UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Bahram YARAHMADI »

« Static and dynamic compiler support for intermittently powered computer systems »

Thèse présentée et soutenue à Rennes, le 1er juillet, 2021
Unité de recherche : Institut National de Recherche en Informatique et Automatique (Inria)
Thèse N° :

Rapporteurs avant soutenance :

Abdoulaye GAMATIÉ Directeur de Recherche, CNRS, LIRMM
Fabrice RASTELLO Directeur de Recherche, INRIA

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Olivier SENTIEYS	Professeur, Université de Rennes 1
	Henri-Pierre CHARLES	Directeur de Recherche, CEA
	Maria MENDEZ REAL	Maître de conférence, Polytech Université de Nantes
Dir. de thèse :	Erven ROHOU	Directeur de recherche INRIA

Invité(s) :

Prénom Nom Fonction et établissement d'exercice

Acknowledgement

I would like to express my sincere gratitude to my advisor Dr. Erven ROHOU for giving me the chance to work under his supervision. I would like to thank him for his incredible patience and guidance. He has served as an excellent mentor and advisor with his continuous support and encouragement.

I would like to thank the members of the jury for having agreed to examine my work.

I am extremely thankful to the members of the PACAP team at INRIA for their help, support and kindness. I consider myself truly lucky to have worked with them.

Finally, a special thanks to my parents, family, and friends for their support and for everything they have done for me.

Table of Contents

Abstract

With the advent of Internet of Things (IoT), there is a need to provide energy for a massive number of tiny smart devices without using large, heavy, and high-maintenance batteries. One promising way is to harvest energy from the environment and store it into an energy buffer such as a capacitor which is in charge of supplying energy to the device. However, harvested energy sources are all unstable making the execution of programs intermittent. That is, the program is executed as long as there is available energy in the capacitor, and crashes when it exhausts. As a result, programs with long-running processing time cannot be completed with a single charge of the capacitor. Recently, different software and hardware-based checkpointing strategies have been proposed to make forward progress toward execution for energy harvesting IoT devices. This thesis introduces two different software solutions based on the static and dynamic compilation. The proposed static compiler inserts checkpoints based on statically-computed worst-case energy consumption of code sections. Moreover, it applies classical compiler optimizations in order to decrease the required number of checkpoints at runtime. The proposed dynamic compilation technique shifts checkpoint placement and specialization to the runtime and takes decisions based on the past power failures and execution paths taken before each power failure. Both proposed solutions guarantee making forward progress as well as keeping the memory consistent. Furthermore, they aim to increase portability by not using any hardware feature of the IoT device. In addition, the proposed dynamic compiler approach is transparent to programmers and the proposed static compiler approach limits the burden on programmers to a negligible additional effort.

Chapter 1

Résumé

Nous vivons à l'ère de l'Internet des objets (IoT), où le monde qui nous entoure est composé d'un grand nombre de minuscules objets qui détectent, communiquent et traitent des données dans notre environnement. Pour ces minuscules objets, l'approvisionnement et la consommation d'énergie sont un défi : il n'est pas économiquement viable, ni même physiquement possible de les configurer avec des batteries de grande taille, lourdes et nécessitant beaucoup d'entretien. Récemment, il a été proposé d'utiliser des techniques de collecte d'énergie comme moyen alternatif de fournir de l'énergie sans avoir recours à des batteries. Dans ces techniques, l'énergie est extraite de différentes sources dans l'environnement (par exemple, la lumière du soleil ou le vent) [60, 75] et stockées dans un tampon tel qu'un condensateur. Cependant, l'un des problèmes des sources d'énergie récoltées est qu'elles sont toutes instables. Cette instabilité des sources d'énergie et la faible quantité d'énergie qu'un condensateur peut stocker font que l'exécution des programmes est interrompue par des pannes de courant. Par conséquent, les tâches dont le temps de traitement est long ne peuvent pas être réalisées avec une seule charge du condensateur. Une façon de garantir la progression vers l'achèvement des tâches est de tirer parti de l'idée de prendre des points de contrôle. C'est-à-dire en stockant toutes les données volatiles nécessaires telles que l'état du processeur, la pile de programmes et le tas dans une mémoire persistante avant l'épuisement de l'énergie. Lorsque l'énergie redevient disponible, tout l'état volatil est recopié et le programme peut continuer son exécution.

D'une part, le pointage de l'état volatil du programme dans la mémoire non volatile disponible dans les systèmes embarqués semble prometteur, car un programme peut avoir une exécution intermittente jusqu'à son terme. D'autre part, la prise imprudente de points de contrôle fait soit que le système ne progresse pas, soit qu'il souffre d'une dégradation des performances et de l'énergie. Par exemple, un nombre de points de contrôle inférieur à ce qui est nécessaire, appelé nombre optimal de points de contrôle, fait qu'au moins une partie du code consomme plus d'énergie que la quantité maximale d'énergie dans le condensateur. Par conséquent, la section est exécutée de manière répétée sans aucune

progression. Au contraire, en prenant plus de points de contrôle que le nombre optimal, on gaspille l'énergie du système pour effectuer des travaux inutiles, car prendre un point de contrôle n'est pas sans coût. De plus, Ransford et Lucia [61] ont mis en évidence que le fait de prendre des points de contrôle et de reprendre l'exécution peut entraîner des violations de la correction du programme lorsque celui-ci a des effets de bord, comme la modification des données non volatiles. Par exemple, considérons le cas où un point de contrôle est pris et où le programme lit et modifie certaines données en mémoire non volatile. Si une panne de courant se produit avant d'atteindre le point de contrôle suivant, le système doit revenir au point de contrôle précédent et exécuter à nouveau les mêmes instructions. Cependant, la deuxième fois, les données en mémoire non volatile ne sont pas correctes. Pour faire face au problème de performance et au bogue susmentionné, les chercheurs ont proposé différentes approches de solutions entièrement logicielles [51, 23, 52, 69, 8] à des solutions matérielles/logicielles communes [62, 37, 10, 9, 33] ainsi que des solutions uniquement matérielles [81, 73].

Les solutions matérielles et logicielles communes peuvent prendre des points de contrôle au prix de l'ajout de fonctionnalités matérielles supplémentaires ou de l'utilisation des fonctionnalités matérielles des microcontrôleurs du commerce qui sont conçues à l'origine pour être utilisées par le programmeur (par exemple, les minuteries, les CAN) et non par le logiciel système d'un tiers. Toutefois, les solutions au niveau logiciel peuvent résoudre les problèmes d'exécution intermittente sans utiliser de caractéristiques matérielles spéciales ou dédiées. En conséquence, les solutions logicielles sont plus portables. Dans cette recherche, nous abordons les problèmes susmentionnés au niveau du logiciel.

1.1 Objectifs

Les objectifs de cette thèse sont :

— permettre d'éviter la non-terminaison du programme et avoir une mémoire cohérente ;
— faciliter la programmation du système de collecte d'énergie, soit en fournissant des solutions transparentes, soit en limitant la charge des programmeurs à un effort supplémentaire négligeable ;
— fournir une solution portable ne nécessitant aucun matériel supplémentaire, ni utilisant une fonction matérielle dédiée.

1.2 Solutions proposées

Un compilateur statique peut analyser statiquement la consommation d'énergie d'un programme sur un matériel particulier et placer des points de contrôle dans le graphe de flot de contrôle du programme en conséquence. De plus, inspiré de l'idée utilisée dans les compilateurs dynamiques, un compilateur peut suivre l'exécution du programme pour trouver le point où se produit la panne de courant et placer le point de contrôle au moment de l'exécution en conséquence. Cette thèse propose deux solutions indépendantes de compilateurs statiques et dynamiques.

WCEC : placement des points de contrôle et optimisation du compilateur

Dans ce travail, nous proposons de définir l'emplacement de ces points de contrôle sur la base de la consommation d'énergie des sections de code, calculée statiquement dans le pire des cas. Nous appliquons également des optimisations classiques des compilateurs afin de réduire le nombre de points de contrôle requis au moment de l'exécution. Comme notre méthode est basée sur la consommation d'énergie dans le pire des cas, nous pouvons garantir la cohérence de la mémoire et l'avancement des calculs.

La détermination de l'emplacement des points de contrôle est basée sur le graphe de flot de contrôle (CFG) du programme. Il est similaire au calcul des estimations du pire temps d'exécution (WCET) dans le domaine des systèmes temps réel. L'estimation statique de la consommation d'énergie dans le pire des cas (WCEC) nécessite d'avoir une représentation du programme ainsi qu'un modèle énergétique qui reflète la consommation d'énergie du système. Pour le premier, le CFG du programme représente des structures complexes telles que les boucles, les conditions et les appels de fonction. Pour le second, les modèles énergétiques aux niveaux inférieurs du logiciel tels que l'architecture du jeu d'instructions (ISA) sont plus précis car ils sont plus proches du matériel [28]. Notre mise en œuvre consiste en un composant d'estimation de WCEC fonctionnant au-dessus d'un outil appelé Heptane. Heptane est un outil conçu à l'origine pour calculer le WCET. Il s'agit également d'un algorithme de localisation de points de contrôle. La figure 1.1 montre l'aperçu de notre flux. De plus, comme Heptane ne fonctionne que sur des fichiers binaires exécutables et que notre placement final de point de contrôle se fait au niveau LLVM IR [44], nous avons adopté une cartographie entre Heptane et LLVM IR.

Comme indiqué, l'entrée de la chaîne d'outils proposée est la taille du condensateur et le code C de haut niveau annoté avec des informations liées à la boucle. Il convient de

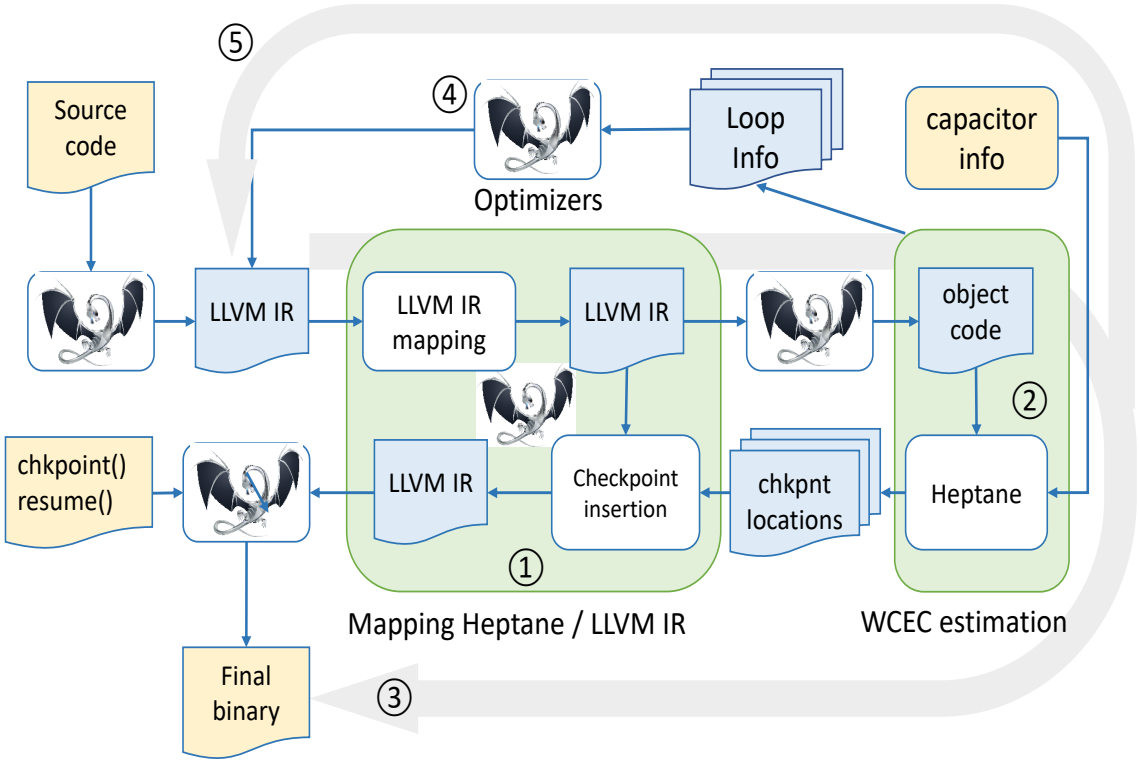


Figure 1.1: Aperçu de notre flux

noter que la spécification des informations liées à la boucle est le seul effort supplémentaire demandé au programmeur. La sortie de notre chaîne d'outils est un code binaire enrichi d'appels de déclenchement de points de contrôle. Chaque déclencheur de point de contrôle est un appel à une bibliothèque d'exécution qui est responsable de la copie de l'état volatile du programme dans la mémoire non volatile. L'overhead et le coût énergétique du contrôle lui-même dépendent fortement de l'architecture sous-jacente. Pour une architecture avec une mémoire non volatile comme mémoire unifiée, le coût du checkpointing est presque constant puisque seuls les registres du CPU doivent être copiés. Cependant, pour les systèmes configurés avec une mémoire volatile comme la SRAM ainsi qu'un type de mémoire non volatile, le coût du contrôle est variable et dépend de l'état du programme comme la taille de la pile et du tas, ainsi que de la quantité de données actives au moment du contrôle. Dans ce dernier cas, nous devons garantir que nous avons suffisamment d'énergie pour effectuer le contrôle dans le pire des cas, et l'emplacement du point de contrôle est important. Dans le pire des cas, le système doit avoir suffisamment d'énergie pour contrôler toute la mémoire volatile du point de contrôle. Notre travail peut fonctionner avec les deux types d'architecture. Cependant, dans ce travail, nous partons du principe qu'il

y a toujours assez d'énergie disponible pour contrôler un nombre constant de registres CPU dans le premier cas ou toute la mémoire volatile dans le second cas, et nous nous concentrons sur le placement des points de contrôle qui garantit l'exactitude et la progression. Nous avons appliqué des optimisations du compilateur pour réduire le nombre de contrôles à l'exécution. Notre objectif ici n'est pas de fournir une étude exhaustive de l'impact de chaque optimisation, mais plutôt de démontrer l'avantage de certaines optimisations pour la gestion énergétique des systèmes à alimentation intermittente. Comme les boucles sont la partie des programmes la plus consommatrice de temps et d'énergie, nous avons choisi des optimisations sur les boucles. De plus, le placement de points de contrôle dans les boucles peut augmenter la consommation d'énergie de l'exécution, ce qui rend essentiel de traiter les boucles avec soin. Il est clair que toutes les optimisations ne peuvent pas être appliquées à toutes les boucles, car les dépendances des données doivent être vérifiées pour garantir la préservation de la sémantique du programme. Dans ce travail, nous avons choisi des points de contrôle conditionnels, le fractionnement de boucle, la fission de boucle et l'échange de boucle. Nos résultats montrent que les optimisations ont la capacité de réduire le nombre de points de contrôle nécessaires, ce qui montre que les optimisations classiques des compilateurs peuvent être exploitées pour réduire le nombre de points de contrôle.

Placement dynamique des points de contrôle basé sur un code auto-modifiant

Dans ce travail, inspiré par l'idée utilisée dans les compilateurs dynamiques, nous avons adopté un cadre comprenant plusieurs passes d'analyse et de transformation du compilateur ainsi qu'un système d'exécution. Notre travail reporte au runtime les décisions finales concernant l'emplacement des points de contrôle. Les décisions qu'il prend au moment de l'exécution sont basées sur le chemin d'exécution que le programme avait pris avant la panne de courant. De cette façon, le système d'exécution apprend des pannes de courant afin de placer les points de contrôle. Notre travail s'appelle SFSG (So Far So Good). Intuitivement, SFSG est basé sur l'observation qu'un programme IoT exécute une série de tâches en continu pendant une longue période de temps et que les flots de contrôle se répètent pendant l'exécution. Par exemple, les petits dispositifs IoT consistent souvent en une boucle infinie qui détecte certaines données de l'environnement, les calcule, les chiffre éventuellement et transmet le résultat. Mais la répétition du flux de contrôle s'applique - à des degrés divers - également à la plupart des applications. C'est la raison d'être des optimisations guidées par profiling, mais aussi des caches d'instructions et des

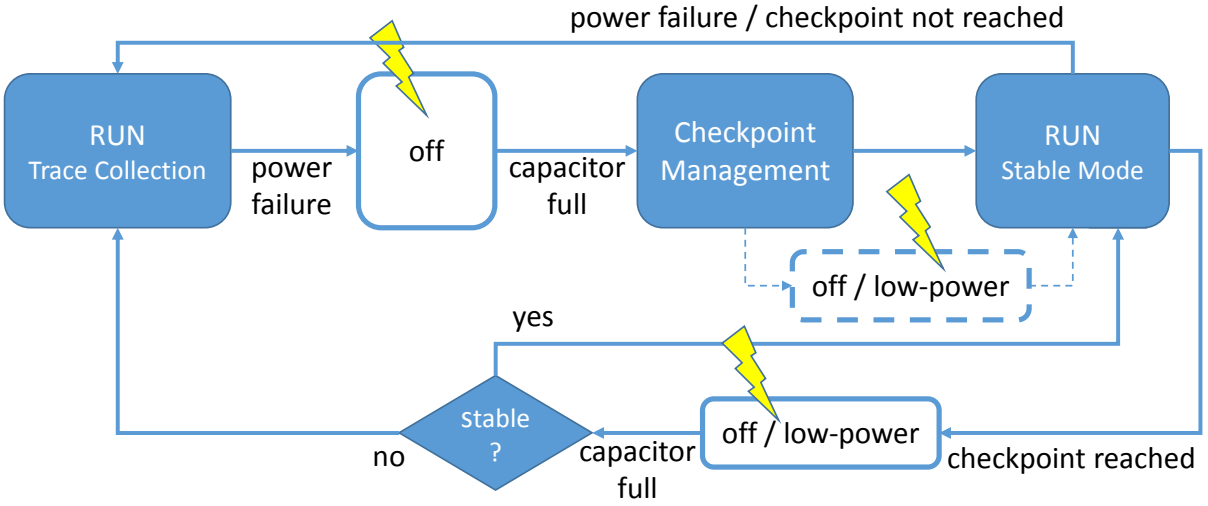


Figure 1.2: Flux des états d'exécution

prédicteurs de branchements. En bref, nous laissons le programme s'exécuter, légèrement modifié pour générer une trace d'exécution en mémoire non volatile, jusqu'à épuisement de l'énergie. Lorsque l'énergie reprend, et que le système redémarre, nous savons, grâce à la trace, où il s'est arrêté. Nous insérons un point de contrôle un peu avant, et nous relançons l'exécution depuis le début. Avec un peu de chance, l'exécution atteint le point de contrôle. La ré-exécution reprendra à partir de ce nouveau point, garantissant ainsi une progression vers l'avant. Si le point de contrôle n'est pas atteint, nous l'insérons un peu plus tôt dans le CFG et reprenons l'exécution. Au point de contrôle, nous attendons que le condensateur soit complètement rechargé. Notre système paie une pénalité lorsqu'un chemin de code est vu pour la première fois. Un prix plus élevé est payé si les chemins de contrôle changent souvent. Cependant, même dans ce dernier cas, notre système reste correct et progresse dans son exécution : nous ne sommes pas limités à une classe particulière de programmes.

Le système d'exécution (la partie dynamique) est responsable de l'orchestration globale des différentes étapes en jeu. Il est illustré par une machine à états finis dans la figure 1.2. La phase statique transforme les CFG, extrait les propriétés du programme. Elle permet également au programme d'ajouter des points de contrôle à différents endroits. La figure 1.3 montre comment `main()` et `Foo()` Les CFGs sont transformés. La combinaison de la partie statique et de la partie dynamique peut fournir un mécanisme permettant de collecter des traces d'exécution lorsque le programme est en cours d'exécution. La trace peut fournir des informations sur le moment où la défaillance s'est produite. Le système d'exécution peut placer un point de contrôle dans le code sur la base des informations de

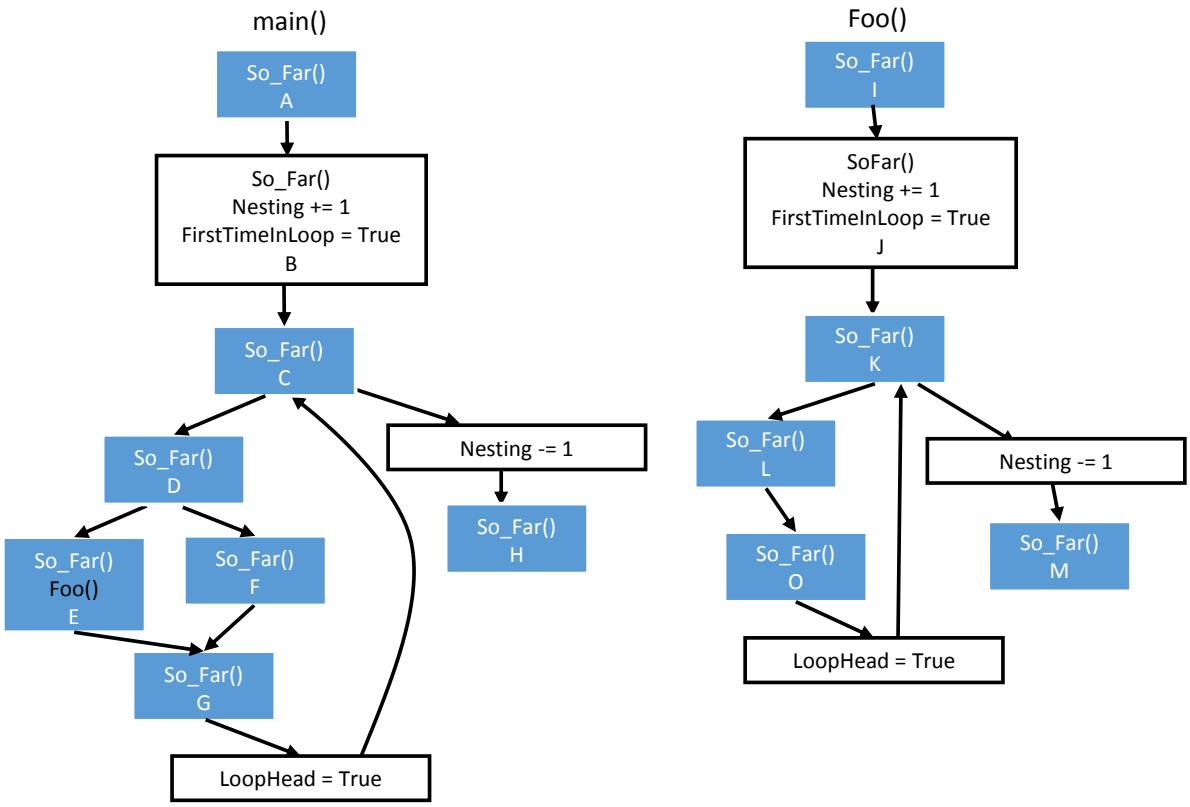


Figure 1.3: Transformations statiques des CFGs

la trace. En outre, SFSG découvre automatiquement les parties du code qui consomment plus d'énergie que le condensateur du microcontrôleur ne peut en fournir et insère des points de contrôle pour garantir la terminaison. Nous avons évalué SFSG sur un système à base de TI MSP430, avec différents types de bechmarks ainsi que différents intervalles d'interruptions pour simuler des pannes de courant. Dans tous les cas, SFSG pourrait permettre de progresser sans devoir s'interrompre.

Chapter 2

Introduction

2.1 Context

A new kind of computing systems has recently emerged, loosely referred to as Internet of Things (IoT) devices. While these devices broadly differ in purposes and form factors, many are tiny and meant to be deployed in hard-to-reach locations to monitor various properties (temperature of remote places, solidity of bridges...). For this reason, they are not equipped with batteries. Instead, they harvest energy from their environment to fill an energy buffer. They run until the energy buffer depletes and stop. The energy buffer is responsible for providing the energy of the device. When it runs out of energy, the device stops executing tasks and waits until the energy level of the energy buffer reaches a threshold. Figure 2.1 shows the schematic of an energy harvesting device. It contains a microcontroller (MCU) having peripherals such as sensors and radio to interact with the world. Also, it includes a capacitor acting like an energy buffer for the device. The harvester is another entity of an energy harvesting device that is in charge of extracting energy from the environment and storing it in the energy buffer.

There are various energy harvesting sources. Radio frequency (RF) [84], solar [60], thermal [50], kinetic [11] and vibration [74] are examples of energy harvesting sources that are accessible depending on the environment. However, one problem with these sources of energy is that they are all unstable. This instability of energy sources and the small amount of energy a capacitor can store make the execution of programs interrupted by power failures. As a result, tasks with long running processing time cannot be completed with a single charge of the capacitor, because, after each power failure, the execution starts from the beginning. One way to guarantee forward progress to the completion of tasks is by leveraging the idea of taking checkpoints used in distributed systems. In a distributed system checkpointing is a highly used technique to provide fault tolerance and rollback recovery by saving a snapshot of necessary information periodically. These information are stored in a stable storage such as a disk. Therefore, when a system crashes due to a

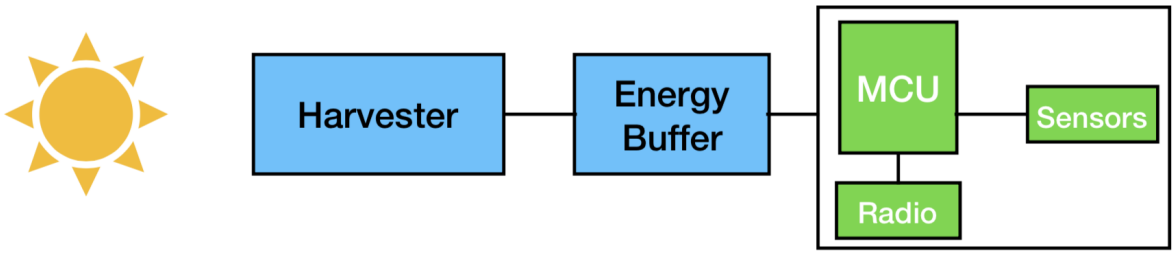


Figure 2.1: The schematic of an energy harvesting system

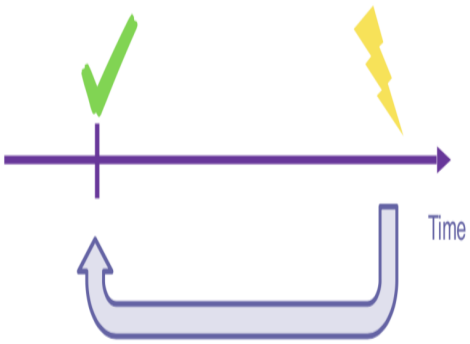


Figure 2.2: Rollback to a checkpoint location

failure, it can be re-stared and roll back to the point that the checkpoint had been taken and resume its execution [42]. This technique can be used in an energy harvesting device that is subjected to power failures because of energy depletion. Herein, a checkpoint can be taken by storing all necessary volatile data such as processor state, program stack, and global variables into a persistent memory before energy depletion. In this way, when the energy becomes available again after the power failure, all the volatile state will be copied back and the program can continue its execution. Figure 2.2 shows rolling back to a location where a checkpoint has been taken when a power failure happens during the execution time. It is worth noting that the MCU of the energy harvesting device includes a type of persistent memory to store non-volatile data such as the text (code segment) of the program that can be used to store checkpoint data as well. As a result of checkpointing and the abundant power failures, programs running on an energy harvesting device are executed intermittently. That is, a program is executed as long as there is available energy in the capacitor, and crashes when it exhausts. It waits until the energy will be available again to resume the execution. Figure 2.3 shows that an energy harvesting device is switching between two cycles during its lifetime. When the voltage is at `Vdeath`, the

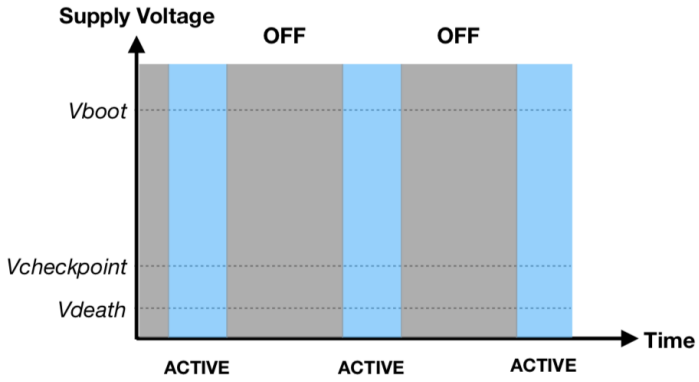


Figure 2.3: Various starges and cycles of an energy harvesting device

device is off or it does not have energy to proceed with the execution of the program. In this stage, the device is extracting the energy from the environment. When the voltage of the capacitor reaches a specific threshold (`Vboot`), the device starts a new life-cycle and resumes the execution of the program that had been interrupted. Taking checkpoints also consumes energy and the cycles of the system that must be taken into consideration. For that, in some energy harvesting devices, a new voltage threshold for specifying the point of checkpoint must be set. For instance, in Figure 2.3, when the voltage is at `Vcheckpoint`, the system still has the energy to take a checkpoint.

There are some issues related to this model of execution which we discuss in this Chapter.

2.2 Problem definition

Writing programs that are executed intermittently is challenging as the program execution might be interrupted frequently because of the unpredictable nature of energy sources. Even with the existence of the checkpointing mechanism, it is not always guaranteed that the program will make forward progress to the completion. The programmer must be sure that the program will complete its task when it is running in the deployment environment. In addition, another unique bug of energy harvesting devices that use checkpointing techniques is related to the consistency issue of persistent memories which may lead to program correctness violation. In this thesis, we define the former problem as `Lack of forward progress` and the latter as `Inconsistent memory`.

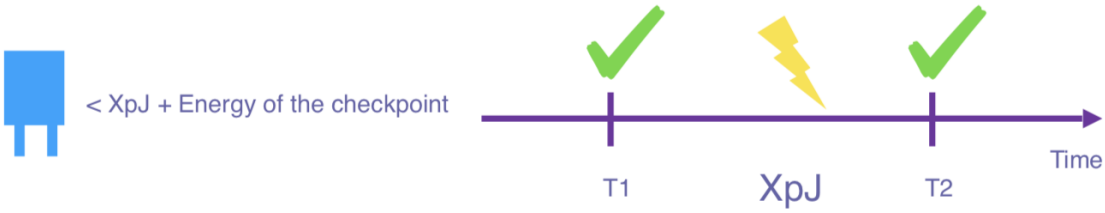


Figure 2.4: Lack of forward progress

2.2.1 Lack of forward progress

As checkpoints have overhead, they can affect the overall performance of the system in terms of execution time and energy consumption. Taking checkpoints too early wastes the energy and cycles of the system for useless work. On the other hand, taking checkpoints too late, may put the system at risk of unhandled power failure. In the worst case, the program faces non-termination when the code between two checkpoints requires more energy than the capacitor can provide. Consequently, the program cannot reach the next checkpoint and will face power failure each time it tries to re-execute the code from the first checkpoint location. This is one of the novel bugs introduced by intermittent execution. Figure 2.4 shows that there are two checkpoints in the program (in times T1 and T2). For reaching the second checkpoint from the first checkpoint, `XpJ` of energy is needed. Plus, the energy consumption of the checkpoint itself must be taken into account. However, the capacitor cannot provide this amount of energy.

2.2.2 Inconsistent memory

A program under intermittent execution may face a memory inconsistency issue when it uses both volatile and non-volatile memory of the system for allocating variables and data, and they are involved in an instruction sequence that performs Write-After-Read (WAR) [69]. In this scenario, re-executing instructions due to the checkpointing and resuming may lead to program correctness violations. For example, consider a case that a checkpoint is taken and then the program reads and modifies some data in non-volatile memory. If a power failure happens before reaching the following checkpoint, the system must roll back to the previous checkpoint and re-execute the same instructions. However, the second time, the data in non-volatile memory are not correct. Figure 2.5 illustrates an example with two pieces of code containing checkpoint trigger calls. In this example, the variable `X` is allocated in non-volatile memory by the programmer. The value of variable

```
NV int X;          NV int X;
X = 0;             _checkpoint();
_checkpoint();     X = 0;
X++;               X++;
// Power failure   // Power failure
send(X);           send(X);
  (a)                (b)
```

Figure 2.5: Memory inconsistency problem

`X` at the end of the continuous execution (execution without power failure) of these code sections will be 1. However, if after statement `X++` a power failure happens, in `(a)`, the system rollbacks to the last checkpoint location which is a line just before the statement and re-execute the statement again. As a result, the value of variable `X` that is going to be sent at the end will be 2. The reason behind this is that the statement `X++` is WAR to the non-volatile variable `X`. In `(b)`, the value at the end will be still 1 as the checkpoint trigger call is before the statement `X = 0` and the sequence of statements `X = 0` and `X++` is *idempotent*. A piece of code is *idempotent* if repeated subsequent invocations do not modify the state of the machine. In (b), the aforementioned sequence performs WARAW and the write before the first read makes the sequence idempotent.

2.3 Contribution

To cope with the aforementioned performance issues and bugs, researchers have proposed different approaches from fully software solutions [51, 23, 52, 79, 69, 8] to joint hardware/software solutions [62, 37, 10, 9, 33] as well as hardware-only solutions [81, 73]. Hardware based and joint hardware/software solutions can take checkpoints at the cost of adding additional hardware features or using hardware features of the commercial MCUs which are originally designed to be utilized by the programmer (e.g., timers, analog-to-digital converters) and not by the third-party system software. However, software level solutions can cope with the problems of intermittent execution without using any special or dedicated hardware feature. As a result, software solutions are more portable. In this research, we address the aforementioned problems in software level.

2.3.1 Objectives

The objectives of this thesis are :

— to provide forward progress without facing non-termination as well as to have a

consistent memory;
— to ease programming of energy harvesting system either by providing transparent solutions or by limiting the burden on programmers to a negligible additional effort;
— to provide a portable solution by not requiring any extra hardware or using any dedicated hardware feature.

2.3.2 Proposed solutions

A static compiler can statically analyze the energy consumption of a program on a particular hardware and place checkpoints trigger calls into the control flow graph (CFG) of the program accordingly. Also, inspired by the idea used in dynamic compilers, a compiler can track the execution of the program at runtime to find the point that power failure happens and place the checkpoint at runtime accordingly. This thesis proposes two independent static and dynamic compiler solutions :

Static Checkpoint Placement

In this static compiler approach, we propose to define these checkpoint locations based on statically-computed worst-case energy consumption of code sections. We also apply classical compiler optimizations in order to decrease the required number of checkpoints at runtime. This work was presented in **International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS 2020)** and will be explained in detail in Chapter 4.

Dynamic Checkpoint Placement

In this work, inspired by the idea used in dynamic compilers, we propose SFSG: a dynamic checkpoint placement and specialization strategy which delays checkpoint placement and specialization to the runtime and takes decisions based on the past power failures and execution paths taken before each power failure. The goal of SFSG is to provide forward progress and to avoid facing non-termination without using hardware features or programmer intervention. This work was presented in **International Workshop on Software and Compilers for Embedded Systems (SCOPES 2021)** and will be presented in Chapter 5.

2.4 Organization of thesis

The reminder of this thesis is organized as follows. Chapter 3 reviews the state of the art in intermittent execution. It also provides background information needed for better understanding this thesis work. Chapter 4 presents our static checkpoint placement strategy for intermittent execution. Chapter 5 presents our dynamic checkpoint placement strategy. Finally, Chapter 6 presents the conclusion of this thesis as well as possible future work.

Chapter 3

Background

3.1 Introduction

We live in the era of Internet of Things (IoT) where the world around us is surrounded with a large number of tiny objects sensing, communicating and processing data in our environment. For these tiny objects, energy provision and consumption are challenging: it is not economically viable, or even physically possible to configure them with large, heavy, and high maintenance batteries. Recently, using energy harvesting techniques as an alternative way to supply energy without resorting to batteries has been proposed. In these techniques, energy is extracted from different sources in the environment (e.g, sun light or wind) [60, 75] and stored in a buffer such as a capacitor. However, one problem with harvested energy sources is that they are all unstable [40, 83]. This instability of energy sources and the small amount of energy a capacitor can store make the execution of programs interrupted by power failures. As a result, tasks with long running processing time cannot be completed with a single charge of the capacitor. One way to guarantee forward progress to completion of tasks is by leveraging the idea of taking checkpoints. That is, storing all necessary volatile data such as processor state, program stack and global variables into a persistent memory before energy depletion. When the energy becomes available again, all the volatile state will be copied back and the program can continue its execution. On one hand, checkpointing volatile state of the program into the non-volatile memory available in embedded systems seems to be promising, as a program can have intermittent execution to completion. On the other hand, incautious taking of checkpoints either makes the system not to have forward progress or to suffer from performance and energy degradation. For instance, fewer checkpoints than what is needed, called the optimal number of checkpoints, causes at least a section of code to consume more energy than the maximum amount of energy in the capacitor. As a result, it makes the section to be executed repeatedly without any forward progress. This is a unique bug in intermittent computation jargon which is also called facing non-termination. On the contrary, taking

more checkpoints that the optimal number, wastes the system energy for doing unnecessary work, since taking checkpoint is not without cost. Also, Ransford and Lucia [61] pinpointed that checkpointing and resuming execution may lead to program correctness violations when the program performs side-effects, such as changing non-volatile data. For example, consider a case that a checkpoint is taken and then the program reads and modifies some data in non-volatile memory. If a power failure happens before reaching the following checkpoint, the system must rollback to the previous checkpoint and re-execute the same instructions. However, the second time, the data in non-volatile memory are not correct.

This chapter overviews the technologies and solutions around energy harvesting systems as well as fundamental methods that are used in this thesis. Section 3.2 outlines technologies used in energy harvesting systems including memory technologies and architecture models. Section 3.3 briefly overviews the techniques for estimating the energy consumption of programs and discusses the advantages and discusses advantages and disadvantages of each work. Section 3.4 briefly introduces the tool the we extensively used to prototype the proposed static checkpoint placer. Section 3.5 reviews some of the existing solutions to cope with the problems related to intermittent execution of programs.

3.2 Intermittent execution

As mentioned earlier, with the help of checkpoints, a program can survive power failures and resume its execution. The checkpoint overhead and the amount of checkpoint data are highly dependent on the architecture model and the type of persistent memory used in the device. Also, depending on the architecture of the device and how checkpoint data are stored, the system may suffer aforementioned memory inconsistency problem. In this Section, we briefly introduce persistent memory technologies that can be used in an energy harvesting device. Also, we discuss two architecture models previously presented for energy harvesting systems.

3.2.1 Persistent memory technologies

Persistent memories can preserve data when the device is off. One of the most known types of persistent memories is Flash. This non-volatile memory can store checkpoint data in an energy harvesting device. However, a Flash write operation can only switch a mem-

ory cell from 1 to 0. Switching a memory cell from 0 to 1 requires erasing a large block of memory. As a result, the process of writing to Flash consumes a significant amount of energy and power resulting in the performance degradation of the whole system. Recently, with the advent of new byte-addressable non-volatile memory technologies (e.g., FRAM [1], MRAM [2]), the process of writing has been enhanced. These memories are superior to Flash memory in terms of power consumption and latency [15]. As a result, the process of taking checkpoints in intermittent computation is more efficient with these memories and outperforms systems configured with Flash memories [35]. Non-volatile memory technologies are different in terms of endurance, capacity, read/write asymmetry, energy consumption and latency. The proposed solutions in this thesis are expected to be compatible with any type of non-volatile memory.

3.2.2 Architecture models

The de facto standard architecture model for intermittent computation is a hybrid model where SRAM is used as volatile memory and a type of non-volatile memory such as FRAM [25] or Flash as the persistent memory. In this model, the NVM is used for program text and backup data. At the time of checkpoint, the volatile state of the program in SRAM such as program stack and global variables as well as CPU registers must be copied to NVM. In this model, the overhead of checkpoint is variable and dependent on, such as the size of stack, as well as the amount of live data at the time of checkpointing. The alternative model for hybrid model is an architecture with fully non-volatile memories. In this model, NVM is used as a unified memory which means that the program's data and stack are also stored in NVM. This can decrease the amount of data that is needed to be backed up at the time of taking checkpoint as only CPU registers are needed to be backed up. However, the performance of the whole execution of a program may suffer because of the NVM's higher access latency and energy consumption in comparison to SRAM [36].

3.3 Energy consumption estimation

Estimating energy consumption of energy harvesting systems is necessary, as one way to provide solutions to problems related to intermittent computation is by having a good

1. Ferroelectric RAM
2. Magnetoresistive RAM

understanding of the energy consumption of the micro-controller (MCU). As we will see in the next chapter, our static checkpoint placement strategy is based on energy consumption estimation. Moreover, to estimate the energy consumption, an energy model is needed. This section reviews previous work on energy consumption.

There are two main approaches to determine the energy consumption of a program on a particular hardware: measurement based techniques and solutions based on static analysis and estimation. Although approaches based on physically measuring the energy consumption of the device are likely to determine the most accurate energy consumption, they require having knowledge and expertise of using sophisticated equipment which for software people is hard and painful. This process can be worst when some MCUs do not provide hardware components for energy measurement [27]. Also, as we will see, measurement based techniques cannot guarantee a safe upper bound estimation of the energy consumption. On the other hand, energy estimation techniques based on static analysis requires to have an energy model defining the energy costs. These energy costs or models can be provided by the hardware manufacturer. However, for most commercial processors these models are not available. As a result, people resort to measurement techniques to construct the energy model.

Estimating energy consumption and constructing energy model have been explored in various levels of the computer system stack. For instance, energy consumption can be estimated in the instruction level architecture (ISA) level by estimating the energy consumption of each instruction. The first instruction level power analysis was proposed based on current drawn of the processor while it executes certain instructions repeatedly for measuring the base energy cost of each instruction together with inter-instruction effects such as circuit state overhead [67, 68]. Inspired by Tiwari work [67], people proposed an energy model for a simple deterministic multi-threaded architecture [41]. Also, Chakrabarti et al. [20] present an instruction level power analysis technique based on gate level power estimation requiring access to the gate level or at least RTL level description.

Above the ISA level, researchers have shown the feasibility of the energy estimation in compiler intermediate representation (IR) level. Grech et al. [29] have developed a static analyzer working on the LLVM IR [44]. It estimates the energy consumption based on the extracted cost relations from the program. These cost relations are recursive equations that are extracted from a program, representing the cost of running the program in terms of its input [29, 6].

A few researchers based on the idea from classical worst-case execution time analy-

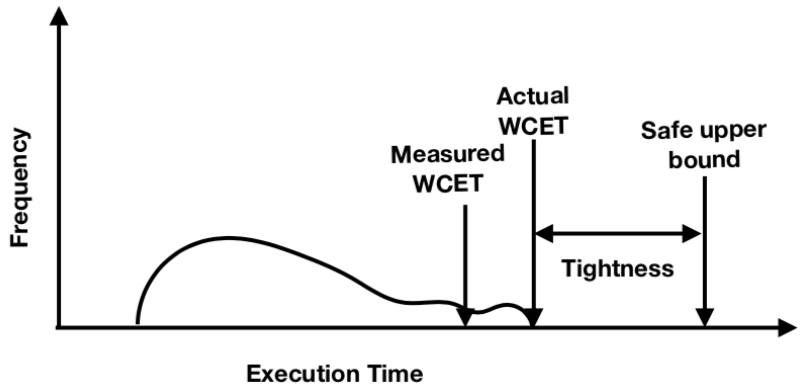


Figure 3.1: Safety and tightness

sis (WCET) [76] have proposed worst-case energy consumption (WCEC) analysis. They tried to benefit from the considerable research on WCET to compute WCEC, as the terminologies used in WCET, can also be applied to WCEC analysis (see Section 3.4). To the best of our knowledge, the work proposed by Jayaseela et al. [38] was the first work to compute worst-case energy consumption. It is stated in the paper that because of the dynamic energy due to switching activity in the circuit, the path corresponding to the WCET of a program may not coincide with the path consuming maximum energy [38]. As a result, using WCET to compute WCEC is not always appropriate. However, they broadly used similar methodologies used in WCET estimation in their work to estimate WCEC. *0g* [71] is another tool for estimating WCEC. If an accurate energy model for the target architecture is available, it computes an upper bound for WCEC by statically analysing the program code. Otherwise, it uses techniques such as genetic algorithm besides measurement techniques to construct an energy model for the target architecture. Moreover, as the energy model may not be available for commercial off-the-shelf hardware platforms and it is an important property for both WCEC and WCET, people have proposed [65, 64] automatic benchmark generators along with measurement techniques to find an appropriate input that can result to WCEC or WCET. There are some issues which are specific to the energy consumption and cannot be inspired by execution time approaches. For instance, external features of a device significantly contribute to the energy consumption. Unlike *0g* and other WCEC tools, *sysWCEC*[70] takes into account the energy consumption of peripherals. Pallister et al. [59] explored the contribution of the operand data on energy consumption. They have showed that unlike execution time, energy is data dependent. Morse et al. [57] proved that determining exact worst-case energy

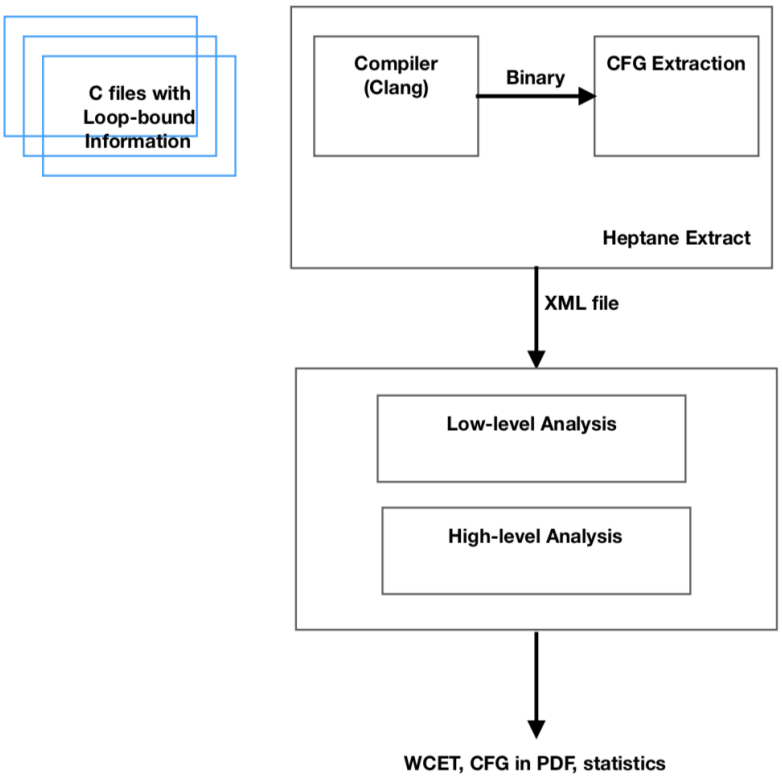


Figure 3.2: The schematic of a WCET tool

consumption with the high level of accuracy in terms of tightness is NP-hard. Therefore, providing a safe upper-bound comes with over estimation proportional to data-dependent energy.

3.4 Worst-case execution time and Heptane

In static WCET analysis the goal is to have a **safe** as well as a **tight** estimation on execution time of a program executing on a hardware. **Safety** means that the actual execution time must be less than, or equal to, the estimated upper-bound, regardless of program input. **Tightness** means that the estimation must be as close as possible to actual WCET. Figure 3.1 illustrates these two important properties of the WCET. Thanks to the extensive research in the WCET analysis, there are commercial and open source static WCET tools [1, 2, 31, 3, 4, 63]. Figure 3.2 shows the schematic of a WCET analysis tool called Heptane [31]. Heptane's functionality is divided into two separated

```
#include "annot.h"
int i;
for (i = 0; i < 10; i++) {
   ANNOT_MAXITER (10)
   . . .
}
```

Figure 3.3: A loop annotated with loop-bound information

components: `Heptane Extract` and `Heptane Analysis`.

`Heptane Extract` generates control flow graphs (CFGs) from source code files written in C or assembly. First, it calls the compiler and linker to generate the binary code. Then, it splits the binary code into basic blocks. A basic block is a sequence of instructions with a single entry and a single exit point. A CFG is a directed graph where each node is a basic block and each edge represent a possible control transfer between two basic blocks. The generated CFG is stored in an XML file inorder to be used by `Heptane Analysis`. Finally, `Heptane Extract` identifies loops and adds loop-bound information provided in the source code into the same XML file. The loop-bound information is one of the constraints of Heptane and the programmer is responsible to provide it for each loop via annotations. Figure 3.3 shows a simple loop that is annotated with loop-bound information. It is worth mentioning that Heptane does not also have support for indirect jumps, jump tables, and pointers to function.

`Heptane Analysis` generally applies several types of analysis on a program. These analyses extract information require to compute WCET from the program. They are applied on various levels of program representation such as object code or source code. The two major analyses are known as high-level analysis or path analysis and low-level analysis. The low-level analysis accounts for low-level features of the processor based on the cost model. `Heptane Analysis` also supports cache analysis, data address analysis, and pipeline analysis. The high-level analysis determines the longest execution paths among all possible flows in a program. It uses the most prevalent technique for high level analysis to compute WCET called Implicit Path Enumeration Technique (IPET) [48]. It is based on Integer Linear Programming (ILP) formulation of the WCET calculation problem. The output of `Heptane Analysis` is one scalar number considered as WCET of the whole program. Also, `Heptane Analysis` can produce statistics about cache as well the CFG of the program in pdf format.

3.5 Checkpointing and program resumption strategies

As mentioned earlier, as a result of unstable energy sources providing the energy of the MCU, a program execution may suffer from abundant power failures. These frequent power failures are unexceptional and unavoidable. Therefore, providing a program resumption strategy is obligatory to make the program continue its execution to completion. Checkpoiting program state into the non-volatile memory is a resumption strategy to make programs survive from failures. However, as it is mentioned before, injudicious checkpointing may rise performance and correctness issues. In this section we review proposed checkpointing solutions and other program resumption strategies used in intermittently powered system.

Researchers have proposed software-only, joint hardware/software as well as fully hardware solutions for making forward progress as well as program correctness in energy harvesting systems. In this section, we will discuss them briefly. Software-only approaches resort to pure software such as a compiler and they do not use any special or dedicated hardware feature of the MCU. Joint hardware/software solutions make use of both hardware features of the MCU (e.g., peripherals of the MCU such as a timer or the ADC of the system) and system software (e.g., compilers, linkers). Hardware solutions try to overcome the problems by modifying the hardware or in the extreme case they come up with a new hardware. It should be noted that this categorization is not strict and depends on the definition of the proposed solution and the context.

Joint hardware/software solutions

To the best of our knowledge, Mementos [62] was the first solution that brought the old concept of checkpointing in large scale and high-performance computing (HPC) to energy harvesting MCUs and made the programs continue their execution across power failures. At compile-time, it instruments trigger points at different program locations such as loop-latches (aka tails of back-edges), and function returns. These trigger points are calls to a function that estimates the available energy at run-time by comparing the capacitor's voltage with a predefined threshold with the help of an analog-to-digital converter (ADC). If the voltage is below the threshold, Mementos checkpoints volatile state of the system onto non-volatile memory. Otherwise, the system continues its normal execution. A driving principle of Mementos was to "reason minimally about energy at

compile time, maximally at run time", because even expert programmers are not reliable when reasoning about energy. However, reading the capacitor voltage with ADC regularly consumes a significant amount of energy. The first version of Mementos uses Flash as the persistent memory which has high write energy overhead.

A few work such as QuickRecall [37], Hibernus [10] and Hibernus++[9] prefer to do on-demand checkpointing (aka Just-In-Time checkpointing [54]) as well as using FRAM as the persistent memory. They monitor the voltage of the capacitor of the system by using hardware support and like Mementos when the voltage is below a threshold, they backup the volatile state. By specifying a proper threshold, they perform efficiently. However, the threshold for each program is different as each program has its own data-size and memory footprints. As a result, finding the threshold is challenging. One of these works [37] uses FRAM as a unified memory which means that programs data and stack are also stored in FRAM. This can decrease the amount of data that is needed to be backed up at the time of checkpointing as only CPU registers are needed to be backed up. However, the performance of the whole execution of a program may suffer because of the FRAM's higher access latency in comparison to SRAM. Jayakumar et al. [36] proposed a mapping technique that maps different program sections in a hybrid FRAM-SRAM MCU aiming to minimize the overall energy consumption. Flexicheck [66] is another software/hardware solution trying to predict future energy profiles based on specific knowledge of the unit that excites the harvester. There are a few works supporting the idea of runtime checkpointing adaptivity by utilizing the timer of the MCU and relying on the timing [53, 21].

Software solutions

Software solutions rely on compilers and/or programmers. A type of software solution that includes modification in the programming language as well as additional responsibilities for the programmer is known as task-based programming models [51, 23, 52]. In a task-based programming model, the programmer is responsible for dividing the code into atomic sections called tasks. These tasks can be re-executed without having side effects on the non-volatile memory. Therefore, the main goal of these models is to cope with the memory consistency issue when the program is being executed intermittently and has allocated data in non-volatile memory. These models apply different methods to take checkpoints and guarantee memory consistency. Dino [51] ensures about non-volatile data consistency with a mechanism called data versioning. That is, making a volatile copy of non-volatile data that are potentially inconsistent. Chain [23] uses a channel-based

memory model. In this model data are send and receive per each pair of tasks. Task's inputs and outputs are stored in distinct memory locations ensuring memory consistency. Alpaca [52] uses automatic privatization. It detects shared variables between tasks and copies them into a private buffer of each task. At the end of the task, modified variables are committed to the main memory. However, in these models, the programmer must be sure that a task's energy consumption does not exceed the maximum available energy in capacitor. Otherwise, the system would face the forward progress problem known as non-termination and would execute the same task repeatedly. To make sure that the application have forward progress, the programmer can act conservatively and place more task boundaries into the code results in wasting more time and energy. Coala [55] is another task based programming model trying to eliminate the forward progress problem in a more efficient way by resorting to the timer of the system.

Ratchet [69] inserts checkpoints at compile-time. It exploits the notion of idempotency for creating restartable code sections. It places checkpoints at idempotent region boundaries. However, because of limitations in alias analysis, the number of checkpoints might be more than needed. Ratchet only works with systems with one unified non-volatile memory. For ensuring forward progress without facing non-termination, it resorts to the programmer and the watchdog timer of the system [3].

A few prior work [13, 24] also consider checkpoint placement by estimating energy. However, at some point in their work, they estimate energy by profiling or measurement techniques or they did not insert checkpoints based on WCEC. As a result, in both cases, their approaches are not safe.

Baghsorkhi et al. [8] proposed undo/redo logs [56, 77] to record non-volatile/volatile data during the execution of the program. They also partitioned the program into regions. At the end of each region, there is a commit point where all recorded redo logs become permanent and undo logs are discarded. However, if a power failure happens during the execution, undo logs are used to recover the old values of non-volatile data and redo logs are discarded. They used dynamic instruction counts as the specifier of boundaries between aforementioned regions which is not safe and the system might face non-termination. Wägemann et al. [72] proposed checkpointing based on WCEC that is very similar to the static compiler presented by this thesis. The work is a runtime kernel which schedules tasks based on the estimated WCEC.

3. In this research we consider Ratchet and Coala as software approaches.

Hardware solutions

To facilitate the process of checkpointing, the community has proposed non-volatile processors (NVP) [81, 73] which are considered as fully hardware approaches. In these architectures all memories in the MCU from main memory to register files are non-volatile. Researchers have proposed software techniques to improve the performance and to ensure the correctness of these specialized processors. For instance, a loop tiling technique has been proposed to reduce the overhead of checkpoints [45]. Also, another software technique has been proposed by Zhao et al. [82] to determine efficient backup positions in these processors. A consistency-aware checkpointing algorithm has been designed to ensure correctness by eliminating the memory inconsistency issue [78].

There are also other hardware approaches [33, 49] trying to cope with the problems in intermittent execution of the programs by modifying hardware. Freezer [58] is a specialized backup/restore controller trying to reduce the overhead of taking checkpoints.

Discussion

It is empirically proved that on demand checkpointing techniques such as QuickRecall, regardless of the difficulty of finding a proper threshold, are efficient in terms of the number of checkpoints as they checkpoint just before power failures [54]. However, they excessively utilize the hardware features of the the device. Generally both hardware based and joint hardware/software solutions can take checkpoints at the cost of adding additional hardware features or using hardware features of the commercial MCUs which are originally designed to be utilized by the programmer (e.g., timers, ADCs) and not by the third party system software. In conclusion to the fully hardware and joint hardware and software solution, using additional hardware features or existing not only increases the energy consumption of the system, but also decreases the portability.

On the other hand, task based software solutions are not transparent to the programmer. The programmer must be aware of the new programming paradigms as well as new libraries and keywords added to the programming language. Also, reasoning about the number and the size of tasks is painful and error-prone for programmers. The burden will be worst when it comes to changing the code or some features of the hardware such as capacitors as the programmer must reconstruct the whole process again.

The proposed work in this thesis are considered as fully software solutions. Both static and dynamic techniques presented in this thesis aim not to use any hardware features of

the MCU and be transparent to the programmer as much as possible. Moreover, they guarantee making forward progress toward execution without facing non-termination as well as keeping the memory consistent.

Our static checkpoint placer is based on WCEC. It guarantees both forward progress and memory consistency. As it is mentioned before, the only checkpoint placer based on WCEC is proposed by Wägemann et al. [72]. Our work differs from it in a way that it is based on the WCEC of program sections and the control-flow graph of the program. Also, our work applies classical compiler optimization in order to decrease the number of checkpoints. To the best of our knowledge, our proposed dynamic compiler approach is the first to bring dynamic compilation to intermittent computation.

Chapter 4

WCEC-based Checkpoint Placement and Compiler Optimizations

4.1 Introduction

This chapter presents the first proposed approach by this thesis to guarantee forward progress without non-termination and to keep memory consistent in energy harvesting systems. The contribution of this work is the following:

— we propose a static, automatic, compiler-based technique for insertion of checkpoints that guarantees correctness and termination of a program on an intermittently powered system;
— we leverage compiler optimizations to reduce the number of checkpoints;
— we limit the burden on programmers to a negligible additional effort;
— we provide a portable solution without requiring any extra hardware support.

The organization of this chapter is as follows. Section 4.2 presents our proposed method with the required background used to adopt it. Section 4.3 evaluates the proposed method. And finally, Section 4.4 concludes the work.

4.2 WCEC-Aware checkpoint placement

4.2.1 Technical background

Determining checkpoint locations is based on the control flow graph (CFG) of the program. It is similar to the computation of worst-case execution time (WCET) estimates in the real-time domain [76].

The goal is to have a safe as well as tight estimation of the energy consumption of

a program executing on the hardware. Safety means that the actual consumption must be less than, or equal to, the estimate, regardless of program input and dynamic events. Tightness means that the estimation must be as close as possible to actual WCEC. Herein, the safety property of WCEC guarantees forward progress and program correctness.

For the forward progress, safety guarantees that the energy consumption for reaching the next checkpoint is less than or equal to the energy a capacitor can provide, since checkpoints are placed based on WCEC with the distance of capacitor's maximum energy. We restrict the system to resume execution after a checkpoint only when the capacitor is full (it may enter a low-power mode for better efficiency). This way, we ensure that when the system wakes up from a checkpoint, it reaches the next one, where it waits for the capacitor the recharge.This property also helps coping with the aforementioned memory consistency issue: as re-execution do not occur, we avoid problems related to memory inconsistency or replaying side effects [61]. With the exception of the checkpoints and resuming code, the application follows its normal control flow.

Also, herein, the tightness of WCEC relates to the number of checkpoints relative to the optimal number. The tighter the WCEC, the lower the number of unnecessary checkpoints.

Estimating WCEC statically necessitates to have a representation of the program as well as an energy model which reflects the energy consumption of the system. For the former, the CFG of the program represents complex structures such as loops, conditions and function calls. For the latter, energy models at the lower levels of the software such as ISA are more accurate as they are closer to the hardware [28]. Figure 4.2 (a) shows a sub CFG of a program generated from the binary representation of the program. It contains eight basic blocks. The number besides each block indicates the worst-case amount of energy that the basic block consumes, computed based on the energy model (e.g. for simple architectures, by adding the amount of energy each instruction of the block consumes). In this CFG, the estimated worst-case consumption is 209 pJ.

For real-life applications, CFGs are large and the overall estimated WCEC is always much larger than the maximum energy a capacitor can provide. Also, due to the branches and loops, the number of paths from the start node to the end node is large. For instance, in the above mentioned simple CFG, the number of paths from node A to node H is `three`. This number increases as the CFG gets larger and more complex with branches and loops.

As typical in the real-time domain, interrupts and preemption cannot be handled at

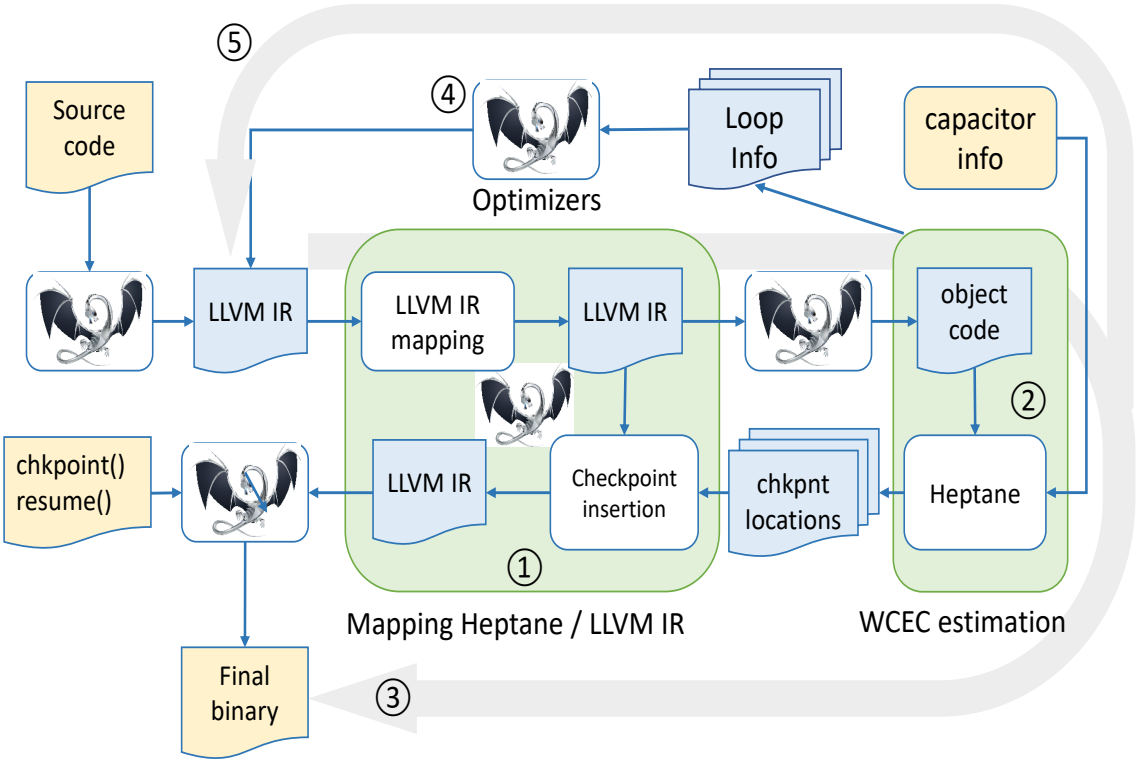


Figure 4.1: Overview of our flow

this level. If present, they should be handled in a distinct part of the system runtime.

4.2.2 Approach

Problem Statement. For any given capacitor size (i.e. energy amount), our approach consists in inserting checkpoints at various places in the program such that, from any checkpoint, another checkpoint can be reached, regardless of the dynamic path taken by the application.

Solution. To make the problem tractable, we adopted an algorithm based on Single-Entry-Single-Exit (SESE) regions [39]. SESE regions may be complex, but they have a single well-defined entry as well as a single well-defined exit node. As such, they provide convenient placeholders for checkpoints. Also, these regions can be nested, sequentially composed or disjoint (see Figure 4.2 (b)). The largest SESE region is the CFG itself as it has one start node and one end node [1]. The smallest SESE regions are basic blocks and

1. Even in the presence of multiple return statements in a function, a compiler can easily create a new block and add edges to it to guarantee a single exit node.

instructions. In our work, since our granularity for checkpoint placement is basic blocks, we chose the basic block as the smallest SESE region.
The input of the algorithm (see Algorithm 1) is the CFG of the program with its corresponding SESE regions, and the capacitor size. The algorithm starts by estimating the WCEC of the outermost region and if the estimated WCEC is bigger than the available energy, it recursively analyzes the nested regions.

Algorithm 1 Checkpoint Locating Algorithm

Data: CFG with Identified SESE Regions

Result: Checkpoint Locations

```
C is the energy of capacitor
CheckpointLocations is a vector of LLVM IR line numbers
r is the outermost region
IdentifyChKLocations(r,C)
 function IdentifyChKLocations(SESERegion R, AvailableEnergy E)
    e ← δ-WCEC(Entry node of R,Exit node of R)
    if e >E then
|     if R has nesting regions then
|     |     E ← E - (Energy consumption of Entry node of R)
|     |     for All Region N_i Nested in R do
|     |         remainingEnergy ← min_i(IDENTIFYCHKLOCATIONS(N_i,E))
|     |     end
|     else
|         remainingEnergy ← C - e, Add this Location to CheckpointLocations
|     end
    else
|     remainingEnergy ← E - e
    end
    return remainingEnergy
 end function
```

For estimating the energy of a region, we rely on partial WCET estimation (δ-WCET) proposed by Bouziane et al. [17, 16]. However, for the sake of clarity in Algorithm 1, we used energy consumption in lieu of execution time wherever relevant. For instance, consider the CFG of Figure 4.2 (b), and assume the capacitor can store 80 pJ. The algorithm first estimates the energy of region R1. Since the estimated value exceeds 80 pJ, it recur-

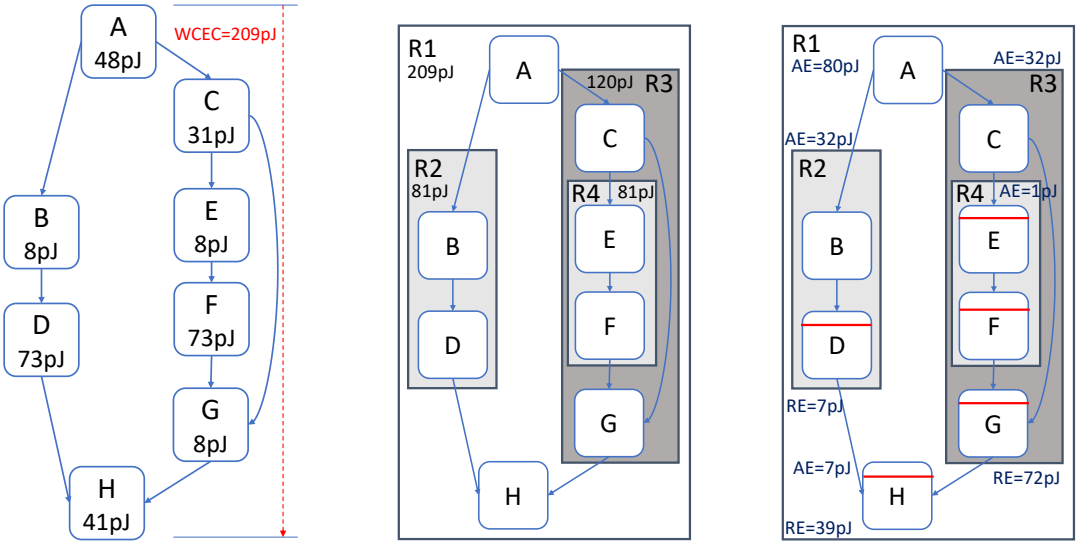
sively estimates the energy of R2 and R3 after subtracting the cost of block A (remaining energy of 32 pJ = 80 pJ - 48 pJ). It continues until it reaches the innermost SESE regions (basic block) and it places a checkpoint at the beginning of that basic block. The algorithm returns the amount of remaining energy. When it places a checkpoint, the return value will be the maximum amount of energy in the capacitor subtracted by the energy consumption of the basic block. Figure 4.2 (c) shows the CFG with inserted checkpoints.

As shown, the input of the proposed toolchain is the capacitor size and high-level C code annotated with maximum loop bound information. It is worth noting that specifying this information is the only supplementary effort requested from the programmer, and it is typical of embedded real-time systems. The output of our tool-chain is a binary code enriched with checkpoint trigger calls. Each checkpoint trigger is a call to a run-time library which is responsible for checkpointing the volatile state of the program into the non-volatile memory. The overhead and the energy cost of checkpointing itself is highly dependent on the underlying architecture. For architecture with non-volatile memory as unified memory, the cost of checkpointing is almost constant since only CPU registers must be copied. However, for systems configured with a volatile memory such as SRAM as well as a type of non-volatile memory, the cost of checkpointing is variable and dependent on program state such as the size of stack and heap, as well as the amount of live data at the time of checkpointing. In the latter case, we need to guarantee we have enough energy to perform the checkpointing in the worst case, and the location of the checkpoint matters. In the worst case the system must have enough energy to checkpoint all volatile memory. Our solution can deal with both types of architectures. However, in this work, we assume that there is always enough energy available for checkpointing a constant number of CPU registers in the former case or all volatile memory in the latter case, and we focus on the placement of checkpoints that guarantees correctness, and forward progress.

4.2.3 WCEC estimation

Our implementation consists of a component for estimating the WCEC (Heptane [31]), augmented with a checkpoint insertion algorithm.

Figure 4.1 shows the overview of our flow. The first component (box ① in the figure) is LLVM augmented by a few passes. It first builds a mapping between LLVM IR and binary code (see Section 4.2.4), compiles the code and invokes our WCEC estimation tool. This tool, called Heptane (box ② in the figure) determines where checkpoints should be located based on information about the capacitor and a power model of the architecture. This



(a) input CFG (8 blocks) (b) computed SESE regions (c) checkpoint locations

Figure 4.2: Input CFG, computed Single-Entry Single-Exit regions, and selected checkpoint locations (red lines, AE=available energy, RE=remaining energy)

information is fed back to LLVM for actual insertion of the checkpoints and production of the binary. This process is highlighted by the arrow designated as ③. We also explored how loop optimizations can be leveraged to decrease the cost of checkpoints (Section 4.2.6). In this case, Heptane selects appropriate loop optimizations that are forwarded to LLVM (box ④). This results in a new version of the program in LLVM IR (arrow designated as ⑤). The process of arrow ③ is repeated.

In addition, since Heptane works in binary code and our final checkpoint placement is in LLVM IR [44], we added a mapping between Heptane and LLVM IR described in Section 4.2.4.

Heptane [31] is a tool originally developed for estimating worst-case execution time (WCET) [76]. Heptane's functionality is divided into two separated components: `Heptane Extract` and `Heptane Analysis`. The former is for generating control-flow graph (CFG) of the program from the object code. The latter performs two types of analysis on the generated CFG: high-level analysis and low-level analysis. The low-level analysis compute an upper-bound for each basic block in CFG by considering the cost of instructions as well as features related to micro-architecture such as cache and pipeline. Then, the high-level analysis can compute the whole program's WCET by performing Implicit Path Enumeration Technique (IPET) [48] which is based on Integer Linear Programming (ILP) formulation of the WCET calculation problem.

In this work, since our concern is energy, inspired by Wägemann et al. [71], an energy cost for each instruction is specified instead of cycle cost that Heptane normally considers. Due to the simplicity of the processors in the domain, that is processors without caches and branch prediction, applying complex analysis in Heptane is not necessary. However, our approach is general and can include more complex architectures as long as an accurate (and safe) energy model is provided by manufacturer. Also, herein, the goal of work is not the WCEC of the whole program; instead we want to fragment the program into code sections which can be executed in one life cycle when the capacitor is fully charged. These code sections are bounded by checkpoint trigger calls. As a result, the location of these checkpoint trigger calls in the program must be identified.

4.2.4 Mapping between Heptane and LLVM IR

Since the analysis part is performed on binary code, and placing checkpoint trigger calls is applied at LLVM IR level, we need a mapping between the two. To achieve this, inspired from Grech et al. [29], we "hijacked" the debug information mechanism that propagates source-location information into a binary: at LLVM-level, we replace source location by the line number in the LLVM IR representation. The regular toolchain processes it as usual, carrying our information instead of traditional source line numbers.

We created two LLVM passes. *Source Line to LLVM IR* traverses the LLVM IR and replaces source location information with LLVM IR location information. After this pass, the binary code is generated and given to Heptane. As mentioned, the output of Heptane is a series of line numbers which specify where to place checkpoint triggers in LLVM IR. The *Checkpoint placement* is responsible to place checkpoint triggers based on the line numbers that Heptane produces. After this pass, the final binary code is generated and the program is ready to be executed on energy harvesting device.

4.2.5 Coping with non-termination

In our work, coping with non-termination and guaranteeing forward progress through the execution of programs is straightforward. In the WCEC analysis part, Heptane checks for every basic block whether its energy consumption exceeds the capacitor energy when it is fully charged. If a basic block consumes more energy than the maximum amount of energy in the capacitor, that basic block can easily be broken into smaller ones by the compiler. For that, Heptane reports the line numbers of the basic blocks that are too long.

LLVM subsequently splits these blocks and applies LLVM IR mapping pass again. It is worth noting that this process could also be done at assembly level. However, herein for the sake of simplicity we preferred to do at LLVM IR level.

4.2.6 WCEC-Aware compiler transformations and optimizations

Compiler optimizations heavily modify the structure of programs, hence the locations of checkpoints. As shown in the experimental section, optimizations are vital to decrease the amount of checkpointing at runtime. Our objective here is not to provide an exhaustive study of the impact of every optimizations, but rather to demonstrate the benefit of selected optimizations to the energy management of intermittently powered systems. As loops are the most time and energy consuming part of programs, we have chosen optimizations on loops. Also, placing checkpoints in loops can increase the energy consumption of the execution, making it critical to process loops with care.

Clearly, not all optimizations can be applied to all loops, since data dependences must be checked to guarantee program semantics is preserved. In this section, we briefly introduce the optimizations that we have selected. An exhaustive survey of compiler optimizations, their applicability and advantages can be found in Bacon et al. [7].

Conditional checkpoints is the simplest optimization that our toolchain applies, making checkpoints conditional on the loop iteration number. This is advantageous when a full charge of the capacitor is not enough to execute the entire loop, but we can prove that it can execute several iterations. Checkpointing at each iteration would be inefficient. This is illustrated on Figure 4.3 (a) and (b) respectively for the original loop, and the transformation, assuming we can execute three iterations of the loop with a single charge. This is always correct. Note that the same effect would be achieved by unrolling the loop three times and inserting a single checkpoint. Unrolling, however, increases code size and it is rarely applied on small systems with limited amount of memory.

Loop splitting divides the iteration space in two pieces (or more) with the same loop body. An example is given in Figure 4.3 (c). The effect is similar to conditional checkpoints, but it saves the cost of the condition, at the expense of duplicating the loop body.

Each generated loop can be executed in one charge of capacitor. Splitting is always legal.

Loop fission (aka loop distribution) also divides a loop into pieces, but takes statements apart. See the example on Figure 4.3 (d). Depending on the energy cost of the statements (S1 and S2 in the figure) and the value of n, it may offer more interesting tradeoffs. In this optimization new loops have less energy consumption and can be processed in one charge of capacitor. The checkpoint trigger call is also between two loops. Fission is not always legal, and depends on the type of dependence between S1 and S2, if any.

Loop interchange consists in swapping two loops in a loop nest (see Figure 4.3 (e) and (f) respectively for original and transformed loop nests). This is profitable when $n < m$ and the body of the nested loop can be executed m times with a single charge. This reduces the number of taken checkpoints from m to n. Interchange is not always legal, depending on dependences. It also impacts locality, however small intermittently powered devices typically do not feature a data cache. In case they do, the reduction in the number of checkpoints should be balanced with the overhead in cache misses.

Optimizations are not hindered by checkpoint insertion which is applied later in the compilation flow. It is worth noting that each optimization affects the estimated WCEC. For instance, in conditional checkpoints, a few instructions are being added which alter the WCEC. For preserving the safety of the WCEC, we run Heptane for the second time after each optimization. Also, one complication derives from the fact that loop bounds must be statically known for the tools to estimate a worst-case execution time/energy consumption. This is typically done by manually adding annotations to loops in source code, and disabling compiler optimizations to guarantee that the CFGs at source and binary levels match. Tracing loop bounds through the entire compilation flow is complex. We address a simpler problem, focusing on particular loops and applying a limited repertoire of transformations. For example, the bounds of a split loop are obtained by dividing the original bound by two. Fission creates a new loop with the same bound. In the case of interchange, bounds must be interchanged as well. For a larger set of optimization, careful tracking of annotations must be enforced. However, previous work by Li et al. [47] has shown the feasibility.

To preserve the safety of the WCEC estimation, after each optimization, the WCEC estimation part of our tool-chain checks that with the insertion of the checkpoint and the transformation the new WCEC provides safety. When the safety is approved with the WCEC part, the generated binary code can be executed on the device.

```
for(i=0; i < n; i++) {
  S1;
  S2;
}
```
(a) original loop

```
for(i=0; i < n; i++) {
  if (i % 3 == 0)
    checkpoint();
  S1;
  S2;
}
```
(b) conditional checkpoint

```
for(i=0; i < m; i++) {
    for(j=0; j < n; j++) {
        S1;
    }
}
```
(e) original loop nest

```
for(i=0; i < n/2; i++) {
  S1;
  S2;
}
checkpoint();
for( ; i < n; i++) {
  S1;
  S2;
}
```
(c) loop splitting

```
for(i=0; i < n; i++) {
  S1;
}
checkpoint();
for(i=0; i < n; i++) {
  S2;
}
```
(d) loop fission

```
for(j=0; j < n; j++) {
    checkpoint();
    for(i=0; i < m; i++) {
        S1;
    }
}
```
(f) loop interchange

Figure 4.3: Loop optimizations

Benchmark	Description
bs	Binary search for the array of 15 integer elements.
bsort100	Bubblesort program.
crc	Cyclic redundancy check computation on 40 bytes of data.
edn	Finite Impulse Response (FIR) filter calculations.
fdct	Fast Discrete Cosine Transform.
fibcall	Iterative Fibonacci, used to calculate fib(30).
insertsort	Insertion sort on a reversed array of size 10.
minmax	Simple program with infeasible paths and without loops.
ndes	Complex embedded code. Bit manipulation, shifts, array and matrix calculations
mm	rectangular matrix multiplication

Table 4.1: Benchmarks used for the evaluation

4.3 Evaluation

4.3.1 Settings

We assigned an energy cost to each instruction for ARMv6-m ISA, derived from an actual core synthesized in 28 nm ST FDSOI. This ISA is used for a number of processors such as ARM Cortex-M0+ in low-power domains. We run the final executable generated by our tool-chain in a modified version of a cycle-accurate simulator for the mentioned ISA [69].

4.3.2 Benchmarks

We selected nine benchmarks (Table 4.1) from the Mälardalen suite [30]. They are highly used in research publications on embedded systems and sensors. In addition, in terms of CFG complexity, they contain loops, inner-loops and function calls. Therefore,

they can show the effectiveness and correctness of the proposed checkpoint placement strategy. Also, they perform a significant amount of computation in the main MCU core, as this work is only focusing on the main CPU computation. We added a matrix multiplication we developed ourselves to experiment with cases not covered in the suite.

4.3.3 Experiments

To show the sensitivity of our approach, we tested several benchmarks with a wide range of capacitor sizes, selected with respect to the overall WCEC of the benchmark. We apply our algorithm to insert checkpoints, run the program in the simulator, and report the number of checkpoints taken during execution. On Figure 4.4, the x-axis represents the capacitor size in picojoules (pJ). The y-axis represents the number of checkpoints taken at runtime (logscale). The bold purple line shows the number of checkpoints in the absence of optimizations, while other lines show the impact of individual optimizations. minmax is not reported here because it contains no loop, we discuss is below.

We explored with very small capacitor sizes, such as 200 pJ, which may not be realistic for deployed IoT systems [2]. The reasons are: 1) the Mälardalen benchmarks have rather small execution times and we still wanted to experiment with a wide range of values; 2) we wanted to test the scalability of our approach towards small as well as large values.

Number of executed checkpoints. As expected, our strategy is sensitive to capacitor size, and the number of taken checkpoints decreases when the capacitor size increases. However, we also observe long plateaus where the number of checkpoints remains constant for a wide range of capacitor sizes (e.g. bs between 500 pJ and 1600 pJ). The main reason for that is the presence of loops. As long as the execution of the entire loop (with worst-case trip count) requires more energy than the capacitor can provide, a checkpoint must be placed inside the loop body. In some cases, it also happens that our toolchain cannot place the checkpoints exactly where Heptane suggested (i.e. just outside the loop) because LLVM IR instructions are sometimes slightly coarser than assembly instructions. This explains the occasional peaks of the curves.

Our work in comparison to related work, namely Mementos [62] from Ransford et al. can guarantee forward progress and program correctness. Similar to our work, Mementos

2. The available energy can be related to the capacitance C through the formula $E = \frac{1}{2}C(V_{start}^2 - V_{stop}^2)$, where V_{start} and V_{stop} represent voltages when the capacitor is full, and when the system is forced to stop because the voltage is too low.

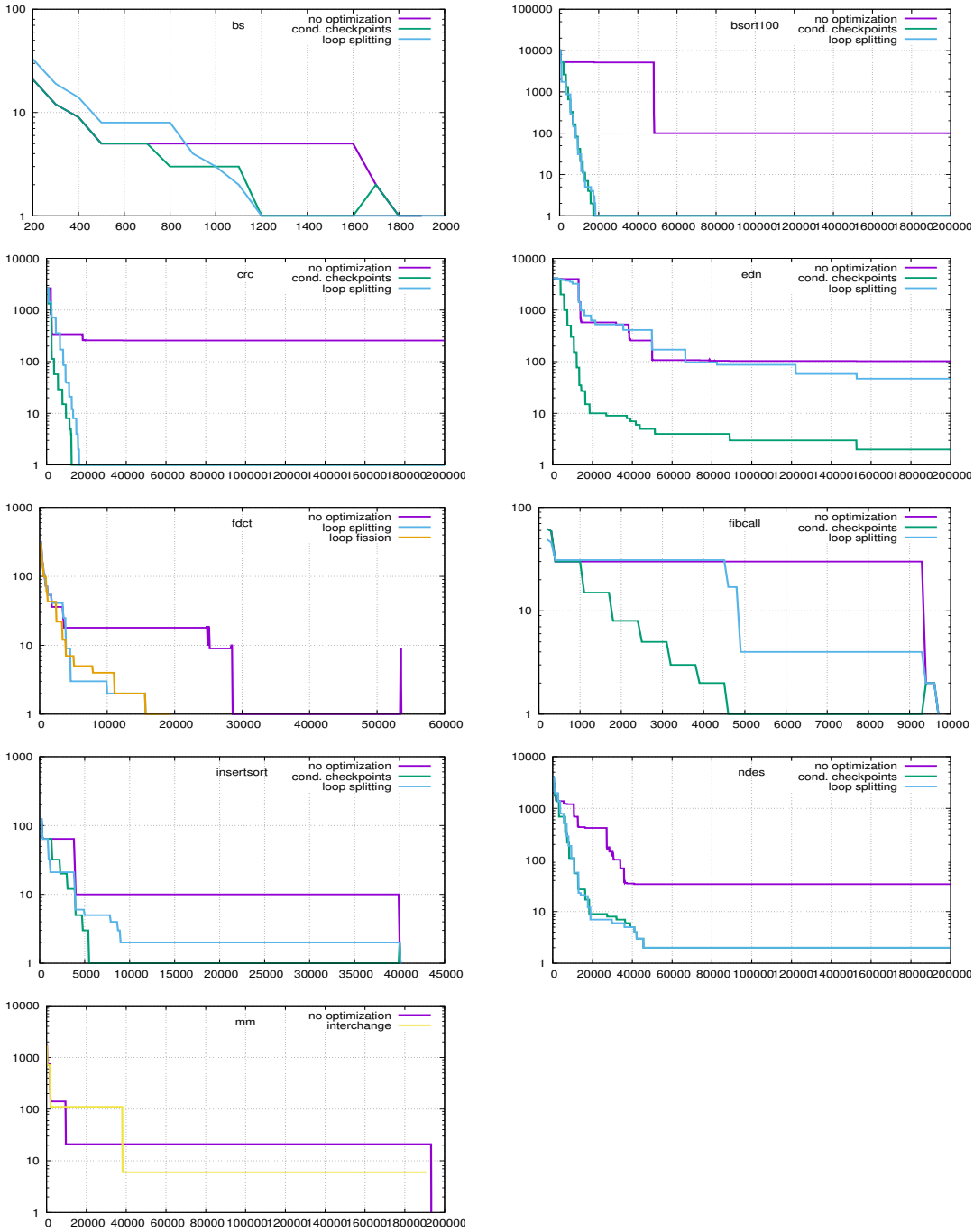


Figure 4.4: Number of taken checkpoints when the capacitor size changes, without and with optimizations

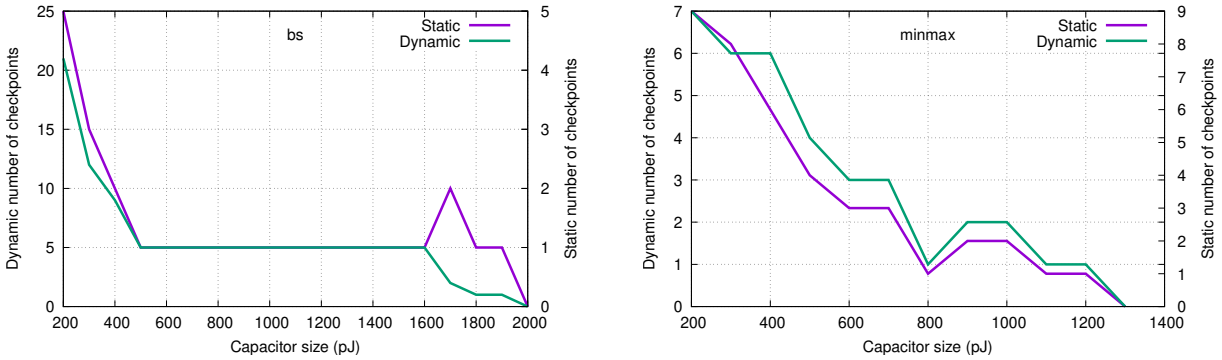


Figure 4.5: Evolution of the dynamic and static numbers of checkpoints.

inserts checkpoints in the CFG. However, they only considered specific locations and opted for loop latches or function returns. Mementos checks the remaining energy (actually the voltage as a proxy for energy) only at these predefined locations. In case a loop body or a function (without loop) requires more energy than the capacitor can provide, they cannot prevent unprotected energy depletion, and thus cannot guarantee forward progress. Also, if Mementos had the ability to modify the non-volatile memory by the program semantic, a failure may corrupt the memory and cause the program to be incorrect. In comparison to Ratchet [69], which checkpoints between every WAR dependence, our work is more efficient in terms of number of both static and dynamic checkpoints. For instance, Ratchet is forced to insert checkpoints in the example provided Baghsorkhi et al. [8] no matter how much energy is in the capacitor. But, as our checkpoint placement is sensitive to the capacitor size, it can place checkpoints outside the loop whenever it is possible to process a loop with one full charge of capacitor.

Impact of optimizations. Our results show that optimizations have the capability to reduce the number of necessary checkpoints.

First, making checkpoints conditional is always beneficial, and this can have dramatic impact. For example, unoptimized fibcall requires 30 checkpoints to complete execution, whereas conditional checkpoints can progressively reduce this number down to 15, then 8, and so on. The same behavior is visible in all benchmarks.

Loop splitting is also very effective, and in most cases similar to conditional checkpointing. When they differ, splitting is usually underperforming. This is visible in bs for small capacitor sizes, as well as fibcall and crc. Yet, in some occasions, splitting slightly outperforms conditional checkpointing: particular values in bs, or ndes between 20,000 pJ

and 40,000 pJ.

Fission was applicable to fdct. It is similar in effect to splitting, slightly better or slightly worse, depending on values.

Interchange was not successfully applied to our set of benchmarks. This is why we added a simple matrix multiplication example. Our results show that interchange can reduce significantly the number of checkpoints for 141 to 111 for small sizes, and from 21 to 6 for larger sizes. For intermediate values, interchange degrades performance.

In some cases, optimizations produce worse results than unoptimized code. This can be observed with splitting in bs for small values, edn, and marginally in fibcall (31 vs. 30), as well as interchange as discussed. This behavior is not different from optimizing for performance that is tuned for average case and occasionally results in poor performance.

So far, we only experimented with applying a single optimization at a time, in order to show the potential. It is well known that optimizations interact in complex. A promising direction consists in combining many optimizations to decrease the number of checkpoints. Iterative compilation can certainly be applied to explore a region of interest of the optimization search space, and heuristics be developed in a way similar to compiling for performance.

Number of static checkpoints. Figure 4.5 shows the static number of checkpoints inserted along with the number of times they are taken at runtime (dynamic number on the left axis, static on the right) without any optimization. We only report a limited number of benchmarks, as they all exhibit the same behavior.

Overall, the number of static checkpoints is decreasing as the number of dynamic checkpoints, as expected. However, there are exceptions, as in bs. The reason for that is when it is possible to process the whole loop with a full charge of capacitor size, our algorithm places two checkpoints right before and after loop instead of placing the checkpoint inside the loop. The first checkpoint is to have energy for processing loop and the second checkpoint is to have energy for continuing the rest of the code. The number of static checkpoints only impacts on the code size. However, each checkpoint that is taken at run-time consumes time and energy. So, it is worthy to increase the number of static checkpoints whenever it is possible to decrease the number of dynamic ones. For minmax, when the capacitor size is increased from 800 pJ, an increase in the number of static and dynamic checkpoint is observed. This is because our algorithm is biased to place checkpoints before a function call as a function might have more than one context (call

site). However, minmax is a very simple program, all functions have only one context. It is better to process the function and place the checkpoint when it is necessary. In the future we will consider the number of contexts of a function and improve the number of checkpoints. Also, minmax is a benchmark without loop and it has infeasible paths which are never taken at runtime. This is the reason why the number of static checkpoints for some capacitor sizes is larger than the number of dynamic checkpoints.

4.4 Conclusion

We propose a compile-time checkpoint insertion strategy for intermittently powered system. Our approach simultaneously guarantees program correctness and forward progress. It does not require any additional hardware. To achieve this, our toolchain inserts checkpoint trigger calls based on worst-case energy consumption of program sections. The called function saves the state of the program to non-volatile memory before the energy depletes. In addition, we show that classical compiler optimizations can be exploited to reduce the number of checkpoints, hence the overhead.

Chapter 5

Dynamic Checkpoint Placement based on self-modifying code

5.1 Introduction

This chapter presents the second approach to eliminate problems related to intermittent execution of programs running on an energy harvesting device. As mentioned before, both hardware-based and joint hardware/software solutions can take checkpoints at the cost of adding additional hardware features or using hardware features of the commercial MCUs which are originally designed to be utilized by the programmer (e.g., timers, ADCs) and not by the third-party system software. Therefore, like the previous chapter, again our main goal is to propose a fully software solution.

The contribution of this work is SFSG [1]: an **adaptive** checkpointing scheme **at run-time** based on the actual execution of the program in the deployment environment, and **without using any dedicated hardware feature**, and totally **transparent** to the programmer.

The organization of this chapter is as follows. Section 5.2 presents the motivation behind the work. Section 5.3 presents the overview of SFSG. The compile-time part of SFSG is described in Section 5.4, while the run-time part is covered by Section 5.5 (trace collection) and Section 5.6 (checkpoint management). We evaluate SFSG in Section 5.7, and discuss results as well as possible future extensions in Section 5.8. Finally, Section 5.9 concludes.

1. SFSG stands for *so far, so good*: in the learning steps, we execute the program as far as we can and crash when we run out of energy. We then remember where we crashed and introduce a checkpoint a bit earlier.

5.2 Motivation

In this work, inspired by the idea used in dynamic compilers, we have adopted a framework including several compiler analysis and transformation passes as well as a runtime system. Our work postpones the final decisions about the location of the checkpoints to the runtime. The decisions that it takes at runtime are based on the execution path that the program had taken before power failure happened. In this way, the runtime system learns from power failures in order to place checkpoints. Depending on the variability of the harvesting source and the environment, in most cases the program takes the same path and consumes the same amount of energy, after it faces a power failure. Also, our approach is self-adaptive in a sense that with the changes in program runtime behavior, the runtime system can change and specialize checkpoint locations (see Section 5.6.2). Moreover, our work guarantees termination by identifying those sections of program's code that consume more energy than the capacitor's energy when it is fully charged. Our work is a fully software approach. It does not require any special hardware features or preempt any features originally available to the programmer (peripherals of the MCU such as a timer or the ADC of the system) which may be required by the application. The concept presented here can thus be applied in a wide range of commercial and commodity devices. We illustrate it on an MSP430 board from Texas Instruments.

However, two things that complicate applying concepts of dynamic compilers for ultra-low power IoT devices are (1) lack of time to generate code, as the typical active time in these devices are in terms of milliseconds [18] (2) lack of sufficient memory, as usually dynamic compilers consume significant amounts of memory whereas the typical available memory for an energy harvesting device is in terms of kilobytes [34]. To cope with these limitations, we have adopted a fast with moderate memory overhead runtime technique for placing and specializing checkpoints into the code. We achieve this by applying a runtime Self-Modifying Code (SMC) technique which leverages the information provided by static analysis. This synergy – sometimes referred to as *split-compilation* [22] – has been shown to deliver good runtime results at low cost.

5.3 Overview of SFSG

Intuitively, SFSG is based on the observation that an IoT program performs a series of tasks continuously for a long period of time and control paths repeat during the execution.

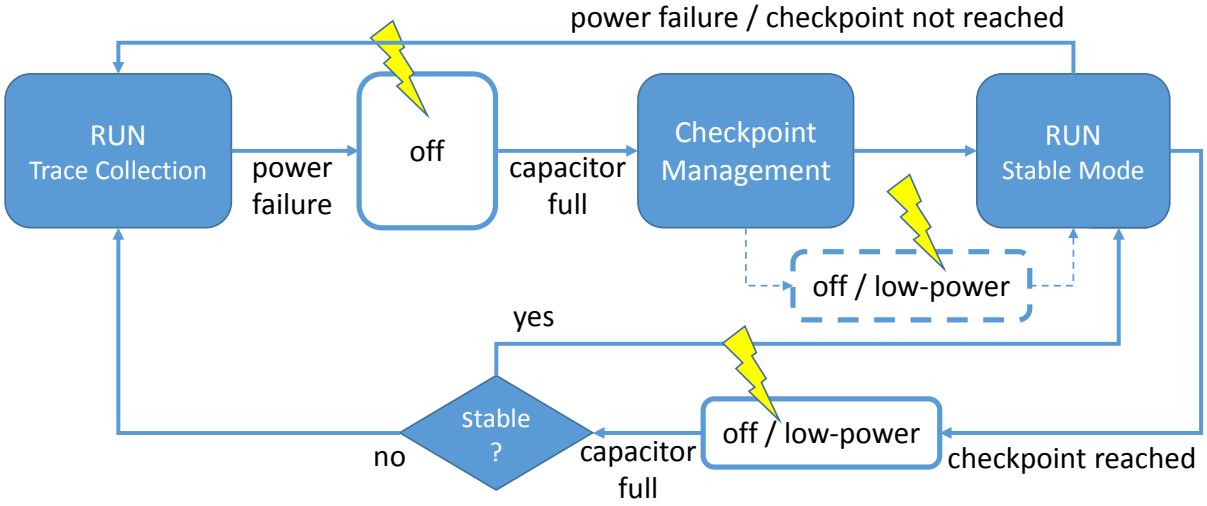


Figure 5.1: Flow of runtime states

For instance, small IoT devices often consist of an infinite loop that senses some data from the environment, computes, possibly encrypts, and transmits the result. But repetition of control flow applies – to varying degrees – also to most applications. This is the reason behind profile-guided optimizations, but also instruction caches and branch predictors.

In a nutshell, we let the program run, only slightly modified to generate an execution trace in non-volatile memory, until energy depletes. When power resumes, and the system restarts, we know from the trace where it crashed. We insert a checkpoint based on the collected trace and restart execution from the beginning. Hopefully, execution reaches the checkpoint. Future re-execution will resume from this new point, thus guaranteeing forward progress. If the checkpoint is not reached, we insert it slightly earlier in the CFG and restart. At the checkpoint, we wait until the capacitor recharges completely.

Our system pays a penalty when a code path is seen for the first time. A higher price is paid if control paths change often. However, even in the latter case, our systems remains correct and makes progress towards its execution: we are not limited to any particular class of programs.

The runtime system is responsible for the overall orchestration of the various steps at play: collecting execution traces, inserting and specializing checkpoints calls, managing and preparing trampolines, and finally checkpointing and restoring the necessary state of the program. Figure 5.1 illustrates the corresponding finite state machine. The states behave as follows.

Trace Collection: In this state, while the program is executing, the trace is also be-

ing collected. These traces are stored in non-volatile memory so that, when a power failure happens during execution (the transition state from **Trace Collection** to **off**), they are still available.

off: In this state, the system is inactive or totally shut-down. In this situation the system is harvesting energy from the environment. When the system stores enough energy it wakes up and goes to the **Checkpoint Management** state.

Checkpoint Management: In this state, the runtime's job is to place, modify and specialize a checkpoint call trigger based on the available collected trace in non-volatile memory. This is discussed in detail in Section 5.6. After this state, the system goes to the execution with `Restore` routine if there is a backup in the system or if not, it starts executing from the beginning of the program which is the main function.

It is worth noting that, some MCUs provide different level of low-power modes. As the state **Checkpoint Management**, consumes some amount of energy depending on the job that it does, sometimes it is better to go to a special **low-power** state and come back when the capacitor is fully charged. However, here in this research, we have not used this capability of MCUs. We have just shown this state in order to illustrate that SFSG is extendable to support this feature depending on its availability on the MCU.

RUN: In this state, the program is executing but this time without collecting trace and with the hope of reaching the checkpoint trigger call placed in state **Checkpoint Management**. If the system reaches the checkpoint and manages to do the checkpoint successfully, it goes to a state called **off/low-power**. Otherwise, it goes to **Trace collection** state and starts executing and collecting a trace again.

off/low-power: In this state, whether the system is in a low-power mode or totally inactive, it does not do anything and is waiting for a wake-up signal to be raised. At the time when the wake-up signal is raised, if `stable` variable is *true*, the system goes to state **RUN** and executes without collecting any trace. Otherwise, it goes to **Trace Collection** state and execute whilst collecting a trace. The `stable` variable is in NVM and it is being set to *true* when the system is stabilized in terms of checkpoint locations (see Section 5.5).

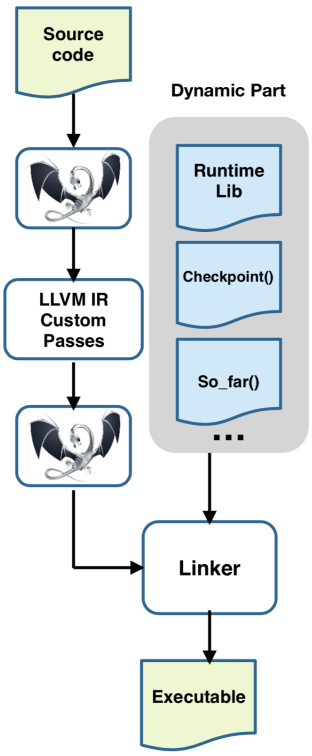


Figure 5.2: The overview of SFSG

5.4 Static program preparation

Figure 5.2 illustrates the overall process. It includes static and dynamic phases and management of the checkpoints is shared between them. Static phase transforms CFGs and extracts properties from the program and stores them alongside the binary. It also instruments the program to add control points in various places. Figure 5.3 shows how `main()` and `Foo()` CFGs are transformed. We also developed a library that contains all the mechanisms needed by SFSG runtime.

We leveraged the LLVM compiler [44]. Our new LLVM passes perform the following steps.

1. They add preheaders to loops, if not already present. A preheader is a node outside the loop that immediately dominates the loop head node (such as **B** and **J** nodes in `main()` and `Foo()` CFGs in Figure 5.3. Therefore, preheaders are good candidates for placing checkpoints, when power failures happen within a loop. As repeated checkpointing in a loop body may have higher overhead for the system, it is better to place the checkpoint outside the loop in the first step to give a chance to system

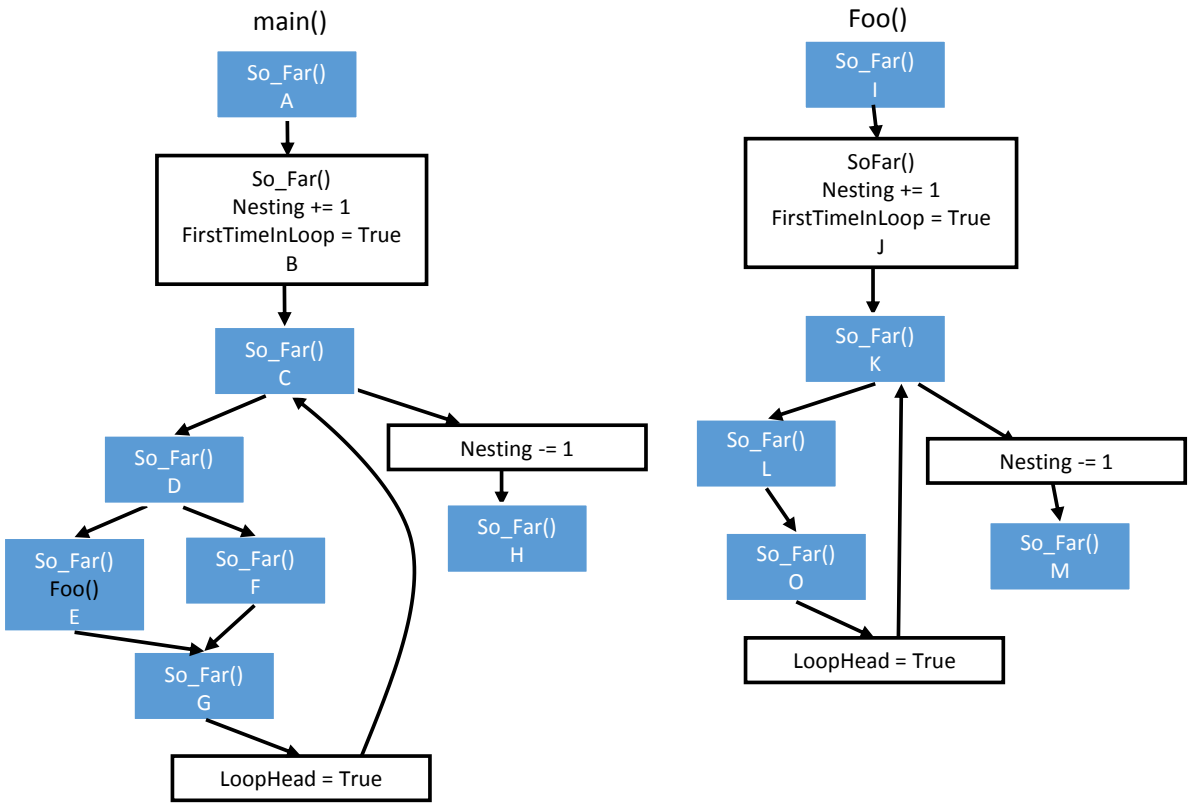


Figure 5.3: Static transformations of CFGs

to process a loop with one charge of capacitor.

2. They insert the special function trigger call `so_far()` at the beginning of each basic block in the CFG of the program. The purpose of this function is to collect execution trace at run-time.

3. As during the execution of the program, the runtime system must be aware of the type of the basic blocks (e.g., loop head), loops and nesting, at compile-time, we add some statement into the different locations in the program and loops:

   (a) At each preheader, it increments a variable called *Nesting* and changes the value of another variable called *FirstTimeInLoop* to *true*.

   (b) At each loop exit, it decrements the *Nesting* variable.

   (c) At each loop-latch, it changes the value of a variable called *LoopHead* to *true*

4. They rename the main function of the program, and substitute our own entry point to perform overall orchestration of the system at startup (note that when

the system resumes from an earlier checkpoint, the original main is not where we want to direct execution). They also add `so_far()`, checkpoint and restore routines into the address space of the program. For the latter two routines, we use checkpoint and restore routines from Mementos [62].

5. They provide in non-volatile memory some information for the run-time part, for instance, the address of the beginning and the end of all basic blocks.

5.5 Trace management

We redefine a trace as a sequence of basic blocks that are executed in a power cycle (before a power failure). Traces may become huge, and can easily exceed the total memory size of small IoT devices, typically in the order of kilobytes. We have devised a simple and efficient trace collection algorithm to collect the information we need within a small amount of memory.

5.5.1 Trace collection

The `so_far()` function which is provided by our library contains a few low-level C and assembly instructions appending the address of the first instruction of each basic block (the address of `CALL so_far()`) to a reserved area in non-volatile memory as a trace element. Figure 5.4 shows the skeleton of this function. On this Figure *FirstTimeInLoop*, *Nesting* and *LoopHead* are volatile variables. *TraceInfo* and *LoopInfo* are representing the locations in non-volatile memory for stroring trace elements and loop information. We demonstrate SFSG trace collection by making different power failures scenarios during the execution of the CFGs of Figure 5.3. The runtime system uses reserved locations in non-volatile memory for different types of information. For instance, in Figure 5.5 which represents the execution of `main()` function over time, it reserves two locations for trace and loop information.

The program starts its execution from basic block `A` of the CFG `main()`. At this point, the reserved areas for trace elements and loop information are empty (t_0). When `so_far()` is being executed, it appends element `A` in trace location and increments the number of trace elements (t_1). In t_2, which shows the execution after basic block `B`, the same scenario is repeated. However, this time just after the execution of `so_far()`, volatile variables' values *FirstTimeInLoop* and *Nesting* are changed. The purpose of these variables is to

inform the runtime system that the next basic block is a loop head and is nested. Also, as these variables are volatile, if a power failure happens during the execution of basic block `B`, they do not have any effect on the trace. Still the runtime will know that the last execution basic block was `B` and it was outside any loop or nesting. When the basic block `C` is being executed, the runtime can identify the basic block as a loop head. It first appends element `C` to trace locations. Then, it allocates space for two pieces of information related to loop information: the trace element `C`, a pointer to the trace location that contains trace element, and the iteration of the loop (t_3). It also saves the *Nesting* variable in non-volatile memory. In this example, in the first iteration of the loop, the program takes `DFG` path and the corresponding collected trace in t_4. In the loop latch of `main()` function, the volatile *LoopHead* value will be changed to true and when `so_far()` in basic block `C` is being executed it starts the trace from the location of loop head `C` in loop information location and also increment the loop iteration (t_5). At runtime in the second iteration, the program takes the path `DEIJKLO` as shown in t_6.

5.5.2 Ephemeral tracing

Tracing is expensive, and we attempt to limit the cost to where it is needed, that is the discovery of new code paths. Disabling and enabling tracing can also have a cost, and disabling too soon may be detrimental. We designed a two-step approach to gradually limit overhead: *tracing mode* and *stable mode*.

Tracing mode is on when we discover new code, but to reach a recently placed checkpoint, SFSG temporarily turns the tracing mode off and runs without collecting trace. However, after reaching the checkpoint or facing a power failure during execution, the system enters *tracing mode* again. After each iteration of the outermost loops of the program, a function (`is_stable()`) which has been inserted by the static part of the SFSG will be triggered. Typically, an IoT program contains one outermost loop that performs a series of tasks. But SFSG is not limited to this type as it places the function for all loops with dynamic nesting level of one. At runtime this function checks whether at least one checkpoint has been inserted. If there is a checkpoint in the system, the system enters stable mode.

In *stable* state, we assume we have reached a steady state, and the system stops collecting traces for better performance. This is speculative, and tracing may have to be temporarily re-enabled, should a new power failure happens. Stable mode is enabled by setting a non-volatile global variable (`is_stable`). In this state changing the values of

```
void so_far()
{
  if (System.state == DoNotCollectTrace) {
      if (FirstTimeInLoop || LoopHead)
        /* modify loop information */
    return;
   }
  if (Nesting) {
    if (FirstTimeInLoop) {
      TraceInfo.append(element);
      LoopInfo.append2(LoopHead.details);
      FirstTimeInLoop = false;
     } else if (LoopHead) {
        if (TraceInfo.find(LoopHead)) {
          TraceInfo.goto(LoopInfo.LoopHeadPtr);
          TraceInfo.append(LoopHead);
        } else {
          TraceInfo.clear();
          TraceInfo.append(element);
        }
       LoopHead = false;
     } else {
       TraceInfo.append(element);
     }
   } else {
     TraceInfo.append(element);
   }
 Nesting_NV = Nesting; /* save to NVRAM */
}
```

Figure 5.4: Skeleton of trace collection entry point.

loop information is still mandatory.

5.6 Runtime checkpoint management

We first discuss how SFSG determines the location of checkpoints, and second how insertion of checkpoint is performed.

5.6.1 Determining checkpoint locations

Based on our trace collection methodology discussed in the previous section, after each power failure, the runtime system has information about the last basic blocks that had been executed and the corresponding loop iteration and nesting.

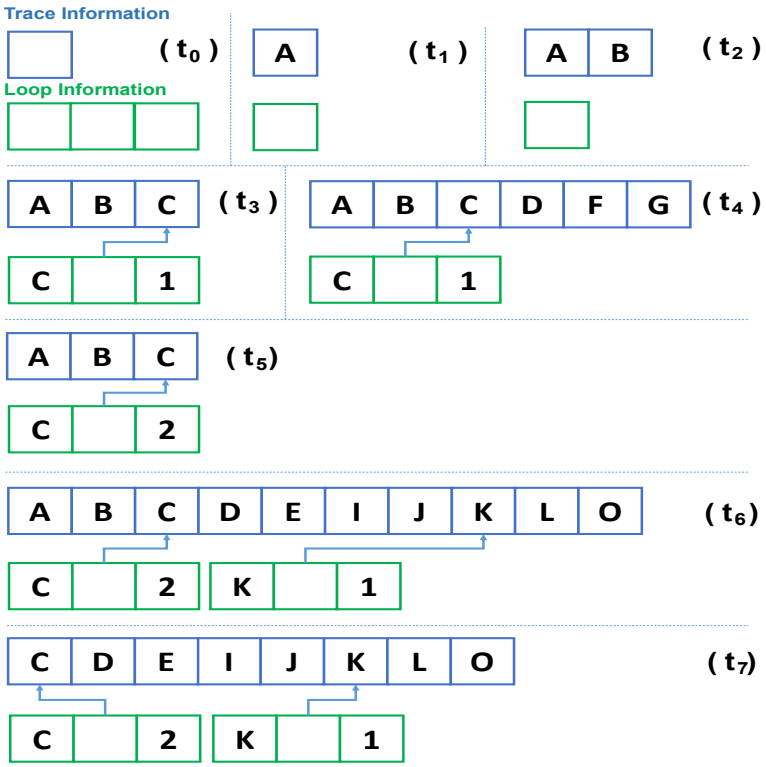


Figure 5.5: Trace Collection while executing basic blocks

If the selected trace element is a basic block in a loop, the runtime first (optimistically) attempts to place the checkpoint outside the loop, in the hope to execute the entire loop with a single charge of capacitor. For that purpose, the call is inserted in the loop pre-header (whose existence is guaranteed by the static compiler) as well as the loop information location. For instance, if at t_6 a power failure happens, the compiler starts with the outermost loop (`C`). The pre-header of the loop is one element before the loop head in the trace (`B`). In our trace example, after restart, the program will be re-executed from the beginning but this time at basic block `B`, a checkpoint will be taken and the system will stop executing until the capacitor will be fully charged. If the program executes the same path and fails at basic block `O` again, the corresponding collected trace will be t_7. However, this time the runtime, places the checkpoint in the pre-header of basic block `K` as the pre-header of basic block `C` does not exist in the trace anymore (`C` element is in the beginning of the trace)

If (after restart) the power failure happens in the same loop again, the runtime inserts a second checkpoint in the loop body where the power failure happened. Additionally, the checkpoint is *specialized*, i.e. made conditional upon the iteration count. If a single charge

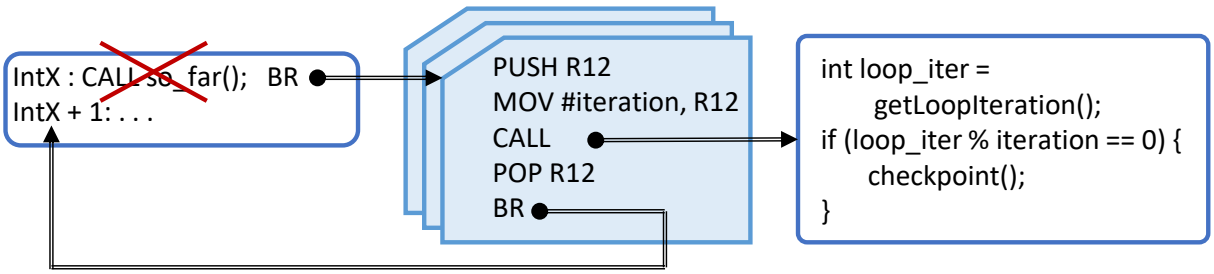


Figure 5.6: Checkpoint specialization with Trampolines

of the capacitor was able to execute n iterations, we only take the checkpoint every n^{th} iterations (if $n \neq 1$).

In the rare case the trace element was a basic block outside any loop, the runtime simply inserts the checkpoint routine call without any specialization. It is worth noting that each time the runtime inserts a checkpoint in the code, it saves the address of the location of checkpoint as well as the trace to which it belongs in a reserved area in memory.

If the program cannot reach the checkpoint and faces a power failure, this implies that the location of the checkpoint was not correct or the program took a different execution path. In both cases, since the runtime saved the last checkpoint location and its trace, it can decide whether to revert the checkpoint location with a `call so_far()` instruction or not. In SFSG, if the saved trace and new trace are totally different, it keeps the checkpoint trigger call. Otherwise, it reverts the checkpoint location with a `so_far()` instruction. SFSG can find a new checkpoint location based on recent collected trace.

5.6.2 Specialized checkpoints

The process of checkpoint specialization is illustrated in Figure 5.6. A trampoline is created for each selected location in non-volatile memory. It is a small chunk of memory where we generate a few instructions that assign a constant to R12 (in the MSP430 ABI, R12 is used for passing the first parameter of a function) and call the actual checkpointing routine. The constant is used to specialize the checkpoint to a subset of loop iterations, as described previously. In the checkpointing routine, *getLoopIteration()* reads the value of the current loop iteration count which is in loop information location in non-volatile memory. Whenever the checkpoint should not be specialized (i.e. R12=1), we bypass the trampoline and call directly into the checkpoint function. We then overwrite the call to `so_far()` by a jump to `checkpoint()`. Note that we also need to save and restore the value of R12, hence the `PUSH/POP` instruction pair.

5.6.3 Coping with non-termination

There is a scenario that could prevent the program from making forward progress in its execution and that is when the energy consumption of a single basic block is more than the energy provided by the capacitor when it is fully charged. The reason for that is the basic block might be too large in terms of the number of instructions or it may contain some energy consuming operations such as I/O (input/output, e.g. writing to a device, sensing, or sending a radio packet). As I/O operations are not latch-able and they must be executed atomically, our framework does not place checkpoints inside them. However, the SFSG runtime can identify these basic blocks. If a power failure happens and after that there was not any collocated trace, it means that the energy consumed by the basic block is more than the energy that the capacitor can provide. To resolve this issue, the runtime of SFSG heuristically splits the basic block and chooses the middle of the block for placing a checkpoint. It places a branch instruction as a trampoline in the middle of the basic block by overwriting the existing instruction. The branch instruction branches into a location in NVM. The runtime then places the overwritten instruction there and places a checkpoint trigger call after that. However, for identifying the middle of a basic block the beginning address and the end of a basic block are needed. To achieve that, the static part of the SFSG toolchain uses information provided from the object code of the program. It identifies addresses of the first (`so_far()`) and the last instruction of each basic block and stores them in-order somewhere reserved in NVM. At runtime, the system can identify the beginning and the end of the problematic basic block. In a fixed-length ISA, the middle of the basic block can be found in a straight forward way. The system can find the middle address of the basic block for placing the trampoline. However, for a variable length ISA such as MSP430, we have implemented a simple and fast instruction decoder for identifying instructions' sizes. It scans instructions from the beginning of the basic block until it reaches the middle of the basic block. This process can be applied recursively in binary search manner until the system can make forward progress.

The whole process of splitting a basic block may not be completed in a single charge of the capacitor. However, it is possible to extend SFSG to support partitioning the process. For instance, to find the middle of the basic block, it can store the address of the recently decoded instruction in NVM. If a failure happens during the process, the system can resume finding the middle of the basic block from the last decoded instruction.

5.7 Evaluation

The typical applications we target are meant to run forever. Total execution time makes little sense in this context. We evaluated separately the behavior of the discovery phase (early execution phase) and the stable mode (most of the execution time, after checkpoint insertions have stabilized). For the former, we studied the number of inserted checkpoints (Section 5.7.1), how fast insertion stabilizes (5.7.2). For the latter, we measured the overhead of tracing, and the residual cost when most of the overhead is removed (Section 5.7.3). Finally, we also measure the code size increase (Section 5.7.4).

Hardware. We evaluate our work on a TI MSP430FR5969 launchpad. This board utilizes a 16-bit MSP430 RISC instruction set architecture. The core is clocked at 8 MHz. It also leverages 2 KB SRAM as volatile memory as well as 64 KB FRAM as non-volatile memory.

Benchmarks. We have chosen five benchmarks: A Fast Fourier Transform (FFT) and an RSA cryptography benchmark from Mementos repository [5], CRC32 (CRC error checking 32-bit), aha-mont64 (Montgomery multiplication) and ud (Simultaneous Linear Equations by LU Decomposition) from Embench [19]. They only perform computations and they do not contain any I/O. However, these benchmarks are highly used in the IoT domain. As mentioned earlier, IoT applications typically run for a long period of time. Therefore, we made each benchmark iterate a number of times to see the runtime behavior of our work. To show that SFSG makes forward progress to completion without facing non-termination in extreme cases, we have developed a benchmark which uses the timer of the system periodically (Figure 5.7). The body of `func` function in our test benchmark is linked after static program preparation phase of SFSG. So, it is executed atomically at runtime as it is not visible for SFSG runtime to place a checkpoint in it. It spends the time of the system based on its input.

Compilation infrastructure. We used the Clang compiler and the LLVM infrastructure for the static part of our work, and the gcc-msp430 linker to produce the final executable. We compiled each benchmark with `-O1`.

Protocol. To explore how SFSG adapts to diverse situations (different capacitors, changing environment, aging), we experimented with varying capacitor sizes. Since it

```
for (unsigned int i = 0; i < N; i++) {
   for (unsigned int j = 0; j < M; j++) {
            func(t1);
            if (i % 3 == 0)
                    func(t2);
            else
                    func(t3);
   }
}
```

Figure 5.7: A part of test benchmark

is hardly practical to physically replace a capacitor a large number of times, we developed an alternative setup. An external device is in charge of sending a reset signal to our experimental board at preconfigured time intervals [2]. This simulates a power failure followed by a restart when the capacitor is full (we do not take into account the energy harvesting time, which heavily depends on external factors, such as the technology used and ambient conditions). We considered intervals varying from 0.1 s to 2 s. We have chosen an ideal checkpointing strategy as an Oracle to compare SFSG with. For that, we utilized a timer in our experimental board raising a backup signal just before the reset signal being sent from the external device. Our Oracle is very similar to on-demand checkpointing systems [37, 10, 9] which proved to have low checkpointing overhead as they checkpoint immediately before power failure [54]. However, in reality, these systems utilize a hardware voltage monitoring system which has minor impact on the overall performance. This makes our Oracle to be superior to them but infeasible in reality.

5.7.1 Number of checkpoints

The number of taken checkpoints at runtime, for different interval sizes, are shown in Figure 5.8. As it is observed, the number of overall checkpoints decreases when the interval size increases. We can also observe outliers. The reason for that is SFSG is a greedy approach based on a heuristic. A slight variation in interval size may result in a checkpoint placed in different basic blocks, possibly crossing a loop boundary. Local variations are thus expected. Yet, we also observe that these outliers remain rare, showing that our heuristic is fairly robust to changes in interval sizes.

Most checkpoints are able to complete before power failure happens. When power failure happens during the checkpointing operation, we say the checkpoint was unsuccessful.

2. The energy that a capacitor can provides is proportional to the capacitance: $E = \frac{1}{2}C.(V_{max}^2 - V_{min}^2)$ and execution time is directly related to available energy. Each interval size corresponds to a capacitance which can be derived from particular values of the board specifications.

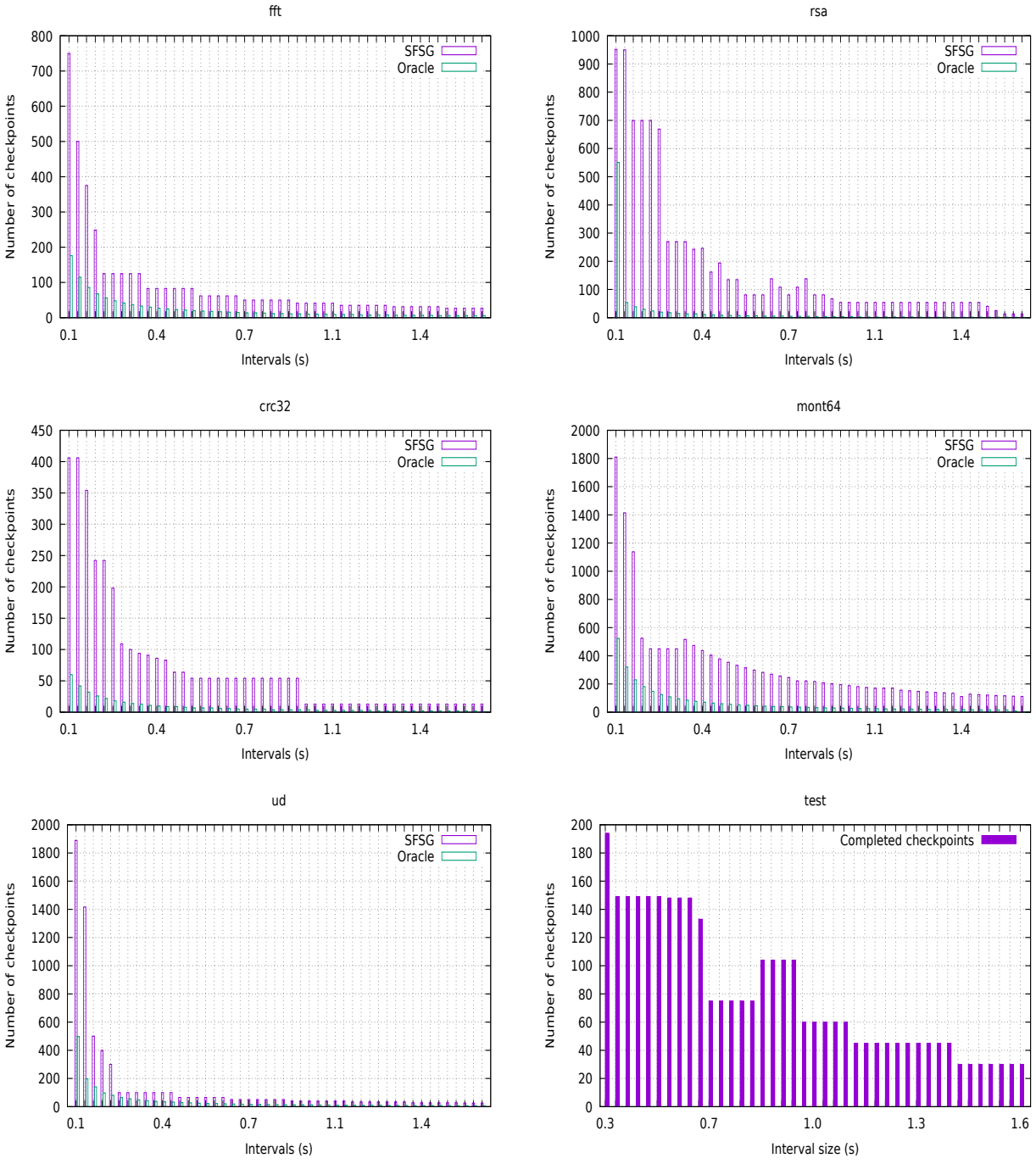


Figure 5.8: Dynamic number of checkpoints taken, for different uninterrupted interval sizes

This case is extremely rare. It occurs once in fft, twice in ud, and five times in mont64, with the smallest capacitor sizes (under 260 ms uninterrupted execution time).

To test that our basic block splitting strategy works correctly, we changed the code of our test benchmark. We called two external functions consecutively. Each of which is responsible for activating a LED for about 160 microseconds and we set the interval of the system to 200 microsecond. SFSG could successfully split the basic block and have forward progress to completion of the benchmark.

5.7.2 Stabilization of checkpoint insertion

Figure 5.9 shows the behavior of benchmarks at runtime, shortly after startup. Two interval sizes are shown (small is 325 ms and large is 2 s). The x-axis represents the number of executed basic blocks. The y-axis represents the static number of checkpoints inserted so far (equivalently, the number of times the system entered the checkpoint management state). The moment the system was considered stable is shown by arrows. As shown, benchmarks take different amounts of time before they stabilize. With longer intervals, the initial performance of the execution suffers from the longer execution paths until failure followed by longer re-execution. However, this is temporary. This penalty is amortized by the fewer number of checkpoints in a long term of execution. In contrast to the longer intervals, shorter intervals go to the stable mode faster, as they suffer from failures earlier. However, they continue their execution with a higher number of checkpoints. As further discussed below (see Section 5.7.3), when the system is collecting a trace, it suffers from the high overhead of writing to the NVM. In stable mode, this overhead is reduced as in each `so_far()`, no writing to NVM happens. However, at each `so_far()` point some extra instructions still need to be executed (such as reading the status from NVM, comparing...). Stabilizing faster means that this overhead is eliminated earlier.

5.7.3 Tracing overhead

Table 5.1 shows the impact of the runtime execution modes compared to Pure C [3]. As expected, trace collection mode has a severe overhead in the execution of the program, degrading performance up to 8× in the worst case. However, the trace collection step is temporary and it is the penalty that the system pays for locating the checkpoints when

3. Note that the performance of *Pure C* is an optimistic upper bound: no checkpoint is inserted, and benchmarks would not survive a power depletion.

Benchmark	Trace collection	Stable mode	minimal overhead
fft	4.0×	2.0×	+9 %
rsa	8.3×	3.4×	+27 %
crc32	5.3×	2.5×	+8 %
ud	3.4×	1.8×	+9 %
aha-mont64	2.1×	1.4×	+2 %
test	1.0×	1.0×	+0 %

Table 5.1: Overhead compared to Pure C.

discovering the code. The overhead of this mode is dependent on the benchmarks and their number and size of basic blocks. For instance, rsa has a large number of basic blocks with the small code size as well as loop nesting. This causes this benchmark to have a high overhead when it is in a tracing mode. In contrast to rsa, test has a small number of basic blocks with two levels of loop nesting. The program is waiting most of the time for pseudo-peripheral operations that take significant time. As a result, collecting traces has a marginal overhead on the execution.

In stable mode, the system does not have the high overhead of the trace collection. However, there is still a function call to `so_far()`, and a few instructions to read a status variable from NVM, compare its value, and return to the caller. This makes the execution still slower than the Pure C, but cuts the overhead of trace collection in half, resulting in slowdowns from 1.4× to 3.4× (excluding test).

To better observe the overhead of `so_far()` function calls when the system is in stable mode, we have replaced every `so_far()` in the code by NOPs. As shown in the last column of Table 5.1, in benchmarks with large number of basic blocks and less time consuming instructions, `so_far()` has non-negligible overhead. As a future research direction to reduce the overhead of `so_far()`, the static part of SFSG can decide not to place `so_far()` on some basic blocks. For instance, those that have a small number of instructions without using any peripherals. However, it must ensure that the system will make forward progress. The runtime system could also decide to dynamically overwrite the calls by NOPs when the system is deemed stable enough, with a rollback mechanism in case tracing must be enabled again.

Benchmark	Pure C	SFSG
fft	3640	8762
rsa	9366	16752
crc32	2800	7556
ud	8188	13616
aha-mont64	8282	13394
test	1076	6414

Table 5.2: Code size in bytes

5.7.4 Code size

In terms of code size, we compare our work with pure code that is generated by clang. The code size of our work includes checkpoint and restore routines from Mementos, `so_far()` routine, our runtime system, traces, and the addresses of basic blocks. Table 5.2 reports the code size, defined as the number of bytes transmitted to the board when we "flash" it. The difference between the size of the code of our work with Pure C code generated by clang is significant in relative value, but is a mere 5 KB on average. The checkpoint and restore routine contribute a large amount of the increase of the device text sections. However, these two routines contain the basic features that a software approach must have in order to be able to survive power failures, and they are similar to other solutions proposed by related work.

We also measured the size of the dynamically objects during execution (traces, trampolines) in the worst case. Thanks to our algorithm and the existence of the loops as well as the benchmarks in IoT domain, it is on average 150 bytes across all benchmarks with different intervals. Our runtime can currently allocate 16 trampolines for each program, however, we also observe that in none of benchmarks we needed all 16 trampolines.

5.8 Discussion and future extensions

5.8.1 Memory protection unit

As far as we know, all commercial MCUs suitable for IoT energy harvesting either do not have any memory protection unit (MPU) or allow the user to disable it. Therefore, SFSG is compatible with these platforms as it requires the MPU to be disabled so that it can modify the protected text section of the program at runtime. This may raise some security issues for the system. Previous work has proposed approaches to overcome those

issues. For instance, Belleville et al. [12] have proposed to rely on control-flow integrity (CFI) techniques in the absence of MPU. SFSG can be extended along these lines to mitigate the aforementioned security issue.

5.8.2 Dealing with closed-source

Some benchmarks depend on MSP430 helper functions, provided by the compiler toolchain, whose source code may not be available. However, we have not experienced any problem in forward progress as those helper functions do not consume a lot of energy and time. In case it happened, our basic block splitting algorithm (see Section 5.6.3) can handle this situation. It is worth to note that it is possible to include open source libraries such as libc to SFSG to provide more robust behavior at runtime.

5.8.3 NVM and memory consistency issue

As mentioned in the related work (Chapter 3), when the programmer allocates data in NVM, the combination of re-executing a code section and checkpointing may cause memory to be inconsistent if the re-executed code section was not idempotent. This is not supported by SFSG. However, previous works [8, 53] proposed applying *undo logging* to protect variables that reside in NVM and used in non-idempotent code sections. SFSG is compatible with those techniques and can be extended. However, for the sake of time, here we assume that all data are allocated in volatile memory.

5.9 Conclusion

We introduced SFSG: an adaptive checkpointing strategy for intermittently powered systems. Our approach makes decisions about locations of the checkpoint routines and specialize them at runtime based on the actual execution of the program in the deployment environment. SFSG automatically discovers the parts of the code that consume more energy than the capacitor of the MCU can provide and inserts checkpoints to guarantee termination. In addition, SFSG requires no extra programming effort as well as no extra hardware support, leaving existing features to the user (ADC, timers). We have also evaluated our work in terms of memory space that it consumes including code and trace size.

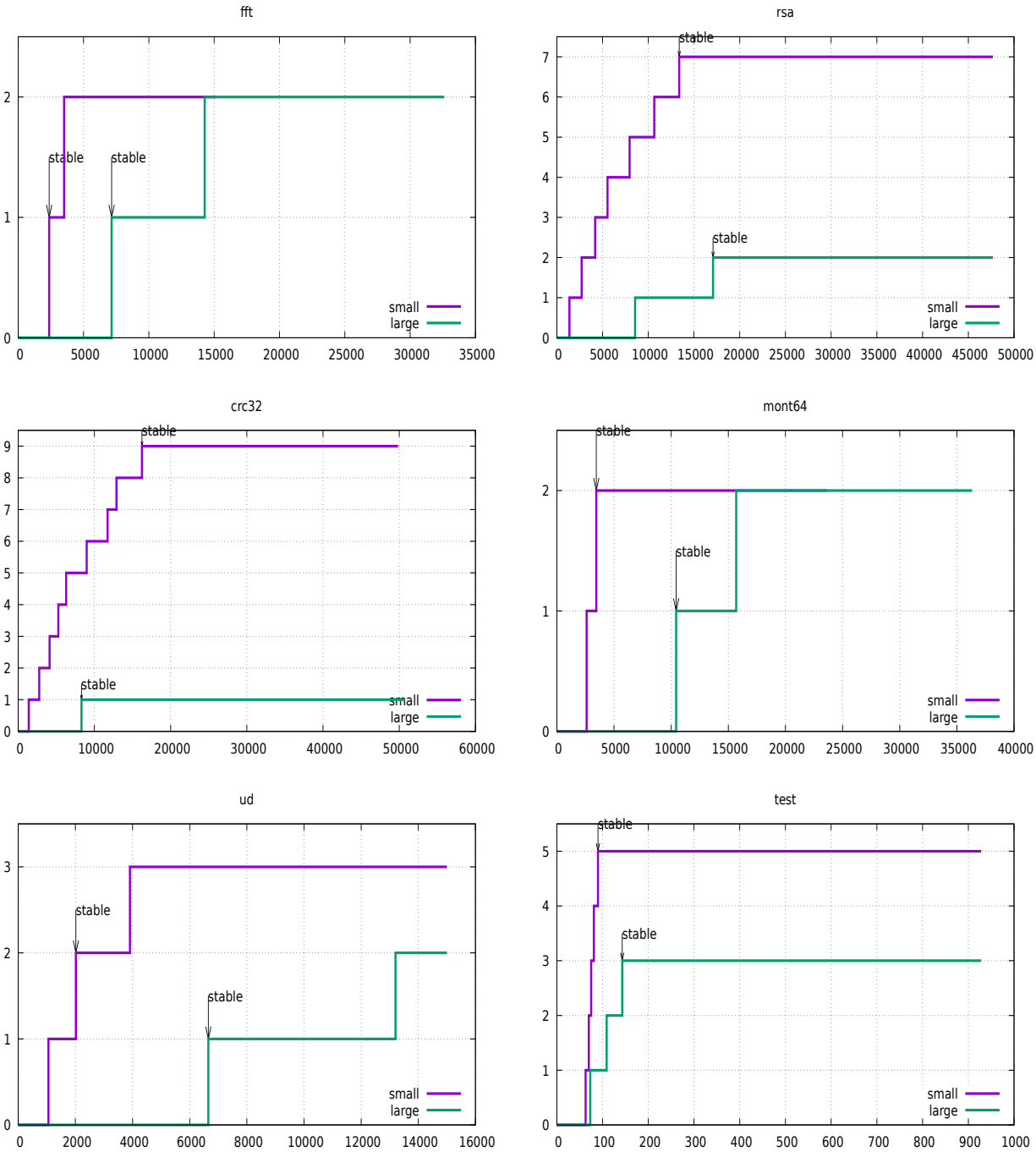


Figure 5.9: Behavior of benchmarks in early execution steps after startups.

Conclusion

With the advent of Internet of things (IoT), there is a need to provide energy for a massive number of tiny smart devices without using large, heavy, and high-maintenance batteries. One promising way is to harvest energy from the environment and store it into a capacitor which is in charge of supplying energy to the device. However, harvested energy sources are all unstable making the execution of programs interrupted by power failures. Recently, different software and hardware based checkpointing strategies have been proposed to make forward progress toward execution for energy harvesting IoT devices. This thesis presents two different fully software approaches based on static and dynamic compilation techniques. The main goal of these solutions is to provide forward progress alongside with a consistent memory without using any hardware feature of the device. The proposed solutions aim to increase portability and the concepts are applicable to any type of commercial and custom energy harvesting device. The proposed solutions also try to limit the burden on programmers.

The static compilation approach is based on the worst-case energy consumption (WCEC). The checkpoint trigger calls are inserted statically by the compiler based on the WCEC. Therefore, the energy consumption between two checkpoints does not exceed the available energy of the capacitor. This can guaranty forward progress and memory consistency as unhandled power failures will not happen during the execution of the program. Also, to reduce the number of checkpoints at runtime, we applied a few selected classical compiler optimizations.

The presented dynamic compilation approach called SFSG and it is based on self-modifying code (SMC). In SFSG, the checkpoint trigger calls are instrumented into the code at runtime. SFSG learns from past power failures and instruments and specializes checkpoints based on them. For finding the point where the power failure happened, SFSG utilizes a novel tracing methodology with moderate overhead in terms of time and space. SFSG can provide forward progress without facing non-termination. For that, it breaks basic blocks that consume more energy consumption than the maximum energy provided by the capacitor. SFSG is portable and its concept can be applied in different commercial and custom energy harvesting devices. Also, as mentioned earlier, SFSG is extendable to

overcome the problem of memory consistency when data are allocated in both volatile and non-volatile memory.

Comparison between two proposed solutions

As mentioned previously, to guarantee forward progress without facing non-termination, together with avoiding unnecessary checkpoints, it mandates having a safe and tight bound of the energy consumption of a program on the particular micro-controller (MCU). The energy model that we used for our static checkpoint placer can satisfy these properties for a simple processor. However, for more complex processors, finding a safe and tight bound for energy consumption is NP-hard [57]. Also, the energy consumption is highly dependent on unknown factors such as temperature of the deployment environment. Capacitors are also known to age [43], losing capacity over time makes the proposed static work doom to fail when the device grows older. Furthermore, the proposed work is not fully transparent. The programmer must provide loop-bound information which is not always easy. Inaccurate loop-bound information makes the system under-optimized or at risk of facing non-termination.

On the other hand, the proposed dynamic approach is fully transparent which means that the programmer does not need to provide any extra information to the system. It is also energy consumption agnostic and adapts when the capacitor of the system grows older. Yet one problem with this approach is the overhead of re-execution in the early stages of the program execution.

Publications

SCOPES

Bahram Yarahmadi and Erven Rohou. **"So Far So Good Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code."**, 2021

SAMOS [79]

Bahram Yarahmadi and Erven Rohou. **"Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems."**, 2020.

COMPAS [80]

Bahram Yarahmadi and Erven Rohou. **"Worst-Case Energy Consumption Aware Compile-Time Checkpoint Placement for Energy Harvesting Systems."**, 2019

Further Extension

As mentioned earlier, re-executing sections of code may lead to inconsistency issues and incorrect behavior if special care is not taken. Re-executing some critical I/O operations (e.g. writing to a device, sending a radio packet) modifies the semantics of the program. For instance, this may alter the protocol in a network of IoT devices when packets are sent multiple times. To handle this situation in our dynamic approach, we envision extending SFSG by using annotations. A programmer can specify regions containing those critical operations that must be executed only once. The compiler then statically inserts checkpoints before and after that operation. As a result, just before the critical operation, the system takes a checkpoint. It executes the critical operation when the capacitor becomes fully charged and takes a checkpoint just after the execution of the critical operation. In this way, the critical operation is executed once and if a power failure happens after the execution of the operation, the system will roll back to the checkpoint location after the operation and will not re-execute the operation again. Also, some code regions must be executed atomically, without any checkpointing activity in between. For instance, sensing data and sending it with timestamps. Checkpointing between sensing and sending rises a timing inconsistency issue [32]. The difference between the timing inconsistency problem and the previous problem about the critical I/O operations is that in the timing inconsistency problem the re-execution is allowed but the operations must be executed atomically. Inspired by the approach proposed by Maeng and Lucia [53], we can again resort to annotations. A programmer can annotate sections of code that need to be executed atomically, and the static compiler does not place `so_far()` trigger call in basic blocks that are involved in that section of code. These blocks are thus invisible to our runtime, and checkpoints cannot be inserted in them. Identifying the set of basic blocks involved can be performed by a DFS-like algorithm [16] for the basic block with the start annotation to the basic block specifying the end of region.

The energy consumption of the checkpoint routine itself has not been taken into account in the proposed static compiler. Sometimes it is better to take a checkpoint earlier to reduce the overhead of checkpoint. For instance, placing a checkpoint before a function trigger call reduces the amount of data that needs to be copied into NVM as before the function call the program stack is smaller.

Our static compiler applies a limited number of classical compiler optimizations separately to reduce the number of checkpoints. However, nowadays modern compilers are applying tons of optimizations to improve performance, energy consumption and code

size. Also, it has been investigated that different sequences of optimizations result in significant performance variation. Therefore, people have been researching to find the optimum sequence of optimizations. This problem is known as the phase ordering problem and researchers have proposed different techniques such as iterative compilation [14] and machine learning [26] to answer this problem. It will be interesting to support more optimizations in our static WCEC checkpoint placement work and find the best sequence that can lead to fewer checkpoints at runtime. However, the first step to enrich our WCEC checkpoint placer with more compiler optimizations is to transform automatically the flow information in the source code such as loop bound information down to the binary code. Li et al. [46] have done it to some extent for WCET estimation. Our work can be integrated with this work to propose more opportunities for compiler optimizations. Also, the proposed static WCEC based checkpoint placer does not take into account the energy consumption of peripherals in the device. This is one of the limitations of this work as usually an IoT device is made to interact with the environment through these peripherals. The reason behind this limitation is the simple ISA energy model that we used. By providing an accurate energy model of device peripherals, it is possible to extend the current work to support programs that use peripherals.

In this thesis, the static and dynamic work have been presented separately. However, it is possible to combine them to benefit from the best of both worlds as well as to eliminate the disadvantages. For instance, one way to reduce the re-execution overhead in SFSG which is a part of its learning phase is by placing checkpoints statically by the WCEC checkpoint placer. In this way, for large energy buffers and capacitors, the system will not pay a lot of re-execution penalties. Also, one of the negative points of the static WCEC checkpoint placer is that it requires to have a safe and accurate energy model which may not be available. Lack of safety in energy estimation may result in facing non-termination. Herein the dynamic compiler can help to provide forward progress by instrumenting the checkpoint trigger calls when the system does not make forward progress. Also, for over-pessimistic placed checkpoints, the dynamic compiler, can decide whether to take or not to take the static checkpoints and it can gradually omit unnecessary checkpoints when the system is stabilized.

The current thesis focuses on overcoming the problems in energy harvesting systems without using any hardware feature. However, both proposed work can be improved with hardware facilities. For instance, to reduce the tracing in SFSG, hardware can alert the runtime system to trace the execution when the energy outage is close. Also, the WCEC

checkpoint placer can skip checkpoints if it can be informed about the available energy at the checkpoint trigger calls. This information can be provided by a hardware entity of the MCU.

Appendix A

SFSG main function

The `main` function of SFSG is the entry point of the execution after each power failure or when the program starts its execution for the first time. Each macro in the code is an integer number specifying a system state or future or past activities of the system. In the beginning, depending of the last state of the system before a power failure, SFSG changes the state of the system. For instance, if the system was running in the stable mode before power failure, it changes the system state to `DoNotCollectTrace` and the system will not collect any execution trace (lines 2 and 3). If the system was not collecting a trace before power failure, SFSG changes the state of the system to the trace collection (lines 4 and 5). If there is a trace in the system (line 6), it shows that the system was in the trace collection mode and SFSG places a checkpoint based on the collected trace (line 7). If the system was in the trace collection mode, and there is not a collected trace in the system, it shows that either the system starts its execution for the first time or the system was facing non-termination in its execution and could not reach the next basic block. For the former, SFSG changes the value of a global non-volatile variable to show that the program starts its execution. For the latter SFSG calls a routine that is responsible to split the last uncompleted executed basic block (lines 8 to 11). Finally, the system looks to find an available checkpoint in the system. If there is an available checkpoint, SFSG calls restore routine used in mementos to resume the execution from there. Otherwise, SFSG executes the program from the beginning (lines 12 to 15). It is worth mentioning that `original_main` refers to the main function of the program that is executing.

Checkpoint routine

As mentioned before, the checkpoint routine of Mementos [5] is used in SFSG. Mementos checkpoint routine copies registers, global data and the stack into the NVM. For our purpose, we added some code to save SFSG loop information in non-volatile memory with checkpoint data. Also, when all data are completely copied into the NVM, depending on the current state of the system, SFSG decides to keep or change the state for specifying

```
1  void main() {
2     if (System.state == Stable)
3         System.state == DoNotCollectTrace;
4      else if (System.state == DoNotCollectTrace)
5         System.state = CollectTrace;
6      else if (TraceSize != Zero)
7           placeCheckpoint();
8      else if (TraceSize == Zero && FirstExecution == Zero)
9          FirstExecution = One;
10      else
11          splitBB();
12     if (lookForAvailableCheckpoint())
13        mementos_restore();
14      else
15         original_main();
16 }
```

the future activity of the system. For instance, if the system has been stabilized before, SFSG changes the state of the system to stable (line 5) which makes the system not to collect any trace after power failure (line 2 and 3 in SFSG main function). Otherwise, SFSG changes the state to `DoNotCollectTrace` (line 7) making the system to collect a trace after power failure (line 4 and 5 in SFSG main function).

```
1  void checkpoint() {
2     /* mementos checkpoint routine */
3     /* saving non-volatile loop information */
4     if (is_stable())
5        System.state = Stable;
6      else
7        System.state = DoNotCollectTrace;
8  }
```

SFSG splitBB routine

The function `splitBB` is used to split a basic block at runtime when the basic block makes the system facing non-termination. For splitting a basic block, the beginning address and the end address of the basic block are needed. The function `findLastCheckpointLoc()` returns the beginning address of the basic block. Herein the beginning address is the last checkpoint location that the program had resumed its execution from there. The end of the basic block can be found in a very straightforward way as the list of beginning addresses and end addresses had been provided by the static part of SFSG. From the

```
1  void splitBB() {
2     unsigned int beginAddr = findLastCheckpointLoc();
3     unsigned int endAddr = findEndAddrOfBB(beginAddr);
4     unsigned int middleAddr = beginAddr + (endAddr - beginAddr) / 2;
5     unsigned int instAddr = beginAddr;
6     unsigned int instSize;
7     do {
8       instSize = decode(instAddr);
9       instAddr += instSize;
10     } while (instAddr < middleAddr);
11     breakBB(instAddr);
12     editBBList(beginAddr, instAddr);
13 }
```

beginning address and the end address, the middle address is computed (line 4). As mentioned in Chapter 5, for the fixed-length ISAs, this is the point where the basic block must be broken into two parts. However, the size of instructions in MSP430 is variable. To find a proper breaking point, `decode` function identifies the size of each instruction from the beginAddr until it reaches the middleAddr. The length of an instruction in MSP430 is either 2 or 4 or 6 bytes. After finding the proper breaking point, the function `breakBB` (line 11), breaks the basic block into two parts as discussed in Chapter 5. Finally, the function `editBBList` updates the list of beginning addresses and the end addresses.

SFSG trampoline creation routine

The function `createTrampoline` creates a trampoline in a reserved area in NVM. The generated trampoline provides the number of iterations for the function `hook_checkpoint`. Herein, `checkpoint_loc` is the location where the trampoline must be inserted. Also, the function `locateFreeTrampoline` is responsible for finding a free area from the reserved area. Right now, the system can provide 16 trampolines based on the mask variable used in the `locateFreeTrampoline` function.

```
#define NOP 0x4303u
#define PUSH_R12 0x120Cu;
#define POP_R12 0x413Cu
#define CALL 0x12B0u
#define MOV 0x403Cu
#define BR 0x4030u

void createTrampoline(unsigned int *checkpoint_loc,
                                  unsigned int iteration) {

    *checkpoint_loc = BR;
    unsigned int *trampoline_loc = locateFreeTrampoline();
    *(checkpoint_loc + 2) = trampoline_loc;
    *trampoline_loc = PUSH_R12;
    *(trampoline_loc + 1) = MOV;
    *(trampoline_loc + 2) = iteration;
    *(trampoline_loc + 3) = CALL;
    *(trampoline_loc + 4) = &hook_checkpoint;
    *(trampoline_loc + 5) = POP_R12;
    *(trampoline_loc + 6) = BR;
    *(trampoline_loc + 7) = checkpoint_loc + 4;
}
```

```
void hook_checkpoint(unsigned int iteration)
  int loop_iter = getLoopIteration();
  if (loop_iter % iteration == 0) {
     checkpoint();
   }
}
```

```
1
2  unsigned int locateFreeTrampoline() {
3    unsigned int *addr = RESERVED_AREA;
4    unsigned int bitVect = *addr;
5    unsigned int mask = 1;
6    unsigned int *blockAddr = RESERVED_AREA + 2;
7    *blockId = 0;
8    for (int i = 1; i <= 16; i++) {
9      if (bitVect & mask) {
10       *blockId = i;
11       *addr = bitVect & (~mask);
12       break;
13     } else {
14       mask = mask << 1;
15     }
16   }
17   if (*blockId) {
18     blockAddr = (RESERVED_AREA + 2) + (*blockId - 1) * TRAMPOLINE_SIZE;
19   }
20   return blockAddr;
21 }
```

BIBLIOGRAPHY

[1] https://www.absint.com/ait/.

[2] http://www.bound-t.com/.

[3] https://www.comp.nus.edu.sg/~rpembed/chronos/.

[4] http://www.otawa.fr/.

[5] https://github.com/CMUAbstract/mementos.

[6] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. "Cost relation systems: A language-independent target language for cost analysis". In: *Electronic Notes in Theoretical Computer Science* 248 (2009), pp. 31–46.

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300.

[8] Sara S Baghsorkhi and Christos Margiolas. "Automating efficient variable-grained resiliency for low-power IoT systems". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 38–49.

[9] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.12 (2016), pp. 1968–1980.

[10] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems". In: *IEEE Embedded Systems Letters* 7.1 (2014), pp. 15–18.

[11] SP Beeby, RN Torah, and MJ Tudor. "Kinetic energy harvesting". In: (2008).

[12] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. "Automated software protection for the masses against side-channel attacks". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.4 (2018), pp. 1–27.

[13] Naveed Anwar Bhatti and Luca Mottola. "HarvOS: Efficient code instrumentation for transiently-powered embedded sensing". In: *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE. 2017, pp. 209–220.

[14] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. "Iterative compilation in a non-linear optimisation space". In: *Workshop on Profile and Feedback-Directed Compilation*. 1998.

[15] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. "Emerging NVM: A survey on architectural integration and research challenges". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23.2 (2017), pp. 1–32.

[16] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. "Energy-efficient memory mappings based on partial WCET analysis and multi-retention time STT-RAM". In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems*. 2018, pp. 148–158.

[17] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. "Partial Worst-Case Execution Time Analysis". In: *Conférence d'informatique en Parallélisme, Architecture et Système*. 2018.

[18] Michael Buettner, Richa Prasad, Alanson Sample, Daniel Yeager, Ben Greenstein, Joshua R Smith, and David Wetherall. "RFID sensor networks with the Intel WISP". In: *Proceedings of the 6th ACM conference on Embedded network sensor systems*. 2008, pp. 393–394.

[19] Andrew Burgess, Ashley Whetter, George Field, Graham Markall, Hendrik Oosenbrug, James Pallister, Jeremy Bennett, Neville Grech Pierre Langlois, and Simon Cook. *Embench™: A Modern Embedded Benchmark Suite*. `https://embench.org`.

[20] Chaitali Chakrabarti and Dinesh Gaitonde. "Instruction level power model of microcontrollers". In: *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 1. IEEE. 1999, pp. 76–79.

[21] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. “Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019.

[22] Albert Cohen and Erven Rohou. “Processor virtualization and split compilation for heterogeneous multicore embedded systems”. In: *Design Automation Conference*. IEEE. 2010, pp. 102–107.

[23] Alexei Colin and Brandon Lucia. “Chain: tasks and channels for reliable intermittent programs”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016, pp. 514–530.

[24] Alexei Colin and Brandon Lucia. “Termination checking and task decomposition for task-based intermittent programs”. In: *Proceedings of the 27th International Conference on Compiler Construction*. 2018, pp. 116–127.

[25] GR Fox, F Chu, and T Davenport. “Current and future ferroelectric nonvolatile memory technology”. In: *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures Processing, Measurement, and Phenomena* 19.5 (2001), pp. 1967–1971.

[26] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. “MILEPOST GCC: machine learning based research compiler”. In: *GCC summit*. 2008.

[27] Kyriakos Georgiou, Samuel Xavier-de-Souza, and Kerstin Eder. “The IoT energy challenge: A software perspective”. In: *IEEE Embedded Systems Letters* 10.3 (2017), pp. 53–56.

[28] Kyriakos Georgiou, Samuel Xavier-de-Souza, and Kerstin Eder. “The IoT energy challenge: A software perspective”. In: *IEEE Embedded Systems Letters* 10.3 (2018).

[29] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. “Static analysis of energy consumption for LLVM IR programs”. In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. 2015, pp. 12–21.

[30] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. "The Mälardalen WCET benchmarks: Past, present and future". In: *Intl. Workshop on Worst-Case Execution Time Analysis*. 2010.

[31] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. "The Heptane static worst-case execution time estimation tool". In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*. 2017, 8:12–8:12.

[32] Josiah Hester, Kevin Storer, and Jacob Sorber. "Timely execution on intermittently powered batteryless sensors". In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 2017, pp. 1–13.

[33] Matthew Hicks. "Clank: Architectural support for intermittent computation". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 228–240.

[34] Texas Instruments. *MSP430FR59694*. `https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf`.

[35] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. "QuickRecall: A HW/SW approach for computing across power cycles in transiently powered computers". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 12.1 (2015), pp. 1–19.

[36] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in IoT edge devices". In: *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. IEEE. 2016, pp. 264–269.

[37] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. "QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers". In: *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE. 2014, pp. 330–335.

[38] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. "Estimating the worst-case energy consumption of embedded software". In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE. 2006, pp. 81–90.

[39] Richard Johnson, David Pearson, and Keshav Pingali. "The program structure tree: Computing control regions in linear time". In: *ACM SigPlan Notices*. Vol. 29. 6. ACM. 1994.

[40] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B Srivastava. “Power management in energy harvesting sensor networks”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 6.4 (2007).

[41] Steve Kerrison and Kerstin Eder. “Energy modeling of software for a hardware multithreaded embedded microprocessor”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 14.3 (2015), pp. 1–25.

[42] Richard Koo and Sam Toueg. “Checkpointing and rollback-recovery for distributed systems”. In: *IEEE Transactions on software Engineering* 1 (1987), pp. 23–31.

[43] Paul Kreczanik, Pascal Venet, Alaa Hijazi, and Guy Clerc. “Study of supercapacitor aging and lifetime estimation according to voltage, temperature, and RMS current”. In: *IEEE Transactions on Industrial Electronics* 61.9 (2013), pp. 4895–4902.

[44] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[45] Fuyang Li, Keni Qiu, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yong Guan, and Chun Jason Xue. “Checkpointing-aware loop tiling for energy harvesting powered nonvolatile processors”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.1 (2018), pp. 15–28.

[46] Hanbing Li, Isabelle Puaut, and Erven Rohou. “Traceability of flow information: Reconciling compiler optimizations and WCET estimation”. In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. 2014, pp. 97–106.

[47] Hanbing Li, Isabelle Puaut, and Erven Rohou. “Tracing flow information for tighter WCET estimation: Application to vectorization”. In: *RTCSA*. IEEE. 2015.

[48] Yau-Tsun Steven Li and Sharad Malik. “Performance analysis of embedded software using implicit path enumeration”. In: *ACM SIGPLAN Notices*. Vol. 30. 11. ACM. 1995.

[49] Qingrui Liu and Changhee Jung. “Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems”. In: *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE. 2016, pp. 1–6.

[50] Xin Lu and Shuang-Hua Yang. “Thermal energy harvesting for WSNs”. In: *2010 IEEE International Conference on Systems, Man and Cybernetics*. IEEE. 2010, pp. 3045–3052.

[51] Brandon Lucia and Benjamin Ransford. “A simpler, safer programming and execution model for intermittent systems”. In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 575–585.

[52] Kiwan Maeng, Alexei Colin, and Brandon Lucia. “Alpaca: intermittent execution without checkpoints”. In: *arXiv preprint arXiv:1909.06951* (2019).

[53] Kiwan Maeng and Brandon Lucia. “Adaptive dynamic checkpointing for safe efficient intermittent computing”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 129–144.

[54] Kiwan Maeng and Brandon Lucia. “Supporting peripherals in intermittent systems with just-in-time checkpoints”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 1101–1116.

[55] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. “Dynamic task-based intermittent execution for energy-harvesting devices”. In: *ACM Transactions on Sensor Networks (TOSN)* 16.1 (2020), pp. 1–24.

[56] Yoshio Masubuchi, Satoshi Hoshina, Tomofumi Shimada, B Hirayama, and Nobuhiro Kato. “Fault recovery mechanism for multiprocessor servers”. In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE. 1997, pp. 184–193.

[57] Jeremy Morse, Steve Kerrison, and Kerstin Eder. “On the limitations of analyzing worst-case dynamic energy of processing”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 17.3 (2018), pp. 1–22.

[58] Davide Pala, Ivan Miro-Panades, and Olivier Sentieys. “Freezer: A Specialized NVM Backup Controller for Intermittently-Powered Systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[59] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. “Data dependent energy modeling for worst case energy consumption analysis”. In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. 2017, pp. 51–59.

[60] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. “Design considerations for solar energy harvesting wireless embedded systems”. In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE. 2005, pp. 457–462.

[61] Benjamin Ransford and Brandon Lucia. “Nonvolatile memory is a broken time machine”. In: *Proceedings of the workshop on Memory Systems Performance and Correctness*. 2014, pp. 1–3.

[62] Benjamin Ransford, Jacob Sorber, and Kevin Fu. “Mementos: system support for long-running computation on RFID-scale devices”. In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 159–170.

[63] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. “T-CREST: Time-predictable multi-core architecture for embedded systems”. In: *Journal of Systems Architecture* 61.9 (2015), pp. 449–471.

[64] Volkmar Sieh, Robert Burlacu, Timo Hönig, Heiko Janker, Phillip Raffeck, Peter Wägemann, and Wolfgang Schröder-Preikschat. “An end-to-end toolchain: From automated cost modeling to static WCET and WCEC analysis”. In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2017, pp. 158–167.

[65] Volkmar Sieh, Robert Burlacu, Timo Hönig, Heiko Janker, Phillip Raffeck, Peter Wägemann, and Wolfgang Schröder-Preikschat. “Combining Automated Measurement-Based Cost Modeling With Static Worst-Case Execution-Time and Energy-Consumption Analyses”. In: *IEEE Embedded Systems Letters* 11.2 (2018), pp. 38–41.

[66] Priyanka Singla, Shubhankar Suman Singh, and Smruti R Sarangi. “Flexicheck: An adaptive checkpointing architecture for energy harvesting devices”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 546–551.

[67] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. “Power analysis of embedded software: a first step towards software power minimization”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2.4 (1994), pp. 437–445.

[68] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. “Instruction level power analysis and optimization of software”. In: *Technologies for wireless computing*. Springer, 1996, pp. 139–154.

[69] Joel Van Der Woude and Matthew Hicks. “Intermittent computation without hardware support or programmer intervention”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 17–32.

[70] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. “Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems”. In: *Leibniz International Proceedings in Informatics, LIPIcs 106 (2018)* 106 (2018), p. 24.

[71] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. “Worst-case energy consumption analysis for energy-constrained embedded systems”. In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 105–114.

[72] Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, and Volkmar Sieh. “A kernel for energy-neutral real-time systems with mixed criticalities”. In: *RTAS*. IEEE. 2016.

[73] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. “A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops”. In: *2012 Proceedings of the ESSCIRC*. IEEE. 2012, pp. 149–152.

[74] Chongfeng Wei and Xingjian Jing. “A comprehensive review on vibration energy harvesting: Modelling and realization”. In: *Renewable and Sustainable Energy Reviews* 74 (2017), pp. 1–18.

[75] Michael A Weimer, Thurein S Paing, and Regan A Zane. “Remote area wind energy harvesting for low-power autonomous sensors”. In: *2006 37th IEEE Power Electronics Specialists Conference*. IEEE. 2006, pp. 1–5.

[76] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. “The worst-case execution-time problem—overview of methods and survey of tools”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53.

[77] Kun-Lung Wu, W. Kent Fuchs, and Janak H. Patel. “Error recovery in shared memory multiprocessors using private caches”. In: *IEEE Computer Architecture Letters* 1.02 (1990), pp. 231–240.

[78] Mimi Xie, Mengying Zhao, Chen Pan, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. “Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor”. In: *Proceedings of the 52nd Annual Design Automation Conference*. 2015, pp. 1–6.

[79] Bahram Yarahmadi and Erven Rohou. “Compiler Optimizations for Safe Insertion of Checkpoints in Intermittently Powered Systems”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Springer’s Lecture Notes in Computer Science – LNCS. Samos (virtual), Greece, July 2020. URL: `https://hal.inria.fr/hal-02914953`.

[80] Bahram Yarahmadi and Erven Rohou. “Worst-Case Energy Consumption Aware Compile-Time Checkpoint Placement for Energy Harvesting Systems”. In: *COMPAS19-Conférence d’informatique en Parallélisme, Architecture et Système*. 2019, p. 11.

[81] Wing-kei Yu, Shantanu Rajwade, Sung-En Wang, Bob Lian, G Edward Suh, and Edwin Kan. “A non-volatile microcontroller with integrated floating-gate transistors”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2011, pp. 75–80.

[82] Mengying Zhao, Qingan Li, Mimi Xie, Yongpan Liu, Jingtong Hu, and Chun Jason Xue. “Software assisted non-volatile register reduction for energy harvesting based cyber-physical system”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 567–572.

[83] Mengying Zhao, Keni Qiu, Yuan Xie, Jingtong Hu, and Chun Jason Xue. “Redesigning software and systems for non-volatile processors on self-powered devices”. In: *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2016, pp. 1–6.

[84] Adamu Murtala Zungeru, Li-Minn Ang, S Prabaharan, and Kah Phooi Seng. “Radio frequency energy harvesting and management for wireless sensor networks”. In: *Green mobile devices and networks: Energy optimization and scavenging techniques*. 13. CRC Press New York, NY, USA, 2012, pp. 341–368.

List of Figures

List of Tables

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Support des compilateurs statiques et dynamiques pour les systèmes informatiques alimentés par intermittence

Mot clés : Exécution intermittente, analyse statique, point de contrôle, compilation dynamique

Résumé : Récemment, différentes stratégies de checkpointing basées sur le logiciel et le matériel ont été proposées pour avancer vers l'exécution pour les dispositifs IoT de récolte d'énergie. Cette thèse présente deux solutions logicielles différentes basées sur la compilation statique et dynamique. Le compilateur statique proposé insère des points de contrôle basés sur la pire consommation d'énergie des sections de code calculée de manière statique. En outre, il applique les optimisations classiques du compilateur afin de réduire le nombre de points de contrôle requis à l'exécution. La technique de compilation dynamique proposée reporte le placement et la spécialisation des points de contrôle au moment de l'exécution et prend des décisions en fonction des pannes de courant passées et des chemins d'exécution empruntés avant chaque panne de courant. Les deux solutions proposées garantissent une progression vers l'avant ainsi que le maintien de la cohérence de la mémoire. En outre, elles visent à accroître la portabilité en n'utilisant aucune caractéristique matérielle des systèmes IoT. En outre, elles sont transparentes pour le programmeur.

Title: Static and dynamic compiler support for intermittently powered computer systems

Keywords: Intermittent Execution, Static Analysis, Checkpoint, Dynamic Compilation

Abstract: With the advent of Internet of things (IoT), there is a need to provide energy for a massive number of smart tiny devices without using large, heavy, and high maintenance batteries. One promising way is to harvest energy from the environment and store it into an energy buffer such as a capacitor. In this way, programs are being executed as long as there is available energy in the capacitor, and crash when it exhausts. Recently, different software and hardware based checkpointing strategies have been proposed to make forward progress toward execution for energy harvesting IoT devices. This thesis introduces two different software solutions based on static and dynamic compilation. The proposed static compiler inserts checkpoints based on statically-computed worst-case energy consumption of code sections. Moreover, it applies classical compiler optimizations in order to decrease the required number of checkpoints at runtime. The proposed dynamic compilation technique delays checkpoint placement and specialization to the runtime and takes decisions based on the past power failures and execution paths taken before each power failure. Both proposed solutions guarantee making forward progress as well as keeping the memory consistent. Furthermore, they aim to increase portability by not using any hardware feature of the IoT device. In addition, they are transparent to the programmer.