optimiser le profil de vitesse du processeur. Néanmoins, cette optimisation peut avoir un coût énergétique qui est le coût d'une interruption. Il est important d'effectuer une analyse énergétique pour évaluer l'intérêt d'une telle optimisation.

D'autre part, du fait qu'une partie des calculs soit déportée vers du matériel dédié, on peut envisager d'augmenter les performances en exécutant, en parallèle du traitement matériel, du code sur le processeur lorsque des tâches sont prêtes à être exécutées. Cependant, bien qu'il soit possible d'effectuer des optimisations sur la vitesse de fonctionnement, ce type de parallélisme coûte en énergie puisqu'il est alors nécessaire d'interrompre le processeur et d'effectuer des changements de contexte. De plus, cela suppose une implantation de l'ordonnanceur qui est coûteuse en mémoire. Ce type d'augmentation de performance a donc un impact non négligeable sur la consommation d'énergie.

Conclusion et perspectives

Conclusion

Les perspectives applicatives offertes par les réseaux de capteurs sans fil sont nombreuses et variées. Cependant, la durée de vie des réseaux est encore actuellement un facteur limitant le développement de ces applications. Il est donc nécessaire de concevoir des protocoles de communication et d'auto-organisation permettant de minimiser l'énergie consommée dans le réseau, ainsi que des plateformes matérielles ultra faible consommation.

Si on fait un bilan énergétique au niveau d'un nœud, une grande partie de l'énergie est consommée par la radio. Cependant, la consommation due à la partie numérique de celui-ci est loin d'être négligeable. C'est pourquoi il est nécessaire de réduire la consommation de celle-ci afin d'augmenter la durée de vie du réseau de capteurs, et permettre le développement de l'intelligence ambiante.

Les circuits asynchrones possèdent de nombreux avantages. Tout d'abord, contrairement aux circuits synchrones, comme la synchronisation entre les blocs est effectuée localement, seules les parties matérielles effectuant un calcul consomment de l'énergie. Les autres parties sont donc en veille et se réveillent immédiatement lorsqu'une donnée est présente en entrée. Ceci rend les mécanismes de réduction de l'activité des circuits synchrones inutiles, et simplifie donc leur gestion au niveau logiciel.

De plus, comme on peut le voir à la section 3.1.1, l'utilisation d'un microcontrôleur asynchrone ultra faible consommation permet de réduire de manière importante la consommation du logiciel, ou bien d'augmenter nettement les performances du système, comparativement à des microcontrôleurs synchrones très faible consommation classiques tels que le msp430. Ceci permet d'ouvrir de nouvelles perspectives applicatives aux réseaux de capteurs en permettant par exemple d'effectuer plus de calculs sur les nœuds du réseaux.

Bien que surdimensionner un système en choisissant un microcontrôleur trop puissant pour une application donnée coûte de l'énergie, il peut être intéressant de faire fonctionner ce microcontrôleur en-dessous de ses capacités. Ainsi, si le coût en énergie d'une telle approche est faible, ce qui est le cas si on considère les microcontrôleurs asynchrones présentés dans cette thèse, on peut alors utiliser un micro-

contrôleur assez puissant, ce qui permet de garder le même cœur numérique pour toutes les applications de réseaux de capteurs sans fil, ou au moins de limiter le nombre de plateformes. Ainsi, le temps et le coût de développement pour chaque application s'en trouvent réduits. Le coût de fabrication peut aussi être réduit lorsque les nœuds sont fabriqués en plus grand nombre.

Au niveau logiciel, l'utilisation d'un système d'exploitation dédié aux réseaux de capteurs ou plus généralement aux systèmes très fortement contraints en mémoire permet de réduire la consommation d'énergie de manière significative. Une étude comparative [44] a permis de montrer que l'utilisation de TinyOS permet de réduire par 55 l'énergie consommée par l'OS comparativement à eCos, un système d'exploitation embarqué généraliste.

Un moyen très efficace de réduire la consommation d'énegie est d'implanter des tâches applicatives en matériel. Ce partitionnement logiciel-matériel doit être soigneusement effectué pour optimiser la consommation d'énergie en fonction des contraintes de surface matérielle, coût de production, ou encore d'évolutivité de la plateforme.

Une manière d'envisager ce partitionnement a été abordée en utilisant une description globale de l'application à l'aide d'un langage basé sur les processus communicants, le CHP. Après simulation de l'application globale, il est aisé de définir les blocs à spécifier en matériel, et ceux à spécifier en logiciel, afin de minimiser la consommation. Cependant, modéliser finement une application globale à l'aide de processus communicants n'est pas une tâche facile. De plus, une fois le partitionnenment effectué, ordonnancer statiquement les tâches logicielles décrites en CHP n'est, en général, pas possible. C'est pourquoi nous avons abandonné cette modélisation au profit d'une approche plus classique.

Par ailleurs, l'absence d'horloge dans les circuits asynchrones QDI permet de gérer simplement les variations de tensions. Ceci permet, d'une part, de s'affranchir des problèmes de stabilité des alimentations lors de la conception et, d'autre part, de modifier les valeurs de la tension librement, sans avoir à modifier la fréquence d'une horloge. Ainsi, des techniques comme l'adaptation dynamique en tension (DVS) ou comme la polarisation du substrat (ABB) permettent de réduire la consommation d'énergie dynamique et les courants de fuite des circuits sans aucun surcoût énergétique ou temporel pour synchroniser l'horloge.

Afin de gérer efficacement la technique du DVS, un coprocesseur a été spécifié. Il permet d'asservir la vitesse d'un processeur en fonction d'une consigne définie par le logiciel. La vitesse du processeur est alors mesurée dynamiquement par ce coprocesseur, puis comparée à la consigne logicielle. En fonction de la différence et des différences passées, un contrôleur numérique modifie la tension d'alimentation du processeur pour modifier sa vitesse. Cette approche permet de conserver un comportement moyen en vitesse, et non un comportement pire cas utilisé lors des caractérisations tension/vitesse, notamment avec les circuits synchrones lors de leur conception. Ceci permet, à une vitesse donnée, de réduire la tension d'alimentation du processeur, et donc sa consommation d'énergie. Une modélisation en VHDL-AMS a permis de valider un tel coprocesseur à l'aide de simulations et d'évaluer l'énergie consommée par le processeur pour des profils applicatifs donnés.

Au niveau logiciel, afin de calculer la consigne en vitesse passée au coprocesseur, il est nécessaire de doter les tâches s'exécutant sur le système de contraintes temps réel. Pour évaluer les différentes solutions logicielles, nous avons choisi TinyOS comme système d'exploitation, en raison de sa légèreté. Celui-ci a été porté vers une architecture MIPS R4000. Il a été profondément remanié afin d'autoriser les préemptions pour en mesurer l'impact sur la consommation. Une implantation astucieuse, basée sur un ordonnanceur EDF, permet de réduire l'encombrement mémoire en n'utilisant qu'une seule pile. Un simulateur de jeu d'instructions a été utilisé pour mesurer la consommation. Il est enrichi de périphériques tels que le coprocesseur DVS, un périphérique de débogue ou encore un périphérique permettant de tracer l'exécution du code pour évaluer le coût processeur des différentes parties logicielles.

Ceci a permis de montrer que l'utilisation d'un ordonnanceur EDF préemptif permet de réduire la consommation du processeur de manière significative, malgré le coût des préemptions. De plus, l'algorithme de gestion de la vitesse *cycle-conserving DVS* est le plus efficace dans un contexte d'applications de réseaux de capteurs sans fil. Enfin, au niveau du modèle de tâche utilisé, on peut retenir, d'une part, que les tâches périodiques permettent une gestion plus optimisée du temps, contrairement aux tâches sporadiques. D'autre part, il est important de pouvoir arrêter une tâche périodique si le traitement périodique n'est plus nécessaire. Un modèle hybride permettant sporadiquement le commencement et l'arrêt de tâches périodiques est donc souhaitable du point de vue de la consommation.

Le chapitre 4 nous a permis de constater que l'implantation en matériel de traitements logiciels coûteux est très efficace en énergie. Cependant, dans le cas, par exemple, d'un chiffrement AES en matériel, le coût pour synchroniser le processeur et le périphérique à l'aide d'interruptions est bien plus grand que le coût du calcul lui-même. Pour résoudre le problème, il est possible d'utiliser une synchronisation par lecture bloquante (synchrone), basé sur le protocole "poignée de mains". L'implantation de celle-ci peut être réalisée sans coût supplémentaire grâce à l'utilisation de matériel asynchrone. Afin d'augmenter le parallélisme, nous avons spécifié une interface entre le processeur et les périphériques permettant à une lecture bloquante d'être interruptible, afin de traiter des évènements urgents. Cette interface peut être améliorée en autorisant une commutation automatique entre le mode de synchronisation par interruption, et le mode de communication synchrone. Ce type d'interface permet alors une gestion beaucoup plus souple et évolutive des communications entre le processeur et ses périphériques.

Enfin, la théorie pour gérer le DVS doit être étendue afin de supporter le partitionnement en matériel de certaines tâches logicielles. En utilisant cette théorie, il est possible d'étendre la technique du DVS sur les parties matérielles, ce qui permet alors de réduire encore plus l'énergie consommée par une application. De plus, grâce aux améliorations apportées aux interfaces entre le logiciel et le matériel, il est possible d'optimiser le profil de vitesse du processeur. Néanmoins, cette optimisation coûte de l'énergie puisqu'elle peut nécessiter une interruption.

Perspectives

Un certain nombre d'améliorations peut être apporté à ce travail. La première serait de réaliser l'implantation du coprocesseur que nous avons spécifié. En particulier, une étude d'automatique complète doit être réalisée au niveau du contrôleur numérique de ce coprocesseur, afin de définir au mieux les paramètres PID. Ceci permettrait d'avoir un meilleur contrôle de la vitesse et de la tension du processeur, c'est-à-dire en obtenant rapidement la vitesse désirée et en supprimant ou réduisant les oscillations que l'on peut voir lorsque les paramètres sont définis à la main. De plus, une telle implantation permettrait de fournir des informations sur la consommation totale du coprocesseur pour évaluer et comparer globalement et très précisément les gains en énergie. Une thèse dans ce domaine vient d'ailleurs de débuter dans le groupe CIS au laboratoire TIMA.

Une seconde amélioration concerne le simulateur de jeu d'instructions. Actuellement, celui-ci se limite à la consommation du logiciel. Il serait intéressant d'inclure dans les modèles de consommation et de temps du simulateur ceux des périphériques qui entrent en jeu dans l'application, et de modéliser finement les interfaces synchrones entre le processeur et les périphériques. Pour l'instant, la consommation du processeur est une consommation relative qui est utilisée pour comparer les solutions logicielles. Pour inclure les modèles des périphériques, il faudrait donc avoir un modèle de consommation "absolu". Si on essaie de partitionner le système et que le matériel n'est pas encore disponible, il faut alors avoir une description précise du matériel de manière à permettre une simulation électrique. En plus des périphériques, il faudrait ajouter un modèle de consommation des différentes mémoires qui composent le système (SRAM, EEPROM, Flash) afin d'obtenir un résultat encore plus réaliste.

Par ailleurs, nous avons utilisé TinyOS comme base de système pour évaluer nos différents algorithmes d'ordonnancement et de gestion de l'énergie. Ce système n'est pas fait pour supporter les préemptions. En particulier, il n'y a pas de gestion ou de détection d'accès concurrents à une variable partagée. Ce qui est fait actuellement est donc de masquer les interruptions avant l'accès à une variable, puis de les démasquer lorsque l'accès est terminé. Bien qu'il s'agisse de programmation système classique, ceci est contraignant pour décrire des applications complexes en nesC et peut s'avérer être source de bogues difficiles à trouver. Il serait donc intéressant d'évaluer d'autres systèmes d'exploitation pour réseaux de capteurs, comme Think par exemple.

Enfin, pour minimiser la consommation d'énergie, il peut être intéressant de modifier conjointement la tension d'alimentation et la tension de polarisation du substrat. Ceci permettrait de réduire les courants de fuite, qui sont de plus en plus importants avec les technologies récentes, pendant que le processeur effectue un calcul.

Bibliographie

[1] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, and P. Vivet. A new contactless smart card IC using an on-chip antenna and an asynchronous microcontroller. *Solid-State Circuits, IEEE Journal of*, 36(7) :1101–1107, 2001.

[2] F. Aeschlimann, E. Allier, L. Fesquet, and M. Renaudin. Asynchronous FIR filters : towards a new digital processing chain. *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pages 198–206, 2004.

[3] F. AESCHLIMANN, E. ALLIER, L. FESQUET, and M. RENAUDIN. Etude spectrale de l'échantillonnage par traversée de niveaux. 2005.

[4] E. Allier, G. Sicard, L. Fesquet, and M. Renaudin. A new class of asynchronous A/D converters based on time quantization. *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 196–205, 2003.

[5] A. Bachir, D. Barthel, M. Heusse, A. Duda, and R.D. France Telecom. Micro-Frame Preamble MAC for Multihop Wireless Sensor Networks. *2006. ICC'06. IEEE International Conference on Communications*, 7, 2006.

[6] E. BEIGNÉ and P. VIVET. Design of on-chip and off-chip interfaces for GALS NoC architecture. In *12th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 172–181, Grenoble, France, March 2006.

[7] K. Van BERKEL. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2) :103–128, 1992.

[8] K.V. BERKEL, M.B. JOSEPHS, and S.M. NOWICK. Scanning the technology : Applications of asynchronous circuits. *Proc. IEEE*, 87(2) :223–233, February 1999.

[9] P. Bonnet, J. Gehrke, and P. Seshadri. Querying the physical world. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 7(5) :10–15, 2000.

[10] D. BORRIONE, M. BOUBEKEUR, E. DUMITRESCU, M. RENAUDIN, J.B. RIGAUD, and A. SIRIANNI. An approach to the introduction of formal validation in an asynchronous circuit design flow. In *26th Annual Hawaï International Conference on System Sciences*, Big Island, Hawaï, 2003.

[11] F. BOUESSE. *Contribution à la conception de circuits intégrés sécurisés : l'alternative asynchrone*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, 2005.

[12] Fraidy Bouesse, Marc Renaudin, Adrien Witon, and Fabien Germain. A clock-less low-voltage aes crypto-processor. In *16th European Solid-State Circuits Conference (ESSCIRC 2005)*, pages 403–406, Grenoble, France, 2005.

[13] J.G. BREDESON and P.T. HULINA. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20 :114–224, 1972.

[14] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model. *Specification. Draft, France Telecom R&D*, 2004.

[15] W.A. CLARK. Macromodular computer systems. In *Spring Joint Computer Conference, AFIPS*, April 1967.

[16] C. CONSTANTINESCU. Trends and challenges in VLSI reliability. *IEEE micro*, 23(4) :14–99, July 2003.

[17] J. Cortadella. Quasi-static Scheduling for Concurrent Architectures. *Fundamenta Informaticae*, 62(2) :171–196, 2004.

[18] A.L. DAVIS. The architecture and system method of DDM-1 : A recursively-structured data driven machine. In *Fifth Annual Symposium on Computer Architecture*, 1978.

[19] Denyer and Renshaw. *VLSI Signal Processing : A Bit-serial Approach*. Addison-Wesley, 1985.

[20] M.G. Di Benedetto. *Uwb Communication Systems : A Comprehensive Overview*. Hindawi Publishing, 2006.

[21] A.-V. DINH-DUC, L. FESQUET, and M. RENAUDIN. Synthesis of QDI asynchronous circuits from DTL-style petri nets. In *11th IEEE/ACM International Workshop on Logic and Synthesis*, 2002.

[22] A.V. DINH-DUC. *Synthèse automatique de circuits asynchrones QDI*. PhD thesis, INP of Grenoble, 2003.

[23] A.V. DINH DUC, J.B. RIGAUD, A. REZZAG, A. SIRIANNI, J. FRAGOSO, L. FESQUET, and M. RENAUDIN. TAST CAD tools. In *Asynchronous Circuit Design*, Munich, Germany, 2002.

[24] M. Dohler, D. Barthel, F. Maraninchi, L. Mounier, S. Aubert, C. Dugas, A. Buhrig, F. Paugnat, M. Renaudin, A. Duda, et al. The ARESA Project : Facilitating Research, Development and Commercialization of WSNs. *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON'07. 4th Annual IEEE Communications Society Conference on*, pages 590–599, 2007.

[25] A. DUNKELS, B. GRONVALL, and T. VOIGT. Contiki-a lightweight and flexible operating system for tiny networked sensors Local Computer Networks. *Proceedings of the 29th Annual IEEE International Conference onLocal ComputerNetworks (LCN'04)*, 462, 2004.

[26] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads : simplifying event-driven programming of memory-constrained embedded systems. *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, 2006.

[27] Virantha Ekanayake, IV Clinton Kelly, and Rajit Manohar. An ultra low-power processor for sensor networks. *SIGARCH Comput. Archit. News*, 32(5) :27–36, 2004.

[28] V.N. Ekanayake, C. Kelly IV, and R. Manohar. BitSNAP : Dynamic Significance Compression For a Low-Energy Sensor Network Asynchronous Processor. *Proc. ASYNC*, pages 144–154, 2005.

[29] J.P. Fassino, J.B. Stefani, J. Lawall, and G. Muller. THINK : A Software Framework for Component-based Operating System Kernels. *Proceedings of Usenix Annual Technical Conference*, 2002.

[30] B. FOLCO. *Contribution à la synthèse de circuits asynchrones Quasi Insensibles aux Délais, application aux circuits sécurisés*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, 2007.

[31] D. Gay, P. Levis, R. von Behren, E. Brewer Welsh, and David E. Culler. The nesc language : A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA, 2003.

[32] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05 : Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[33] R.I. Hartley and P.F. Corbett. A digit-serial silicon compiler. *Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 646–649, 1988.

[34] S. HAUCK. Asynchronous design methodologies : An overview. *Proceedings of the IEEE*, 83(1) :69–93, January 1995.

[35] T. He, B. Krogh, S. Krishnamurthy, J.A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, and J. Hui. Energy-efficient surveillance system using wireless sensor networks. *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 270–283, 2004.

[36] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, volume 34, pages 93–104. ACM press, 2000.

[37] C.A.R. HOARE. Communicating sequential processes. *Communications of the ACM 21*, 8 :666–677, August 1978.

[38] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, 2005.

[39] C. Kelly IV, V. Ekanayake, and R. Manohar. SNAP : a Sensor-Network Asynchronous Processor. *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 24–33, 2003.

[40] C. Koch-Hofer, M. Renaudin, V. Thonnart, and P. Vivet. ASC, a SystemC extension for modeling asynchronous systems, and its application to an asynchronous NoC. *Proc. of the ACM/IEEE Int. Symp. on Networks-on-Chip (NOCS), May*, 2007.

[41] Estelle Labonne, Gilles Sicard, and Marc Renaudin. Dynamic voltage scaling and adaptive body biasing study for asynchronous design, 2004.

[42] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Battery-driven system design : a new frontier in low power design. *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th*

Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings., pages 261–267, 2002.

[43] K. Langendoen and G. Halkes. Energy-Efficient Medium Access Control. *Embedded Systems Handbook*, pages 34–1.

[44] Suet Fei Li, Roy Sutton, and Jan M. Rabaey. Low power operating system for heterogeneous wireless communication systems. In *PACT 01*, Barcelona, Spain, 2001.

[45] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[46] B. Lo, S. Thiemjarus, R. King, and G. Yang. Body Sensor Network–A Wireless Sensor Platform for Pervasive Healthcare Monitoring. *The 3rd International Conference on Pervasive Computing*, 2005.

[47] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, 2002.

[48] A. MARTIN. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4) :226–234, 1986.

[49] A. MARTIN. *Programming in VLSI : From communicating processes to delay-insensitive circuits*. Development in Concurrency and Communication. Addison-Wesley University of Texas At Austin Year of Programming Series, 1990.

[50] A.J. MARTIN. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[51] AJ Martin, M. Nystrom, K. Papadantonakis, PI Penzes, P. Prakash, CG Wong, J. Chang, KS Ko, B. Lee, E. Ou, et al. The Lutonium : a sub-nanojoule asynchronous 8051 microcontroller. *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 14–23, 2003.

[52] S.M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. *Proc. ICCAD*, 2 :721–725, 2002.

[53] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[54] R.E. MILLER. *Sequential Circuits and Machines*, volume 2 of *Switching Theory*. John Wiley & Sons, 1965.

[55] NIST. Advanced encryption standard (aes). *Federal Information Processing Standards (FIPS) 197*, 2001.

[56] Graz University of Technology. Aes software module for msp430. `http://jce.iaik.tugraz.at/sic/products/crypto_software_for_microcontrollers`.

[57] Papmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, 2001.

[58] J. Polastre, R. Szewczyk, and D. Culler. Telos : enabling ultra-low power wireless research. *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369, 2005.

[59] J. QUARTANA. *Design of Asynchronous Network on Chip : application to GALS systems*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, 2004.

[60] JM Rabaey, MJ Ammer, JL da Silva Jr, D. Patel, and S. Roundy. PicoRadio supports ad hoc ultra-low power wireless networking. *Computer*, 33(7) :42–48, 2000.

[61] M. RENAUDIN. Asynchronous circuits and systems : a promising design alternative. *Microelectronic Engineering*, 54(1-2) :133–149, 2000.

[62] M. RENAUDIN and J.-B. RIGAUD. Etat de l'art sur la conception des circuits asynchrones : perspectives pour l'intégration des systèmes complexes. Technical report, TIMA, Grenoble, 1998.

[63] J.B. RIGAUD. *Spécification de bibliothèques pour la synthèse de circuits asynchrones*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, 2002.

[64] D. RIOS-ARAMBULA, A. BUHRIG, G. SICARD, and M. RENAUDIN. On the use of feedback systems to dynamically control the supply voltage of low-power circuits. *Journal of Low Power Electronics*, 2 :45–55, 2006.

[65] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. GLONEMO : global and accurate formal models for the analysis of ad-hoc sensor networks. *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, 2006.

[66] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[67] I.E. SUTHERLAND. Micropipelines. *Communication of the ACM*, 32(6), June 1989.

[68] J.T. UDDING. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4) :197–204, 1986.

———

Publications de l'auteur

Revues internationales

- D. Rios, A. Buhrig, G. Sicard and M. Renaudin, "On the use of feedback control to dynamically control the supply voltage of low-power circuits", *Journal of Low Power Electronics (JOLPE)*, Volume 2, number 1, April 2006, pp45-55.
- Article de revue en préparation.

Conférences internationales et Workshops avec comité de lecture

- A. Buhrig, M. Renaudin, D. Barthel, "Asynchronous Architecture for Sensor Network Nodes", *Fourth Mediterranean Ad Hoc Networking Workshop (MedHocNet)*, Porquerolles, France, June 2005.
- D. Rios, A. Buhrig, M. Renaudin, "Power Consumption Reduction Using Dynamic Control of Micro Processor Performance", *15th International Workshop on Integrated Circuit and System Design : Power and Timing Modeling, Optimization and Simulation (PATMOS 2005)*, Leuven, Belgium, September 2005, pp 20-23
- Y. Ammar, A. Buhrig, M. Marzencki, B. Charlot, S. Basrour, K. Matou and M. Renaudin, "Wireless sensor network node with asynchronous architecture and vibration harvesting micro power generator", *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence (SoC-EUSAI'05*, Grenoble, France, 2005, pp 287-292.
- M. Dohler, D. Barthel, S. Aubert, C. Dugas, F. Maraninchi, M. Laurent, A. Buhrig, F. Paugnat, M. Renaudin, A. Duda, M. Heusse, F. Valois, "The ARESA Project : Facilitating Research, Development and Commercialization of WSNs", *Proceedings of Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2007)*, San Diego, USA, June 2007.

Conférences nationales

- D.Rios, A.Buhrig, M.Renaudin, “Asservissement de vitesse pour minimiser la puissance consommée par un processeur”, *5ème journées Faible Tension, Faible Consomation (FTFC’05)*, Paris, mai 2005.
- A. Buhrig, M. Renaudin, “Architecture asynchrone des noeuds de réseaux de capteurs”, *9ème édition des Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM’06)*, Renne, mai 2006.
- A. Buhrig, D. Rios, M. Renaudin, “Systèmes embarqués sans horloge : application aux réseaux de capteurs”, *Colloque EmSoC-Recherche*, Villard de Lans, juin 2006.
- A. Buhrig, M. Renaudin, “Gestion de la consommation des nœuds de réseau de capteurs sans fil”, *10ème édition des Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM’07)*, Lille, mai 2007.
- A.Buhrig, M.Renaudin, “On the use of real-time specifications for reducing power consumption in wireless sensor network”, *6ème journées Faible Tension, Faible Consomation (FTFC’07)*, Paris, mai 2007.
- A. Buhrig, M. Renaudin, “Gestion de la consommation des nœuds de réseau de capteurs sans fil”, *Premier colloque national du GDR System-On-Chip & Sytem-In-Package (SoC-SiP)*, Paris, juin 2007.

OPTIMISATION DE LA CONSOMMATION DES NŒUDS DE RÉSEAUX DE CAPTEURS SANS FIL

Résumé :

Les réseaux de capteurs sans fil (WSN), posent de nombreux défis de conception. Ils doivent capter, traiter, recevoir et retransmettre des données tout en ayant une durée de vie atteignant des dizaines d'années selon les applications sans intervention extérieure. Il est donc nécessaire d'optimiser la consommation d'énergie à tous les niveaux de conception. Ce travail propose de réduire la consommation d'énergie de la partie numérique d'un nœud de WSN. Pour cela, la logique asynchrone réputée pour sa faible consommation, est utilisée. Associée à un partitionnement judicieux, celle-ci permet de s'affranchir du coût de communication logiciel/matériel en utilisant des communications synchrones utilisées intrinsèquement par le matériel asynchrone. Par ailleurs, un moyen efficace de réduire la consommation d'énergie est d'utiliser la technique d'adaptation dynamique de la tension (DVS) du processeur. Pour cela, un coprocesseur est défini pour réguler la tension d'alimentation du processeur en fonction d'une consigne logicielle en vitesse. De plus, il est important de contrôler efficacement la vitesse du processeur. Ce travail propose une méthodologie de simulation permettant d'évaluer l'impact sur la consommation d'énergie des algorithmes d'ordonnancement temps réel de tâches et de gestion du DVS, ainsi que les modèles de tâches s'exécutant sur le système.

Mots clés :

Réseaux de capteurs sans fil, faible consommation, DVS, communications synchrones, circuits asynchrones, temps réel, partitionnement matériel/logiciel.

OPTIMIZATION OF THE ENERGY CONSUMPTION IN WIRELESS SENSOR NETWORK NODES

Abstract :

Wireless sensor networks pose many design challenges. They must collect information coming from the environment, treat gathered data, receive and forward it with a lifetime that must reach tens of years depending on the applications without any external intervention. It is thus necessary to optimize the energy consumption at every design level. This work proposes to reduce the energy consumption of the digital part of a network node thanks to the use of asynchronous logic and synchronous communications between software and hardware. In addition, it is important to dynamically manage consumption by scaling the supply voltages. This aspect is treated on the whole system, at the material level, thanks to the specification of a dedicated coprocessor, at the algorithmic level and real-time software, and at the communication interfaces between the software and the hardware.

Keywords :

Wireless sensor networks, low power, DVS, synchronous communications, asynchronous circuits, real-time, hardware-softawre partitioning.

Thèse préparée au laboratoire TIMA (Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs), INPG, 46 avenue Félix Viallet, 38031, Grenoble Cedex 1, France.

ISBN 978–2–84813–116–0