Über unterbrechende Ausführung

Ein Vergleich von Ansätzen zum Verhindern von durch Unterbrechung eingeführten Fehlern

Colin A. Voigt
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Deutschland

ZUSAMMENFASSUNG

Das batterielose Internet der Dinge (**IoT**) ist nicht mehr die Zukunft sondern Realität. Damit diese Realität verständlicher und effizienter wird, gibt diese Arbeit einen kurzen Überblick über das Internet der Dinge und behandelt die unterschiedlichen Fehler, welche mit der Ausführung von Programmen auf batterielosen Geräten einhergehen. Es werden drei unterschiedliche Möglichkeiten, solche Fehler auszuschließen vorgestellt und miteinander verglichen. Dabei soll keine 'Kaufempfehlung' ausgesprochen werden, es soll ein Bewusstsein geschaffen werden, dass jeder Ansatz andere Aspekte priorisiert und somit sich somit ihre Anwendungsbereiche unterscheiden.

1 DAS INTERNET DER DINGE UND ENERGIEGEWINNENDE GERÄTE

Das Internet der Dinge, häufig auch mit **IoT** (**I**nternet **o**f **T**hings) abgekürzt, bezeichnet die Idee, dass Geräte des Alltags selbständig miteinander kommunizieren können. Den Nutzen, welchen man daraus ziehen möchte ist, dass das Leben der Nutzer einfacher und sicherer wird. Ein Beispiel, das jetzt schon Realität ist, sind Überwachungskameras, die ihren Besitzer benachrichtigen, wenn sie verdächtige Bewegungen aufnehmen. Ein weiteres sind Lampen, die sich basierend auf dem Verhaltensmuster ihrer Nutzer ein- oder aus- schalten und auch Lichtfarbe, sowie -temperatur, anpassen können.
Eins haben diese Beispiele jedoch gemeinsam, sie sind gebunden an Batterien oder externe Stromversorgung. Dies limitiert sowohl Laufzeit, als auch Anwendungsmöglichkeiten. Eine Batterie schränkt ein wie klein das Gerät werden kann, außerdem wirft dies das Problem auf wie man diese wechselt. Eine feste Stromversorgung beschränkt die Nutzung auf stationäre Anwendungen. Um diese Limitationen aufzuheben gibt es energiegewinnende Geräte.

1.1 Energiegewinnende Geräte

Energiegewinnende, batterielose Geräte nutzen keine konventionellen Stromquellen. Sie erzeugen ihren Strom selbst indem sie die Quellen ihres direktes Umfeld nutzen, zu diesen gehören: Vibrationen, Hochfrequenzstrahlung, Wärme, sowie Solarenergie. Wenn man betrachtet, wie sich der Energieausgang bei unterschiedlichen Quellen verhält, so stellt man fest, dass man zwei Typen unterscheiden kann [12]. Die einen verhalten sich auf einer Skala von Millisekunden konstant [7, 11], dazu gehört unter anderem Sonnenenergie. Dem anderen Typ angehörig sind auch Vibrationen, diese folgen grob einem sinusförmigen Muster [1].

Dank des potentiell kleineren Formfaktors und der Unabhängigkeit von einer dedizierten Stromquelle erschließen sich neue Anwendungsmöglichkeiten in diversen Bereichen [1]. Jedoch entstehen auch neue Probleme. So kann der nötige Strom während der Ausführung unvorhersehbar nicht mehr ausreichen. Heutige Systeme erfordern jedoch, dass ein konsistenter Zustand erreicht wird, um einen Fortschritt zu garantieren [9]. Die Ausführung muss also, auch wenn sie unterbrochen wird, konsistente Zustände erreichen. Da normaler RAM volatil ist, also bei Stromverlust auch die Daten verliert, nutzen aktuelle Systeme in der Regel eine Mischung aus volatilem SRAM und nicht-volatilen FRAM [9]. Bei unterbrochener Ausführung erzeugt dies jedoch neue Probleme.

1.2 Fehler durch unterbrochene Ausführung

Damit bei Stromverlust der Datenverlust möglichst minimiert wird, kann ein Kontrollpunkt-System eingeführt werden, dass an kritischen Stellen eine Sicherung des aktuellen Zustands durchführt, dies kann jedoch weitere Fehler verursachen. Bei der Identifizierung und Klassifizierung der möglichen Fehler verlassen wir uns auf wesentliche Einblicke über die unterbrochene Ausführung [9]. Dabei können drei Fehlerarten unterschieden werden.

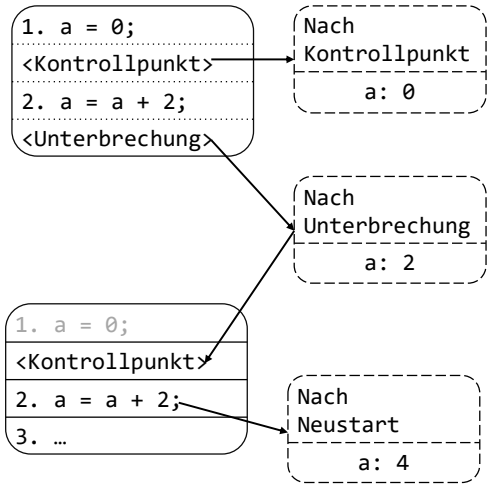


Abbildung 1: Datenzugrifffehler: Durch nicht idempotente Operation wird zu häufig inkrementiert

Eine Art ist der *Datenzugrifffehler* [9]. Er tritt auf, wenn nach einem Kontrollpunkt auf nicht-volatilen Speicher (NVS) eine nicht idempotente Operation ausgeführt wird. Ein einfaches Beispiel ist ein Zähler. Zunächst wird das Datum im NVS gelesen, dann inkrementiert und wieder zurück in den persistenten Speicher geschrieben. Nach dem Schreiben existiert kein Kontrollpunkt. Nun tritt eine

Unterbrechung auf und die Ausführung wird ab dem Kontrollpunkt vor Lesen des Datums wieder aufgenommen. Der nun gelesene Wert ist der bereits inkrementierte Wert, somit ist eine Dateninkonsistenz vorhanden. Im Falle der Abbildung 1 erhält die Variable *a* so den Wert 4 statt 2.

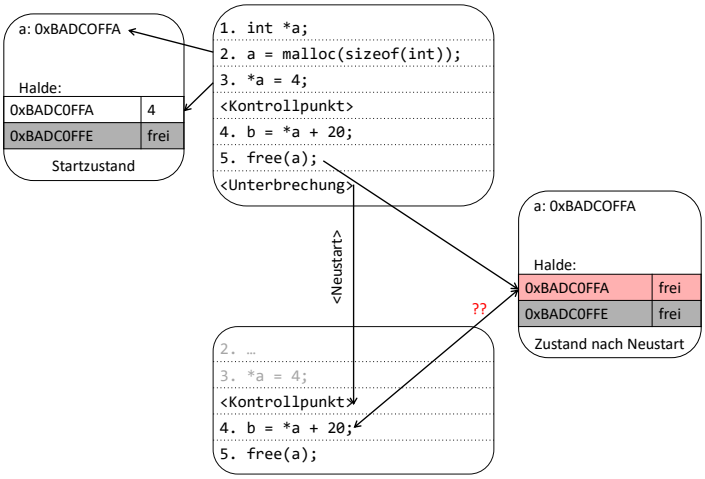


Abbildung 2: Speicherabbild-Fehler: Inkonsistente Zeiger erzeugen Aufruf von nicht allokiertem Speicher

Eine weitere Fehlerart betrifft das Freigeben und Zuteilen von Speicher. Angenommen das Programm bekommt einen Teil der Halde zugewiesen direkt nach der Zuweisung existiert ein Kontrollpunkt. Dieser wird durchlaufen, danach werden Operationen auf dem zugewiesen Speicher durchgeführt und dann wird der Haldenplatz wieder freigegeben. Zwischen den einzelnen Operationen und der Freigabe des Speichers existieren keine weiteren Kontrollpunkte. Wenn die Ausführung nun nach Freigabe des Speichers unterbrochen wird, so startet die Ausführung wieder nach der Zuweisung des Speichers. Die Operationen schlagen dann aber fehl, da der Speicher bereits freigegeben worden war. Diese Art von Fehler nennen wir *Speicherabbild-Fehler*. In Abbildung 2 führt das dazu, dass der Zeiger *a* weiterhin auf die Adresse 0xBADCOFFA zeigt. Nach dem Neustart wird allerdings dann in Zeile 4 ein Fehler für den Versuch unallokierten Speicher zu lesen geworfen. In diesem Fall würde das Programm dann mit Fehlern enden.

Der letzte Fehler betrifft das Rücksprungregister im persistenten Speicher. Er wird *Aktivierungsaufzeichnungs-Fehler* genannt. Wie in Abbildung 3 zu sehen existieren in einer Methode zwei Funktionen. Diese Funktionen werden nacheinander ausgeführt. Damit der Fehler auftritt benötigt die Funktion F_1 einen Kontrollpunkt, zwischen dem Aufruf von F_1 und F_2, sowie in der zweiten Funktion existieren keine weiteren Kontrollpunkte. Nun wird die Ausführung unterbrochen, während F_2 bearbeitet wird. Wenn das Gerät nun neugestartet wird, so ist der letzte durchlaufene Kontrollpunkt in der Funktion F_1. Dadurch aber, dass die Rücksprungadresse im persistenten Speicher gespeichert wurde, springt das Programm am Ende von F_1 an die Adresse für die Funktion F_2 zurück. Da F_2 nun nicht ausgeführt wurde, ist der aktuelle Zustand nicht gültig.

2 VERHINDERN VON FEHLERN

Wie bereits in Abschnitt 1.2 erwähnt kann man Fehler, die durch zufällige Unterbrechungen auftreten, mit Hilfe von Kontrollpunkten [4, 6, 12] verhindern, es gibt jedoch noch eine breite Auswahl an Kontrollpunkt freien Ansätzen [2, 3, 8, 10].

2.1 Kontrollpunkt gestützte Ausführung

Ein Kontrollpunkt basierter Ansatz wurde von Lucia und Ransford [6] vorgestellt. Dabei handelt es sich um ein Programmier- und Ausführungsmodell welches sie **DINO** (**D**eath **I**s **N**ot an **O**ption) genannt haben. Ein wichtige Eigenschaft dieses Ansatzes ist, dass er keine Hardwareunterstützung benötigt. Des weiteren ist bei DINO jede Instruktion eine Transaktion. Das bedeutet jede Instruktion folgt den ACID Kriterien (aus dem Englischen für: atomar, konsistent, isoliert, dauerhaft), daraus folgt, dass diese entweder als ganzes erfolgreich durchlaufen werden oder scheitern, aber keine Seiteneffekte existieren.

2.1.1 Programmiermodell. DINO erweitert das bekannte Programmiermodell von C um eine Semantik, die atomare Aufgaben ermöglicht. Diese erfüllen wichtige Anforderungen, so werden Änderungen erst dann sichtbar, wenn die Aufgabe erfüllt ist. Außerdem wird nach einer Unterbrechung immer von der Aufgabengrenze neugestartet, welche zuletzt passiert worden ist, dabei wird zugesichert, dass sowohl volatile, als auch nicht-volatile Daten konsistent sind. Dabei kann man sich die Verwendung dieser Semantik ähnlich der Synchronisation von Daten in einem mehrfädigen Programm vorstellen. Es bleibt dem Programmierer überlassen zu entscheiden, wo im Programm kritische Stellen sind, welche Aufgabengrenzen benötigen. Zwar sind diese statisch deklariert, die Aufgabe, also der Weg von einer Aufgabengrenze zur nächsten, wird jedoch dynamisch erzeugt. Daraus folgt, dass es in der Verantwortung des Programmierers liegt die Aufgabengrenzen so zu setzen, dass auch alle Programmpfade abgedeckt sind.

2.1.2 Ausführungsmodell. Um eine unterbrochene, aber korrekte Ausführung des Programms zu garantieren nutzt DINO zwei Mechanismen. Zum einen wird ein *Datenversionsverlauf* für potentiell inkonsistente Daten im NVS angelegt, indem die Variablen in den Stapel, welcher im volatilen Speicher liegt, kopiert werden. Dieser ermöglicht zusammen mit dem zweiten Mechanismus die Garantie, dass Daten des persistenten Speichers konsistent bleiben. Der zweite Mechanismus ist das *Nutzen von Kontrollpunkten*. Entscheidend hierbei ist, dass *jede Aufgabengrenze ein Kontrollpunkt ist*. An diesem werden die Register und der Stapel in einen Bereich des persistenten Speicher, welcher für den Binder reserviert ist, kopiert. Um sicherzustellen, dass die Daten im Kontrollpunkt korrekt sind und nicht beim Speichern korrumpiert worden sind, wird als letzte Operation ein festgelegter Wert geschrieben. Sollte beim Neustart, durch Überprüfen des Wertes, festgestellt werden, dass der Kontrollpunkt nicht vollständig geschrieben worden ist, wird der nächste, ältere, korrekte Kontrollpunkt verwendet. Das ist möglich, da alle Kontrollpunkte doppelt gepuffert sind und sie erst ersetzt werden, wenn der nächste Kontrollpunkt vollständig geschrieben wurde.

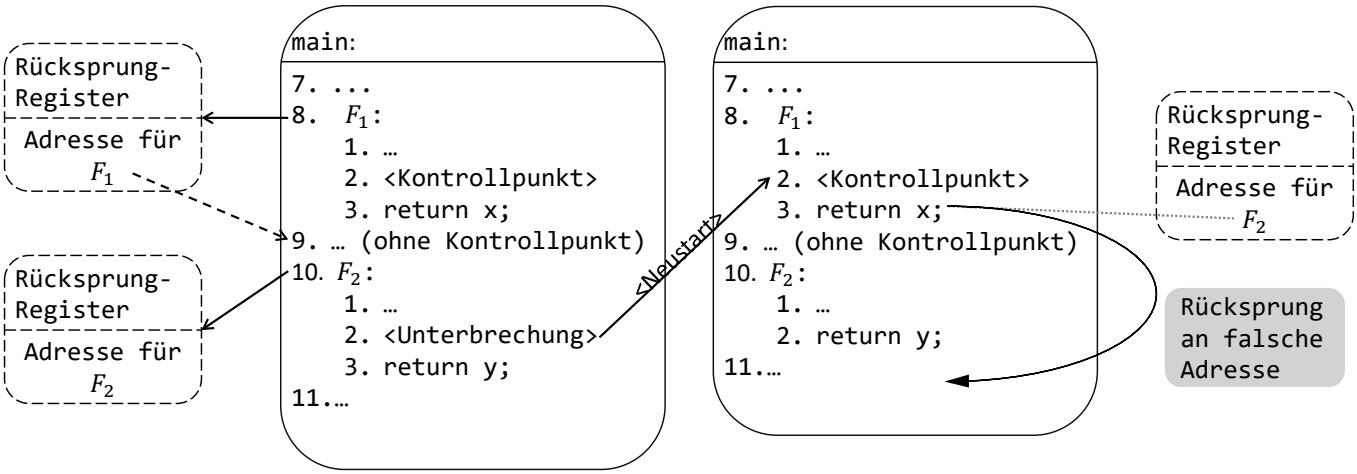


Abbildung 3: Aktivierungsaufzeichnungs-Fehler: Unterbrechung erzeugt ungewollte Sprünge im Programm

2.2 Hardwaregesstützete Kontrollpunkte

Bei DINO ergibt sich jedoch das Problem, dass der Programmierer entscheiden muss, wo ein Kontrollpunkt nötig ist. Das führt unweigerlich zu einer nicht optimalen Lösung. Um dieses Problem zu lösen, existiert der Ansatz von Singla et al.. Mit FlexiCheck [12] stellen sie einen Hardware- / Software- Co-Ansatz vor, der optimieren soll, wann ein Kontrollpunkt gesetzt werden soll. Um möglichst optimal Kontrollpunkte zu setzten, benötigt FlexiCheck jedoch extra Hardware. Diese soll der restlichen Hardware mitteilen, wann ein Kontrollpunkt passiert werden soll.
Um herauszufinden, wann es am besten ist einen Kontrollpunkt zu setzen, wird eine von Singla et al. nicht näher erklärte Hardware trainiert. Als Eingangsinformationen benötigt sie nur die maximal mögliche Umgebungsenergie, die aktuelle Umgebungsenergie, den aktuellen Energievorrat im Gerät und die maximale Kapazität des Speichers für die Energie. Aus diesen Daten wird am Anfang die Hardware darauf trainiert zu bestimmen wann ein Kontrollpunkt gesetzt werden soll, ob die Bearbeitung weitergeht oder gewartet wird. So wird eine Tabelle erstellt, in der die Aktionen passend zum Energieeingang eingetragen werden. Um zu erkennen, wie gut das Modell ist, wird mitgezählt wie häufig das System neustarten muss, sollte diese Anzahl über einem vordefinierten Wert liegen, so wird erneut trainiert. Sobald die Hardware die Trainingsphase verlassen hat, kommt sie in die Prognosephase. In dieser wird das Energieprofil erkannt und aus der vorher angefertigten Tabelle die entsprechende Aktion gewählt.

2.3 Aufgaben gestützte Ausführung

Da Kontrollpunkte eine Betriebslast mit sich bringen, existiert auch die Möglichkeit diese zu umgehen. Einen Ansatz stellt dabei Alpaca [8] von Maeng et al. dar. Ähnlich wie DINO benötigt Alpaca keine Hardwareunterstützung und verwendet *Aufgaben*, auch sie können als atomar betrachtet werden, entscheidend ist allerdings der Unterschied wie das erreicht wird. Durch *Privatisierung* kann garantiert werden, dass sowohl persistenter, sowie volatiler Speicher unabhängig davon, ob Strom vorhanden ist, konsistent bleibt und Aufgaben atomar bearbeitet werden können. Aufgaben werden vom Entwickler definiert, dabei muss er beachten, dass eine Aufgabe nie mehr Energie benötigen darf, als das Gerät, auf dem das Programm ausgeführt wird, auf einmal zur Verfügung stellen kann. Unsichtbar für den Programmierer werden, wie in Abschnitt 2.3.3 erklärt, nach der Ausführung die Änderungen nach außen sichtbar. Dann endet eine Aufgabe immer damit, dass, wenn das Programm nicht terminiert, an die nächste Aufgabe die Kontrolle übergeben wird. Die nächste Aufgabe kann dabei eine andere oder sie selbst sein. Es kann jedoch nie mehr als eine sein, da Alpaca keine parallele Ausführung von Aufgaben zulässt.

2.3.1 Speichermodell. Alpaca unterteilt Daten in zwei Typen, *Aufgabengeteilte* und *Aufgabenbezogene* Daten. Aufgabenbezogene Daten werden nur in einer Aufgabe deklariert und initialsiert. Sie befinden sich im volatilen Speicher. Bei einem Stromausfall gehen sie verloren, das ist aber gewünscht, da nach einem Neustart eine Aufgabe immer neu bearbeitet wird. Wenn mehrere Aufgaben auf die gleichen Daten zugreifen, so handelt es sich um Aufgabengeteilte Daten, sie werden im Kode global deklariert und bekommen Adressen im persistenten Speicher zugeteilt.
Die Privatisierung passiert für den Programmierer unsichtbar, es werden aber für die Aufgaben ein Privatisierungspuffer im volatilen Speicher angelegt, in den alle Variablen gespeichert werden, die geteilt sind und von der Aufgabe genutzt werden. Die Vorgehensweise wird in Abschnitt 2.3.2 beschrieben.

2.3.2 Privatisierung. Um die Betriebslast zu minimieren werden nur skalare aufgabengeteilte Daten in den statisch zugewiesenen Privatisierungspuffer der Aufgabe kopiert, welche Lese- und Schreibe-Abhängigkeiten aufweisen. Um diese zu erkennen erstellt Alpaca den Kontrollflussgraphen des Programm und durchläuft ihn rückwärts, sollte dabei in einem Pfad ein Lesen und dann Schreiben stattfinden, so wird die Variable privatisiert. Anschließend wird vom Kompilierer der Aufgabenkörper so umgeschrieben, dass die Änderungen innerhalb der Aufgabe nur einen Effekt auf das privatisierte Datum haben und erst beim nach außen sichtbar machen das Aufgabengeteiltedatum modifiziert wird.
Damit nicht komplette Felder kopiert werden müssen, wird bei Alpaca mit der Granulität von Feldelementen gearbeitet. Diese privatsierten Elemente werden dann initialsiert, wenn das erste Mal auf ein Feldelement zugegriffen wird. Um dies zu erkennen wird zur

Laufzeit eine Liste, implementiert als Versionierungsmechanismus gestützte Bitmaske, geführt, die alle Feldelemente enthält auf die bereits zugegriffen worden ist. Wenn ein Element nicht in dieser Liste enthalten ist, so ist es der erste Zugriff. Abhängig davon ob der erste Zugriff lesend oder schreibend erfolgt, unterscheidet sie die Vorgehensweise. Ist der erste Zugriff lesend, so wird eine privatisierte Kopie des Elements initialsiert. Handelt es sich um einen schreibenden Zugriff, dann wird das privatisierte Element mit dem zu schreibenden Wert initialisiert. In beiden Fällen wird danach das Feldelement der Liste hinzugefügt.

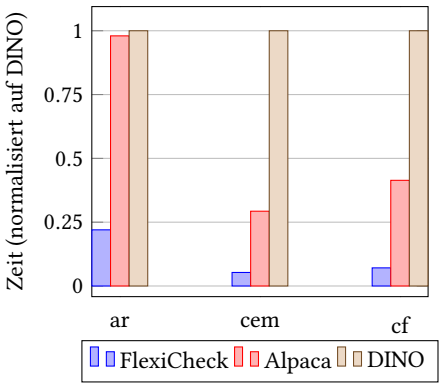
2.3.3 Nach außen sichtbar machen. Da eine Aufgabe atomar ist, müssen die Änderungen am Ende einer Aufgabe sichtbar gemacht werden. Bei diesem Ansatz wird dieser Vorgang in zwei Schritte unterteilt. Das *Vorübergeben* und *Übergeben*. Während der Ausführung der Aufgabe wird im volatilen Speicher durch das Vorübergeben eine Liste mit Änderungen der geteilten Daten erstellt. Dabei werden alle skalaren Daten am Ende einer Aufgabe Vorübergeben, bei Feldelementen wird direkt nach dem ersten Schreiben innerhalb dieser Aufgabe Vorübergeben. Sollte auf das Feldelement erneut geschrieben werden, sind diese Änderungen jedoch sichtbar, da nur Zeiger auf die privatisierten Daten vorübergeben werden. Erreicht nun eine Aufgabe ihr Ende, so wird vor Übergabe der Kontrolle die Liste der Änderungen nauch außen sichtbar gemacht. Dies geschieht beim Übergeben. Dabei wird die vom Vorübergeben erstellte Liste in den persistenten Speicher kopiert und am Ende ein Markierungsbit gesetzt. Sollte dieses Markierungsbit nicht gesetzt sein, impliziert dies, dass die Übergabe nicht vollständig war und es eine Unterbrechung während des Übergebens gab. In dem Fall wird die Aufgabe beim Neustart erneut durchlaufen.

3 VERGLEICH

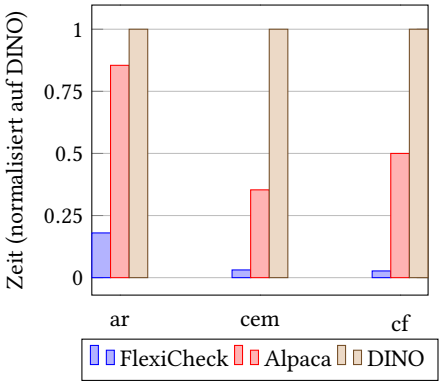
Die vorgestellen Ansätze kann man in vielen unterschiedlichen Aspekten vergleichen. Hier beschränken wir uns zunächst auf die Messungen der Laufzeit, welche bereits durchgeführt worden sind. Eine ausführliche Analyse der Laufzeit, Speichernutzung, sowie zusätzlichen Kode Komplexität der einzelnen Ansätzen überlassen wir zukünftigen Arbeiten. Weitere Aspekte, die zu beachten sind um eine vernünftige Entscheidung zu treffen, welchen Ansatz der Programmierer wählen sollte, sind Hardwarekosten und ob es überhaupt möglich ist solche Hardware hinzuzufügen. Weiterhin sollte betrachtet werden wie viel Komfort der jeweilige Ansatz mit sich bringt und damit einhergehend wie erfahren ein Programmierer sein muss um gute Resultate zu erzielen.

3.1 Geschwindigkeit

Die Laufzeiten werden zwei mal gemessen. Einmal mit einer konstanten Stromversorgung und dann nochmal mit der Energie, die vom Gerät aus der Umgebung gewonnen wird. Die dabei verwendeten Anwendungen unterscheiden sich zwischen FlexiCheck und Alpaca, bzw. DINO. Es gibt jedoch drei Anwendungen, die bei alle Ansätzen getestet wurden, die sind **Activity Recognition (ar)**, **Cuckoo Filtering (cf)**, sowie **Cold-Chain Equipment Monitoring (cem)**. Die Laufzeiten zu DINO und Alpaca stammen dabei von Maeng et al. [8] und die zu FlexiCheck von Singla et al. [12].



(a) bei kontinuierlicher Energie



(b) bei variabler Energie

Abbildung 4: Laufzeiten der Anwendungen

Bei **ar** handel es sich um eine Anwendung, welche Daten eines drei-Achsen-Beschleunigungsmesser nutzt. Diese Daten verwendet es dann um ein Modell zu trainieren, dass dann entscheidet ob das Gerät stillsteht oder geschüttelt wird.
cf speichert eine Reihe an Pseudo-zufälligen Zahlen und nutzt dann den Kuckusfilter um die Reihe wiederherzustellen.
Die Anwendung **cem** protokolliert und LZW-komprimiert Daten, die sie zuvor von einem Temperatursensor gelesen hat. Die Laufzeiten sind in Relation zu DINO angegeben. Abbildung 4a zeigt somit, dass bei einer kontinierlichen Stromquelle, Alpaca bis zu 10mal schneller ist als DINO, Alpaca kann sogar bis zu 15mal schneller Anwendungen bearbeiten [8]. Weiterhin zeigt die Abbildung, dass FlexiCheck bis zu 20mal schneller sein kann [12]. Daraus ergibt sich, dass FlexiCheck zwischen 4,5mal bis 5,8mal so schnell ist, wie Alpaca. Betrachtet man Abbildung 4b, also die Laufzeit, wenn die Energie aus der Umgebung bezogen wird, so ist FlexiCheck zwischen 6 bis 36mal schneller als DINO [12]. Alpaca hingegen ist nur zwischen 1,2 bis 16,6mal schneller [8], somit ist also FlexiCheck potentiell zwischen 4,7 bis 18,5mal schneller als Alpaca.

3.2 Entwickleraufwand

Wie bereits bei der Vorstellung von DINO beschrieben, ist es dem Entwickler überlassen an der richtigen Stelle im Kode einen Kontrollpunkt zu setzten. Diese Methodik ist statisch und erfordert

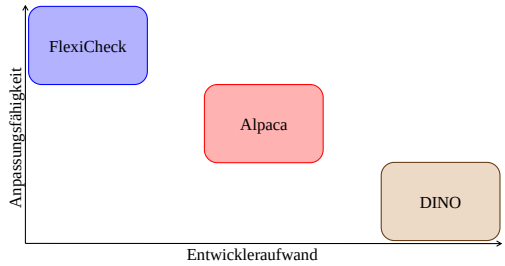


Abbildung 5: Einschätzung des Entwickleraufwand

eine genaue Analyse des Kontrollflusses durch den Entwickler. Dies erfordert nicht nur ein hohes Maß an Erfahrung, sondern auch viel Zeit, da Kontrollflussgraphen schnell sehr groß werden.
Etwas komfortabler wird es für den Entwickler mit Alpaca. Die Privatisierung passiert für den Programmierer unsichtbar, somit bleibt als einziger Faktor, dass eine Aufgabe nie mehr Energie verbrauchen darf, als die Kapazität des internen Energiespeichers des Geräts beträgt [8]. Daraus folgt, dass jedes Programm maßgeschneidert für die Hardware ist. Dabei sollte jedoch bedacht werden, dass sich zwar der Energiebedarf auf anderer Hardware unterscheidet, der interne Energiespeicher allerdings auch. Es ist also denkbar den internen Energiespeicher so zu modifizieren, dass ausreichend Energie zur Verfügung steht, ohne dass der Kode angepasst werden muss. Wenn man die durchschnittliche Länge des Kodes und die Anzahl der neuen Schlüsselwörter von Alpaca und Dino vergleicht, so stellt man fest, dass Alpaca längeren Kode und mehr spezifische Schlüsselwörter benötigt[8]. Man kann also erahnen, dass Alpaca Kode komplexer zu programmieren ist. Wenn der Entwickler das erste Mal mit Alpaca und DINO arbeitet, ist also zu vermuten, dass er die Erweiterung von DINO schneller erlernt hat.
Beim Ansatz von FlexiCheck ist es möglich, dass der Entwickler gar keinen Unterschied zwischen Kode für unterbrochene oder normaler, kontinuierlicher Ausführung erkennen kann. Singla et al. [12] stellen sich dabei eine Modifikation an der existierenden Hardware vor, sodass Kontrollpunkte nur auf Hardwareebene existieren. In diesem Szenario entsteht dem Programmierer kein Mehraufwand. Somit kann man den Programmierkomfort wie in Abbildung 5 grob skizzieren.

3.3 Hardwareanforderungen

Bei großen Sensornetzwerken mit eventuell hunderten von einzelnen Geräten, stellt Hardware einen signifikanten Kostenpunkt dar. Daher sei wohl überlegt ob die Mehrkosten pro Gerät gerechtfertigt sind.

Bei DINO und Alpaca werden keine weiteren Ansprüche an die Hardware gestellt, nur persistenter Speicher muss vorhanden sein. Diese Anforderung ist jedoch im Bereich der batterielosen Geräte einfach zu erfüllen und unterscheidet sich auch nicht weiter von FlexiCheck. Darüber hinaus werden von dem hardwaregestützten Ansatz noch weitere Anforderungen an die Hardware gestellt. Zum einen muss die gesonderte FlexiCheck-Hardware zum Erkennen und Vorraussagen des Energievorrats vorhanden sein, desweiteren soll ein Kontrollpunkt von der Hardware gesetzt werden. Wenn Hardware hinzugefügt wird, erhöht das unweigerlich das Volumen des Geräts. Außerdem muss ein Mikrokontroller geschaffen werden, der die Signale der Prognosehardware akzeptiert und entsprechend verarbeiten kann. Dies erfordert nicht zwingend die Entwicklung und Verifizierung eines komplett neuen Mikrokontrollers, jedoch mindestens die Weiterentwicklung eines bestehenden Mikrokontrollers. Sollte neue Hardware nicht optimal eingesetzt werden, kann es sogar passieren, dass mehr Energie benötigt wird und somit sich die Laufzeit bis zur Beendigung eines Programms erhöht [5]. Des weiteren erfordert neue, spezialisierte Hardware häufig Änderungen an der ISA, wodurch neue Programmierbarrieren erschaffen werden und das Design eines Kompilierers erschwert wird [6].
Durch diese Hardwareanforderung wird es unmöglich FlexiCheck bei bereits existierenden Plattformen zu verwenden. Alpaca und DINO hingegen haben geringe Hardwareanfoderungen und können auf den meisten bestehenden Plattformen betrieben werden.

4 ERGEBNISSE UND ZUKÜNFTIGE ARBEIT

Betrachtet man nun die einzelen Aspekte stellt man fest, dass FlexiCheck signifikant schneller als beide anderen Ansätze ist. Es ist jedoch auch in seiner ersten Implementierung signifikant teurer. So muss erst ein neuer Mikrokontroller entwickelt werden. Dieser kann zwar wiederverwendet werden, erfordert aber ein hohes Initalbudget. Damit ist dann die Ausführung zwar schneller, worauf die Messdaten aus Abschnitt 3.1 jedoch nicht eingehen ist wie lange die Aufladezeiten sind. Eine schnelle Ausführung bedeutet also nicht zwingend eine schnelle Beendigung der Anwendung. Dafür sollte in zukünftigen Arbeiten betrachtet werden, wie lange unterschiedliche Ansätze benötigen um die für das System benötigte Energie zu sammeln.

Welcher Ansatz nun der Beste ist, kann nicht allgemein festgestellt werden, DINO hat jedoch in allen Aspekten schlecht abgeschnitten. Bei bereits bestehenden Plattformen bleibt somit von den vorgestellten Ansätzen nur Alpaca als schnelle und mit wenig Programmieraufwand verbundene Lösung bestehen.
Soll eine neue Plattform entwickelt werden bieten sich Alpaca, als auch FlexiCheck an. Hier muss entschieden werden, ob eine schnellere Ausführung die wesentlich höheren Entwicklungs- und vor allem die Produktionskosten pro Gerät rechtfertigen. Hierbei wäre es interessant zu prüfen, ob neue Ansätze ohne Hardwareunterstützung vergleichbare Ausführungszeiten wie FlexiCheck bieten können. Insbesondere die Weiterentwicklung zu Alpaca, Coala [9], von Majid et al., die einen dynamischen, Aufgaben gestützten Ansatz wählt, sollte dabei betrachtet werden.

LITERATUR

[1] Wahied G. Ali and Sutrisno W. Ibrahim. 2012. Power Analysis for Piezoelectric Energy Harvester. *Energy and Power Engineering* 04, 06 (2012), 496–505. https://doi.org/10.4236/epe.2012.46063
[2] Wei-Ming Chen, Pi-Cheng Hsiu, and Tei-Wei Kuo. 2019. Enabling Failure-resilient Intermittently-powered Systems Without Runtime Checkpointing. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM. https://doi.org/10.1145/3316781.3317816
[3] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358279
[4] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International*

Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016. ACM Press. https://doi.org/10.1145/2983990.2983995

[5] C. Ferri, A. Viescas, T. Moreshet, and M. Herlihy. 2008. Energyimplications of transactional memory for embedded architectures. In *Workshop on Exploiting Parallelism with Transactional Memory andother Hardware Assisted Methods (EPHAM'08)*. https://pdfs.semanticscholar.org/561e/ad061fc98267e63430907decf50cae162818.pdf

[6] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*. ACM Press. https://doi.org/10.1145/2737924.2737978

[7] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/hpca.2015.7056060

[8] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–30. https://doi.org/10.1145/3133920

[9] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. On intermittence bugs in the battery-less internet of things (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems - LCTES 2019*. ACM Press. https://doi.org/10.1145/3316482.3326346

[10] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Transactions on Sensor Networks* 16, 1 (feb 2020), 1–24. https://doi.org/10.1145/3360285

[11] Shad Roundy. 2005. On the Effectiveness of Vibration-based Energy Harvesting. *Journal of Intelligent Material Systems and Structures* 16, 10 (oct 2005), 809–823. https://doi.org/10.1177/1045389x05054042

[12] Priyanka Singla, Shubhankar Suman Singh, and Smruti R. Sarangi. 2019. FlexiCheck: An Adaptive Checkpointing Architecture for Energy Harvesting Devices. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. https://doi.org/10.23919/date.2019.8715130