FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Philip Taffner

Design und Implementierung einer fehlertoleranten Speicherbereinigung für die KESO-JVM

Diplomarbeit

Universität Erlangen Nürnberg, Informatik 4
Martensstr. 1, 91058 Erlangen
Tel.: 09131-85.27277
Fax: 09131-85.8732
E-Mail: i4@cs.fau.de

www4.cs.fau.de

Design und Implementierung einer fehlertoleranten Speicherbereinigung für die KESO-JVM

Diplomarbeit im Fach Informatik

von

Philip Taffner

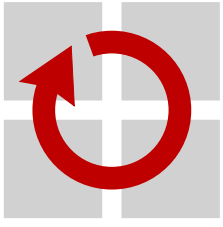
geboren am 02.12.1985 in Ingolstadt

Lehrstuhl für Informatik 4
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreut von:

Isabella Stilkerich
Christoph Erhardt
Martin Hoffmann
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 12. August 2013
Ende der Arbeit: 12. Februar 2014



Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 12. Februar 2014 _____

Zusammenfassung

Um die Leistungsfähigkeit von Mikrochips bei gleichbleibender oder geringerer Baugröße, Leistungsaufnahme und Wärmeabgabe zu steigern, ist es essentiell die Strukturgröße ihrer Fertigungsprozesse weiter zu reduzieren. Unerwünschter Nebeneffekt der Strukturverkleinerung ist die höhere Wahrscheinlichkeit des Auftretens von zufälligen Fehlern im Speicher, den sogenannten Bitkippern. Auch umweltbedingte Einflüsse wie Strahlung oder technische Faktoren wie Unterspannung können Auslöser sein und sind damit als Fehlerquellen zu berücksichtigen. Es ist wünschenswert, dass solche unvorhersagbaren Fehler nicht zu fehlerhaftem Verhalten, reduzierter Stabilität oder gar zu komplettem Ausfall eines Systems führen.

Hardware-Speicherschutz ist oftmals zu komplex und teuer für Mikrocontroller kleiner eingebetteter Systeme. Daher spielt software-basierte Fehlererkennung und -korrektur eine wichtige Rolle, um das Fehlen einer MPU auszugleichen oder eine vorhandene um Funktionen zu ergänzen. Sie kann dabei eine höhere Fehlererkennungsrate bieten, da auch ungültige Zugriffe erkannt werden können, die im eigenen Speicherschutzraum stattfinden.

Im Rahmen dieser Arbeit wurde das Design und die Implementierung einer fehlertoleranten Speicherbereinigung vorgestellt, evaluiert und diskutiert. Dazu wurden die Speicherverwaltungs-Varianten der KESO Multi-JVM, einer virtuellen Maschine für Java, die für statisch konfigurierte eingebettete Systeme konzipiert ist, analysiert und erweitert. KESO besitzt mit JINO einen Ahead-of-Time-Übersetzer, der Java-Anwendungen unter Beibehaltung der Typsicherheit der Sprache nach ANSI-C übersetzen kann. Die Fehlertoleranz der Laufzeitumgebung von KESO, welche zur Integritätsprüfung der Typinformationen verwendet wird, wurde durch diese Arbeit weiter vervollständigt.

Die verschiedenen Datenstrukturen der Speicherverwaltungen und automatischen Speicherbereinigungen wurden auf ihre Anfälligkeit für transiente Fehler hin untersucht und effektive und ressourcenschonende Mechanismen zur Absicherung auf Basis von Paritäten entworfen und vorgestellt.

In einer abschließenden Evaluation konnte durch Laufzeitmessungen und Fehlerinjektions-Experimente gezeigt werden, dass die entworfenen und implementierten Mechanismen ressourcenschonend, effizient und effektiv sind.

Abstract

To increase the performance of microchips, it is essential to reduce the structure width of their manufacturing process. An unwanted side effect is the increased probability of bit flips. Also environmental conditions like radiation and technical aspects like low supply voltages have to be considered as sources of transient errors. It is desirable to avoid faults, decreased stability or a complete system failure despite those unpredictable errors.

A hardware memory protection unit is often too complex and expensive for small embedded systems. Because of this, software based fault detection and correction is important to compensate the absence of a MPU. It is even possible to increase the error detection rate by recognition of invalid addresses inside of the same protected address space.

In this thesis, the design and implementation of a fault tolerant garabage collector was presented, evaluated and discussed. The different heap implementations of the KESO Multi-JVM, a Java Virtual Machine implementation for statically configured embedded systems, were analyzed and extended. KESO comes with JINO, an ahead-of-time compiler that translates Java applications to ANSI-C while maintaining the type safety of the language. The fault tolerance of the KESO runtime environment, which is used to ensure the integrity of the type system, was increased by this thesis.

The various data structures used by the heap implementations and automatic garbage collectors were examined for their susceptibility to transient errors. Effective and resource-efficient mechanisms based on parity checks were designed and presented.

Runtime measurements and fault-injection experiments showed, that the designed and implemented mechanisms are resource- and runtime-efficient and effective against transient errors.

Inhaltsverzeichnis

1 | Einführung

Heutzutage stößt man überall im Alltag auf Computer. Zuerst denkt man üblicherweise an PCs, Notebooks oder Smartphones. Doch schon Geräte wie die Kaffeemaschine, Mikrowelle oder die Waschmaschine enthalten kleine Computer. Auch größere, komplexere Systeme wie Autos, Flugzeuge oder medizinische Geräte wären ohne Computer nicht mehr denkbar. Bei den meisten dieser Systeme handelt es sich um sogenannte eingebettete Systeme. Diese werden von einem oder mehreren programmierbaren Mikrocontrollern betrieben, die für ihren Einsatzzweck maßgeschneidert sind oder zumindest gerade so viel Funktionalität bieten, damit die gewünschten Aufgaben zuverlässig ausgeführt werden können.

Waren Mikrocontroller früher dafür ausgelegt nur eine einzige Aufgabe zu erledigen, geht der Trend dank Fortschritte in der Technologie dahin, dass komplexere und leistungsfähigere Mikrochips entworfen werden, die mehrere Funktionalitäten bieten können. Dabei werden neue Anforderungen an Hardware und Betriebssoftware gestellt.

1.1 | Motivation

Um die Leistungsfähigkeit von Mikrochips bei gleichbleibender oder geringerer Baugröße, Leistungsaufnahme und Wärmeabgabe zu steigern, ist es essentiell die Strukturgröße ihrer Fertigungsprozesse weiter zu reduzieren. Unerwünschter Nebeneffekt der Strukturverkleinerung ist die höhere Wahrscheinlichkeit des Auftretens von zufälligen Fehlern im Speicher, den sogenannten Bitkippern [Bor05]. Auch umweltbedingte Einflüsse wie Strahlung [TN93] oder technische Faktoren wie Unterspannung können Auslöser sein und sind damit als Fehlerquellen zu berücksichtigen. Es ist wünschenswert, dass solche unvorhersagbaren Fehler nicht zu fehlerhaftem Verhalten, reduzierter Stabilität oder gar zu komplettem Ausfall eines Systems führen. Die Steuerung des Airbags im Auto sollte beispielsweise zuverlässig funktionieren und nur bei einem Auffahrunfall auslösen, nicht dagegen während der Fahrt. Um eine Fehlertoleranz zu gewährleisten gibt es verschiedene Ansätze.

Zum einen existiert hardware-basierte Fehlertoleranz, die beispielsweise durch redundante Bauteile und spezielle fehlerkorrigierende Speicherbausteine (ECC) erreicht wird. Da hierfür zusätzliche Hardware verbaut werden muss, steigen Material- und Produktionskosten, was für Massenprodukte meist wirtschaftlich nicht rentabel ist. Des Weiteren benötigen weitere elektronische Bauteile mehr Platz, erhöhen das Gewicht und verbrauchen mehr Strom.

Ein anderer Ansatz hingegen ist die software-basierte Fehlertoleranz. Übliche Techniken sind Replikation, d.h. die redundante Speicherung von Werten und die parallele oder zeitversetzte Ausführung von Programmteilen, und das Anlegen von Prüfsummen. Sie benötigen keine zusätzliche Hardware, können aber statt oder ergänzend zu hardware-basiertem Schutz eingesetzt werden. Typsichere Programmiersprachen können dazu beitragen Teile einer Anwendung von einander zu isolieren, da zum einen bereits bei der Kompilierung (statische Typprüfung) und zum anderen zur Laufzeit (dynamische Typprüfung) sichergestellt wird, dass nicht über Zeiger oder durch Überschreiten der Grenzen von Arrays oder Objekten auf ungültige Speicherbereiche zugegriffen wird, Rücksprungadressen von Funktionen manipuliert werden oder Operatoren mit nicht zueinander passenden Operanden ausgeführt werden. Transiente Fehler können allerdings dazu führen, dass das System, das in der Laufzeitumgebung zur Typprüfung eingesetzt wird, fehlerhaft arbeitet. Eine Speicherschutzeinheit (memory protection unit), ein hardware-basierter Speicherschutz, kann eingesetzt werden, um den Zugriff auf ungültige Speicherbereiche zu verhindern. Diese Einheiten sind komplex und auf vielen Mikrocontrollern, auf denen eingebettete Systeme betrieben werden sollen, nicht verfügbar. Daher ist es wünschenswert, eine Fehlererkennung bzw. -toleranz gegenüber transienten Fehlern für die Laufzeitumgebung, und damit auch die Typprüfung bereitzustellen, die auch ohne Unterstützung durch Hardware arbeitet.

Im folgenden Abschnitt wird die Laufzeitumgebung KESO für die typsichere Sprache Java vorgestellt, für welche im Rahmen dieser Arbeit ein softwarebasierter Schutz gegen transiente Fehler für drei verschiedene Speicherverwaltungen und GCs entworfen und implementiert wurde.

1.2 | Die KESO Multi-JVM

KESO [SSWSP12] ist eine virtuelle Maschine für Java (JVM), die für statisch konfigurierte eingebettete Systeme konzipiert ist. Statisch konfiguriert bedeutet, dass vorab unter anderem sämtliche Tasks, Ressourcen, Alarme und Interrupts festgelegt werden. Das System kann beispielsweise zur Laufzeit keine neuen Programmfäden (Threads) erzeugen, die nicht vorher konfiguriert wurden. Bei KESO handelt es sich zum einen um einen Compiler (JINO), der Java-Byte-Code nach ANSI-C übersetzt und zum anderen um eine zugehörige Laufzeitumgebung, die auf einem Echtzeit-Betriebssystem nach OSEK/VDX [OSE05]- oder AUTOSAR

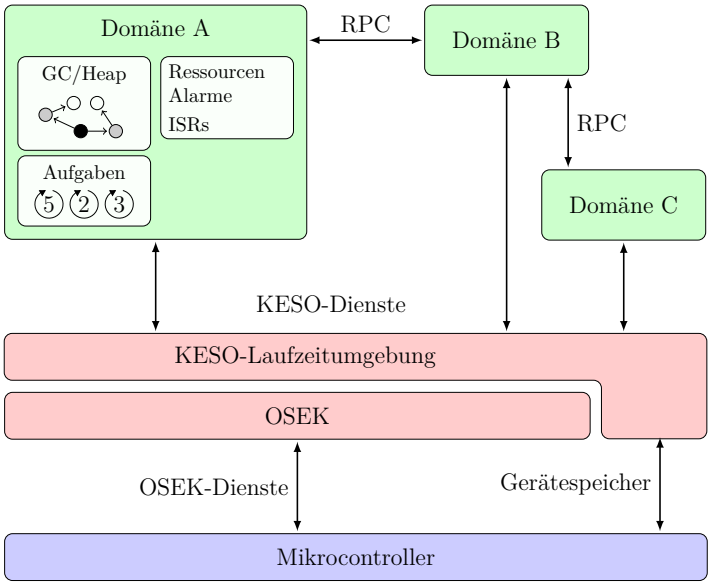


Abbildung 1.1: Schematische Ansicht des KESO-Systems zur Laufzeit

OS [AUT11]-Standard betrieben wird. Alle Anwendungen und Treiber können in Java programmiert werden. JINO übersetzt den Großteil des Quelltexts nach ANSI-C, wobei die Java-Thread-API eine Ausnahme darstellt und auf die Abstraktionsschicht des verwendeten Betriebssystems abgebildet wird (z.B. AUTOSAR OS). Dabei, und auch bei Generierung der zur spezifischen Anwendung optimierten KESO-Laufzeitumgebung, wird die zuvor angelegte Konfiguration berücksichtigt, die genau auf das Zielsystem abgestimmt werden kann. Das Konzept erlaubt außerdem kein dynamisches Nachladen von Programmteilen zur Laufzeit. Dadurch kann der KESO-Compiler mithilfe statischer Code-Analyse sehr effektive Optimierungen durchführen, da die gesamte Code-Basis bereits bei der Kompilierung bekannt ist. Alle Optimierungen werden vollständig bei der Übersetzung und somit vor der eigentlichen Ausführung (ahead-of-time) und nicht wie beispielsweise in der Oracle JVM zur Laufzeit (just-in-time) durchgeführt. Somit kann die Ausführungszeit der ursprünglichen Java-Anwendung auf ein mit einer C-Anwendung vergleichbares Niveau reduziert werden.

Eine schematische Ansicht des KESO-Laufzeitsystems ist in Abbildung 1.1 dargestellt. KESO bildet hierbei eine Abstraktionsschicht zum eigentlichen OSEK-Betriebssystem. Zusätzlich ermöglichen spezielle Schnittstellen die Ausführung von nativem Programmcode und den direkten Zugriff auf den Speicher des Geräts durch Java-Objekte (Memory-mapped I/O).

Ein wichtiger Bestandteil ist das Domänenkonzept. Für jede Applikation können beliebig viele Domänen konfiguriert werden. Durch die in KESO implementierten Sicherheitsmechanismen sind diese Domänen komplett voneinander iso-

liert und es wird verhindert, dass auf fremden Speicher anderer Domänen zugegriffen wird. Dieses System erlaubt eine sichere Koexistenz mehrerer Tasks auf demselben Mikrocontroller, vergleichbar mit dem Prozess-Modell moderner Betriebssysteme. Jede Domäne verwaltet ihre eigenen Ressourcen, Event-Handler, Alarme, Tasks, einen eigenen Heap für dynamisch allokierten Speicher, sowie auf Wunsch eine eigene automatische Speicherbereinigung (engl. garbage collector, kurz: GC), welche individuell pro Domäne feingranular justiert werden kann. Sowohl die Größe des Heaps als auch die GC-Variante kann speziell für die Anforderungen der Domänen-Tasks konfiguriert werden. Jeder Domäne wird ausschließlich Zugriff auf ihre eigenen Objekte und deren Methoden gewährt. Damit dennoch Kommunikation zwischen den Domänen möglich ist, existiert ein RPC-artiger Portal-Mechanismus, der einen gesicherten Informationsaustausch gewährleistet. Aus Sicht der Applikation verhält sich jede der Domänen wie eine eigene JVM, weshalb KESO als Multi-JVM bezeichnet wird.

Im letzten Abschnitt 1.1 wurde dargelegt, warum eine software-basierte Fehlertoleranz für eingebettete Systeme wünschenswert sein kann. Die KESO-Laufzeitumgebung bietet neben der Unterstützung einer typsicheren Ausgangssprache (Java) der Anwendung als zusätzliches Feature bereits einige Mechanismen der Fehlererkennung.

Die Laufzeitumgebung selbst muss abgesichert werden, um die Typsicherheit auch bei transienten Fehlern (Bitkippern) gewährleisten zu können. Hierfür muss sichergestellt werden, dass ungewollte Veränderungen der Objekt- und Feld-Referenzen und der darin enthaltenen Typinformationen erkannt werden. Bei Objekten handelt es sich bei der Typinformation um eine Klassen-ID, welche als Index beim Nachschlagen in Klassen- und Methodentabellen verwendet wird. Felder enthalten zusätzlich eine Information über ihre Größe. Treten Bitkipper bei der in der Referenz enthaltenen Adresse oder den Typinformationen auf, kann dies zu unerlaubten Zugriffen auf fremde Speicherbereiche, beispielsweise die einer anderen Domäne führen. Um diese Fehler zu erkennen, kann KESO daher so konfiguriert werden, dass vor jedem Zugriff auf eine Referenz (dereference check, DRC) oder direkt nach jedem Laden einer Referenz aus dem Speicher (load reference check, LRC) in ein Register eine Überprüfung ihrer Integrität durchgeführt wird. Diese wird beispielsweise anhand einer Prüfsumme (Parität), die in der Referenz selbst abgelegt ist, kontrolliert.

Um zusätzlich auch für die Anwendung Erkennung und Toleranz transienter Fehler zu ermöglichen, bietet KESO die Replikation von kritischen Programmteilen an [TSK+11]. Statische Felder beispielsweise, die normalerweise in einem gemeinsamen globalen Adressraum abgelegt sind, werden dabei in jede Domäne repliziert übernommen. Der Entwickler kann vorab in der Konfiguration festlegen, wieviele Bitkipper erkannt und toleriert werden sollen. JINO erzeugt anhand dieser Information automatisch Mechanismen, die auf Basis eines Mehrheits-Algorithmus fehlerhafte Replikate erkennen und korrigieren können. Die KESO-Laufzeitumgebung kümmert sich automatisch um die Instantiierung aller Repli-

kate beim Start und deren Überprüfung durch Vergleich durch Mehrheitsentscheid.

Wie bereits in diesem Abschnitt erwähnt, bietet KESO die Möglichkeit pro Domäne auf Wunsch einen GC zu verwenden. Der Entwickler kann zwischen zwei Varianten (siehe Abschnitt 2.2) wählen, die je nach Applikation und Domänen-Tasks unterschiedlich geeignet sind. Im Rahmen dieser Arbeit wurde untersucht, welche Maßnahmen erforderlich sind, um diesen kritischen Teil der Laufzeitumgebung ebenfalls fehlertolerant zu implementieren um damit die Fehlererkennung des Gesamtsystems KESO weiter zu vervollständigen.

1.3 | Aufbau dieser Arbeit

Im folgenden Kapitel werden die Datenstrukturen der in KESO implementieren Varianten der Speicherverwaltung und deren Möglichkeit zur automatischen Speicherbereinigung detailliert im Hinblick auf deren Anfälligkeit gegenüber transienten Fehlern analysiert und Möglichkeiten der Absicherung diskutiert.

Kapitel 3 stellt verschiedene im Rahmen dieser Arbeit verwendete Techniken der Fehlererkennung und -korrektur vor. Die Ergebnisse verschiedener Messungen zur Effizienz und Effektivität der durchgeführten Erweiterungen werden in Kapitel 4 vorgestellt.

Kapitel 5 zieht ein Fazit und gibt einen kurzen Ausblick auf mögliche zukünftige Arbeiten.

2 | Problemanalyse

KESO ist eine Multi-JVM, die Java-Bytecode nicht wie viele andere JVM-Implementierungen interpretiert oder zur Laufzeit optimiert (just-in-time, JIT). Der Programmcode wird stattdessen vorab (ahead-of-time, AOT) nach ANSI-C und anschließend in nativen Maschinencode übersetzt. Da es sich bei Java um eine typsichere Sprache handelt, ist es naheliegend und wünschenswert dieses Feature auch nach der Übersetzung zu C beizubehalten. Der KESO-Übersetzer *JINO* erzeugt hierfür zusätzlich zur eigentlichen Applikation eine Laufzeitumgebung, die sicherstellt, dass die Typsicherheit weiterhin gewährleistet wird.

Aufbauend auf einer typsicheren Basis kann eine Sprache wie Java Funktionalitäten bieten, die andere schwach typisierte Sprachen wie C nur schwer oder gar nicht realisieren können. Hierzu zählt die im Rahmen dieser Arbeit analysierte und gegen transiente Fehler abgesicherte automatische Speicherbereinigung. Darunter versteht man die automatische Freigabe von nicht mehr benötigtem zuvor reserviertem Speicher durch eine Laufzeitumgebung. Die Zuständigkeit wird vom Programmierer auf dieses System übertragen. Dieser muss sich keine Gedanken mehr darüber machen, wann er Speicher freigeben kann und muss. Das Verfahren bietet einige Vorteile, beispielsweise die Vermeidung von Fehlern wie Speicherlecks oder doppelt freigegebener Speicher, welche sicherheitsrelevante Aspekte darstellen.

Auch für ein in C geschriebenes System ist es möglich einen GC zu implementieren. Dieser müsste allerdings sehr konservativ agieren. Denn aufgrund der fehlenden Typsicherheit bzw. des Fehlens der dadurch zur Verfügung gestellten Meta-Informationen, ist es dem Mechanismus nicht möglich mit absoluter Sicherheit festzustellen, ob es sich bei Daten im Speicher um Referenzen bzw. Zeiger handelt oder um primitive Ganzzahlen. Aus diesem Grund kann der GC unter Umständen diverse Speicherbereiche nicht freigeben, obwohl es sich um Referenzen handeln würde, die nicht mehr von anderen Objekten referenziert sind.

In diesem Abschnitt wird zunächst das dieser Arbeit zugrunde liegende Fehlermodell vorgestellt. Anschließend werden die bereits in KESO implementierten Speicherverwaltungs-Varianten erläutert. Dabei wird deren Struktur und Funktionsweise im Hinblick auf potentiell kritische Bereiche hinsichtlich transienter

Fehler analysiert und Ansätze zur Realisierung einer Absicherung vorgestellt.

2.1 | Schutz gegen transiente Fehler

Fehlererkennung kann auf mehreren Ebenen realisiert werden. Üblicherweise geschieht dies zum einen auf Hardware- und zum anderen auf Software-Ebene. Die elektronischen Schaltungen der Hardware können durch Redundanz abgesichert werden. Das bedeutet, dass eine Schaltung in doppelter oder mehrfacher Ausführung implementiert ist und diese parallel durchlaufen werden. Am Ende entscheidet eine vergleichende Komponente, ob ein Fehler vorliegt.

Die für diese Arbeit relevante Kategorie des Schutzes gegen transiente Fehler ist jene, die durch Software erreicht werden kann und damit unabhängig von der zugrunde liegenden Hardware eingesetzt werden kann. Es wird sowohl die Erkennung als auch die Korrektur von Fehlern während der Allokations- und GC-Phase der KESO-Laufzeitumgebung diskutiert.

2.1.1 | Fehlermodell

Damit ein software-basierter Fehlertoleranz-Mechanismus effektiv reagieren und eingreifen kann, ist zunächst festzusetzen, dass nur Fehler erkannt und berücksichtigt werden können, die an der Programmierschnittstelle des Prozessors erkennbar sind. Diese Fehlerklasse beinhaltet flüchtige Fehler im Arbeitsspeicher, die beispielsweise durch unvorhersagbare Umwelteinflüsse wie Strahlung oder durch fehlerhafte Bauteile oder mangelnde Stromversorgung erzeugte Spannungsschwankungen verursacht werden. Im Rahmen der Arbeit wird von einzelnen Bitkippern ausgegangen, die wahllos im Speicher auftreten können. Unter Bitkipper versteht man die unvorhersagbare und nicht vorgesehene Wertänderung eines Bits von Null zu Eins oder umgekehrt. Es wird ferner davon ausgegangen, dass sich das Text-Segment und die konstanten Teile des Daten-Segments in einem gesonderten, besser geschützten Bereich (ROM) befinden und dadurch ausführbarer Code und konstante Daten als sicher betrachtet werden können.

Man unterscheidet zwischen Fehlererkennung und Fehlerkorrektur. Je nach Anwendung, Zielsystem und Einsatzgebiet kann Fehlererkennung durchaus ausreichend sein. KESO bietet beispielsweise die Möglichkeit der Anwendungsreplikation auf Domänenebene (siehe Abschnitt 1.2). Wird ein Fehler in einem der Replikate erkannt, wird von der Laufzeitumgebung eine Ausnahme (Exception) generiert und das Replikat wird beispielsweise zurückgesetzt und von neuem gestartet. Eine aktive transparente Korrektur des Fehlers, die mit Mehraufwand in Laufzeit und Ressourcenverbrauch einhergeht, ist nicht immer notwendig und kann vermieden werden.

Beim Design und der Implementierung eines Schutzes für die automatische

Speicherbereinigung wurden sowohl Methoden der Fehlererkennung als auch -korrektur erarbeitet und umgesetzt.

2.2 Automatische Speicherbereinigung in der KESO Multi-JVM

Die KESO-Laufzeitumgebung bietet derzeit zwei unterschiedliche Varianten der automatischen Speicherbereinigung an, die in diesem Kapitel vorgestellt werden. In der Konfiguration werden diese als *CoffeeBreak* und *IdleRoundRobin* bezeichnet. Jede dieser Varianten verwendet teilweise andere interne Datenstrukturen. Um die zuverlässige Funktionalität trotz potentiell auftretender Bitkipper zu gewährleisten, ist es essentiell diese Strukturen zu erweitern und Mechanismen zu implementieren, die eine Fehlererkennung und ggf. Korrektur erlauben.

Alle Varianten der Speicherverwaltung verwenden als Basis für den Heap ein im BSS-Segment abgelegtes mit Null initialisiertes Array, dessen Größe in Byte in der Konfigurationsdatei vor dem Übersetzen festgelegt wird. Jede Domäne besitzt dabei einen von den anderen Domänen isolierten Heap, ergo kann jede Domäne einen unterschiedlich großen Heap verwalten.

2.3 Allokator (Restricted-Domain-Scope)

Die einfachste Speicherverwaltung für den Heap, die KESO bietet, ist die sogenannte *Restricted-Domain-Scope*-Variante (RDS). Dabei wird ein interner Zeiger verwaltet, der auf die Adresse des nächsten freien Speicherbereichs zeigt. Hierbei handelt es sich um einen Index des im letzten Abschnitt erwähnten Heap-Arrays. Wird Speicher über die Allokations-Funktion angefordert, wird zunächst geprüft, ob ab dem aktuell gesetzten Index hin zum Ende des Arrays noch genügend Platz verfügbar ist, um die angeforderte Bereichsgröße vergeben zu können. Abbildung 2.1 stellt die Situation bei einer Allokation dar. Zu erkennen sind das Heap-Array, der Heap-Zeiger und die Menge an Speicher, die gerade angefordert wird.

Die RDS-Variante stellt keine automatische Speicherbereinigung bereit. Einmal allokierter Speicher kann nicht wieder zur Verfügung gestellt werden. Das System reagiert daher mit einem Fehler (Error-Exception), wenn der Heap-Zeiger bereits auf das Ende des Heap-Arrays zeigt. Dennoch hat diese Implementierung ihre Daseinsberechtigung, denn gerade in eingebetteten Systemen, bei denen die oftmals kleinen Applikationen nur wenig Speicher benötigen, ist der Verbrauch vorab bekannt und deterministisch bestimmbar oder hinreichend abschätzbar. Gängige

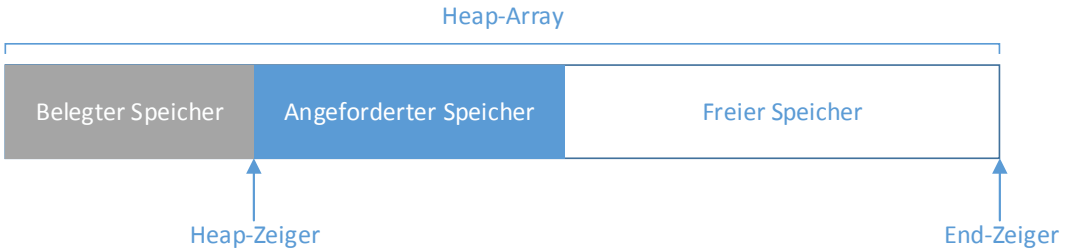


Abbildung 2.1: Schematische Darstellung der Speicherverwaltung des Restricted-Domain-Scope

Praxis beim Entwurf der Anwendungen ist auch, dass diese keine dynamischen Allokationen durchführen, sondern ihren gesamten benötigten Speicher bei ihrer Initialisierung einmalig anfordern. Der Heap kann daher den Anforderungen genügend groß festgelegt werden.

Der Vorteil dieser Variante besteht darin, dass aufwendigere und auch potentiell fehleranfälligere GC-Mechanismen vermieden werden können und sie aufgrund ihrer Einfachheit ein gut vorhersagbares konstantes zeitliches Verhalten aufweist. Nachteil ist, dass die Applikation so konzipiert sein sollte, dass alle Objekte bereits während der Initialisierung der Anwendung allokiert werden, da zu einem späteren Zeitpunkt unter Umständen nicht genügend freier Speicher für dynamische Allokationen zur Verfügung steht.

Kritische Datenstrukturen

Die beiden relevanten Datenstrukturen, die beim RDS Verwendung finden, sind der Heap-Zeiger und ein Zeiger auf die End-Adresse des Heap-Arrays (End-Zeiger). Folglich müssen diese beiden gegen transiente Fehler abgesichert werden. Beim Auftreten eines Bitkippers sind sechs Szenarien denkbar, die zu drei Resultaten bzw. Verhaltensweisen des Systems führen können. Bei der Beschreibung wird von wachsenden Adressen, wie es beispielsweise bei der x86-Architektur der Fall ist, ausgegangen.

1. Zunächst kann sich der Heap-Zeiger durch den Einfluss so verändern, dass er auf eine Adresse außerhalb des noch freien Adressbereichs des Heap-Arrays zeigt. Verweist er auf eine Adresse vor dem Speicherbereich des Arrays, handelt es sich dabei um eine ungültige Speicheradresse. Außerdem könnte auch fremder Speicher überschrieben werden, was zwar nicht unmittelbar in einem Fehler resultiert, jedoch zu anderen Seiteneffekten im weiteren Verlauf der Anwendung führen könnte.

2. Würde der Zeiger auf eine Adresse nach dem Adressbereich des Arrays verweisen, würde die RDS-Implementierung einen Out-Of-Memory-Fehler

(kein freier Speicher übrig) auslösen und das System beenden.

3. Die dritte Möglichkeit besteht darin, dass der End-Zeiger so beeinflusst wird, dass dieser auf eine niedrigere Adresse als der aktuelle Heap-Zeiger verweist. Dies führt abermals zu einem Out-Of-Memory-Fehler.

4. Eine Speicheradresse des End-Zeigers jenseits der Obergrenze des Heap-Arrays kann dagegen bei der aktuellen oder einer zukünftigen Allokation dazu führen, dass fälschlicherweise Speicher vergeben wird, der nicht für den Heap vorgesehen ist. Das wiederum kann sofort nach der Allokation oder im weiteren Verlauf der Applikation zu Fehlern führen.

5+6. Zuletzt besteht auch die Möglichkeit, dass sowohl Heap- als auch End-Zeiger so manipuliert sein könnten, dass sie weiterhin auf eine gültige Adresse innerhalb des Heap-Arrays zeigen. Dadurch wird freier Heap-Speicher verschenkt. Dies muss die Anwendung nicht zwingend negativ beeinflussen. Je nach konfigurierter Größe des Heaps und Position des gekippten Bits ist es durchaus denkbar, dass die Applikation weiterhin korrekt ausgeführt werden kann. Im schlimmsten Fall wird sich KESO zu einem zukünftigen Zeitpunkt mit einem Out-Of-Memory-Fehler beenden.

Maßnahmen zur Absicherung

Um die beiden im letzten Abschnitt beschriebenen Zeiger gegen transiente Fehler abzusichern, können Paritäten oder Replikation verwendet werden. Dadurch können die diskutierten möglichen Fehlverhalten erkannt und auf Wunsch toleriert werden. Die Implementierung beider Mechanismen wird in Kapitel 3 detailliert erklärt.

2.4 | Durchsatzstarke Speicherbereinigung (Coffee-Break)

Im letzten Abschnitt wurde eine Speicherverwaltung ohne automatische Bereinigung beschrieben. Dabei kann bereits vergebener Speicher nicht erneut zur Verfügung gestellt werden, selbst wenn sich darin befindliche Objekte von der Anwendung nicht mehr referenziert, das heißt nicht mehr verwendet werden. Diese Bedingung muss für kleine eingebettete Systeme keine schwerwiegende Einschränkung bedeuten. Allerdings kann es für umfangreichere Anwendungen nützlich sein, dass die Laufzeitumgebung eine automatische Speicherbereinigung besitzt, um die Größe des Heaps gering zu halten. Vorteilhaft wirkt sich diese Situation hinsichtlich der Anwendungs-Replikation auf Domänen-Ebene aus. Kann der Mikrocontroller beispielsweise nur eine geringe Menge an Speicher zur Verfügung

stellen, ermöglicht die reduzierte Heap-Größe der einzelnen Domänen dennoch in gewissem Umfang eine Absicherung durch Replikation, da es aufgrund der geringeren Heap-Größe möglich ist, mehrere Domänen zu konfigurieren und zu betreiben.

Bei der automatischen Speicherbereinigung wird regelmäßig ein spezieller Prozess ausgeführt, der anhand verschiedener Hilfsstrukturen die Bereiche im Heap-Speicher auffindet, welche in vorherigen Allokationen an die Applikation vergeben wurden, inzwischen allerdings nicht mehr referenziert sind.

In diesem Abschnitt wird der sogenannte *Coffee-Break* (CB) vorgestellt. Dabei handelt es sich um einen nicht durch Interrupts unterbrechbaren Prozess. Er ist daher sehr durchsatzstark und kann während eines Laufs unter Umständen viel Speicher auf einmal freigeben. Nachteil ist, dass während des GC-Laufs der Rest der Anwendung blockiert ist. Dies führt zu hohen Blockadezeiten und einer schlechten maximalen Laufzeit (WCET). Festgelegte Zeitschranken zur Ausführung von Programmteilen und damit eine deterministische Laufzeit können nicht eingehalten werden.

Eine weitere, in einigen Programmteilen abweichende Implementierung, die speziell für diese Anforderungen konzipiert wurde, wird später in Abschnitt 2.5 beschrieben.

In den folgenden Abschnitten werden die im Coffee-Break verwendeten Datenstrukturen vorgestellt und die Auswirkung von transienten Fehlern auf diese analysiert, sowie Ansätze zu deren Absicherung vorgestellt.

2.4.1 | Freispeicherliste

Als Basis der Speicherverwaltung dient wie beim RDS ein Array, das im BSS-Segment liegt. Die in diesem Heap-Array verfügbaren Speicherblöcke sind in einer einfach verketteten Liste, der Freispeicherliste, hinterlegt. Jedes Element in dieser Liste besitzt zwei Felder. Zum einen die Größe des Freispeicher-Blocks, auf den das Listenelement zeigt und zum anderen einen Zeiger auf das nächste Element der Liste (Next-Zeiger), also den Index des nächsten freien Speicherblocks im Heap-Array. Der Zeiger auf den Kopf der Liste ist in der Domänen-Struktur gespeichert. Der Next-Zeiger des letzten Elements in der Liste zeigt auf NULL und terminiert diese dadurch. Bei Start der Anwendung besitzt die Liste nur ein einziges Element, dessen Größe der des Heaps entspricht. Durch jede folgende Allokation wird die Größe dieses Elements weiter dekrementiert. Dieser Zustand ändert sich erst, wenn der GC das erste mal ausgeführt wird. Bei der dabei durchgeführten Freigabe von Speicherbereichen des Heap-Arrays wird die Freispeicherliste bei jedem Lauf komplett neu angelegt. Für jeden Freispeicher-Block wird dabei ein neues Element ans Ende der Freispeicherliste angehängt. Das Resultat ist eine verkettete Liste, welche die Lücken im Heap-Array widerspiegelt. Die zur Beschreibung der Listenelemente verwendeten C-Structs werden dabei direkt im

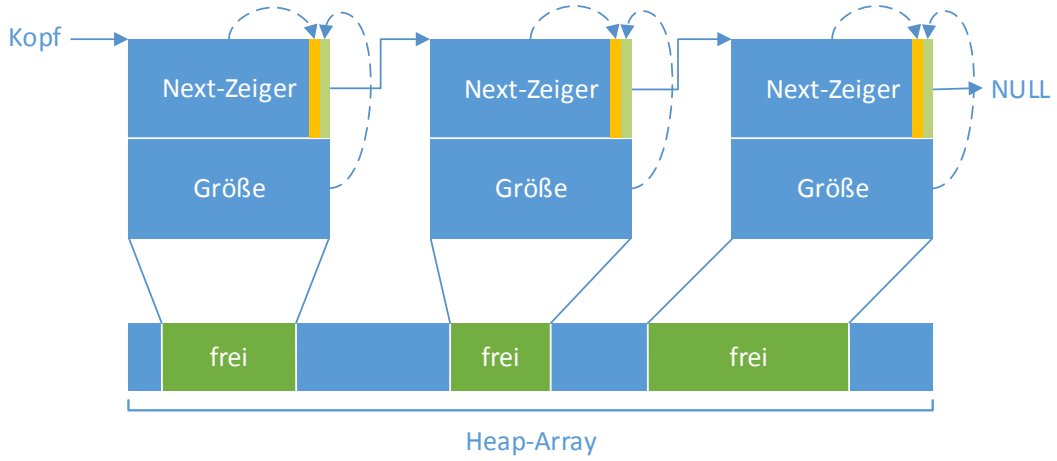


Abbildung 2.2: Schematische Darstellung des Aufbaus der Freispeicherliste und deren Abbildung auf das Heap-Array. Die gestrichelten Pfeile symbolisieren den Speicherort der Parität des Felds.

Heap am Anfang des jeweils zum Element gehörigen freien Speicherblocks abgelegt. Abbildung 2.2 zeigt eine schematische Darstellung der Freispeicherliste und deren Abbildung auf das Heap-Array.

Beim Aufruf der Allokations-Funktion wird die Freispeicherliste durch Folgen der Next-Zeiger solange durchlaufen, bis ein Element gefunden wurde, dessen Größe mindestens der angefragten Speichermenge entspricht. Stimmen diese genau überein, wird das Element aus der Liste entfernt, der Next-Zeiger des vorherigen Elements wird dabei auf die Ziel-Adresse des Next-Zeigers des zu entfernenden Elements gesetzt. Ist die zu allokierende Speichermenge geringer als die Größe des gefundenen Elements, wird dessen Größe entsprechend reduziert. Eine Änderung der Verzeigerung der Liste ist hierbei nicht notwendig, da das Element weiterhin existiert. Wird bei der Suche das Ende der Liste erreicht, ohne dass ein geeignetes Element gefunden wurde, wird das System mit einem Out-Of-Memory-Fehler beendet.

Mögliche Auswirkungen von Bitkippern

Die Datenstruktur der Freispeicherliste ist an zwei Stellen anfällig für Bitkipper. wie im letzten Abschnitt beschrieben besitzt jedes Listenelement zwei Felder. Zum einen die Größe des zugehörigen Freispeicher-Blocks und zum anderen einen Zeiger auf das nächste Element.

Eine Veränderung der Größe eines Listenelements hat zwei mögliche Auswirkungen. Wird die gespeicherte Zahl durch den Bitkipper verringert, steht kommenden Allokationen weniger Speicher zur Verfügung, obwohl theoretisch noch ein zusätzlicher freier Bereich im Heap-Array vorhanden ist. Dies ist allerdings

nicht sehr kritisch, da die Typsicherheit des Systems nicht gefährdet ist und der fehlerhafte Eintrag zudem nur bis zum nächsten Lauf des GC existiert, bei dem die komplette Freispeicherliste neu angelegt wird. Im schlimmsten Fall kann es dazu führen, dass eine Allokation fehlschlägt, und das System durch einen Out-Of-Memory-Fehler beendet wird.

Eine Erhöhung der Größenangabe kann dagegen zu einem ernsteren Fehlverhalten der Anwendung führen. In der Allokations-Funktion würde unter Umständen Speicher an die Anwendung zurückgegeben werden, der sich an einer weiter hinteren Position im Heap-Array befindet und bereits von einer früheren Allokation vergeben wurde. Dabei würde ein Teil der darin gespeicherten Objekt-Daten nach Java-Spezifikation zunächst mit Null initialisiert und später mit neuen Objektdaten überschrieben werden, was zu unvorhersagbaren Seiteneffekten wie beispielsweise falsche Typinformationen, fehlerhafte Daten oder inkorrekte Referenzen im weiteren Verlauf der Anwendung führen würde.

Ebenfalls kritisch sind die Next-Zeiger der Listenelemente. Während der Traversierung der verketteten Liste kann der Zugriff auf einen manipulierten Zeiger bei Vorhandensein einer Speicherschutzeinheit(MPU) zu sofortigen Speicherzugriffsfehlern führen, während ohne Schutzeinheit willkürlich fremder Speicher überschrieben werden könnte. Ebenfalls denkbar wäre, dass ein Zeiger auf eine Adresse innerhalb des Heap-Arrays zeigt. Beim Zugriff würde dabei kein Zugriffsfehler ausgelöst werden. Das kann dazu führen, dass bereits vergebener Speicher fälschlicherweise erneut vergeben und dabei überschrieben wird. Ein ungünstiger Fall wäre außerdem, wenn ein Next-Zeiger so manipuliert wird, dass dieser genau auf ein anderes vorheriges Element der Liste zeigt. Das Resultat wäre eine Endlosschleife in der Allokations-Funktion.

Maßnahmen zur Absicherung

Der naive Ansatz zur Absicherung der beiden Felder wäre eine triviale Bereichsprüfung. Da die Startadresse und Größe des Heap-Arrays bekannt und konstant ist, könnten ungültige Zieladressen von fehlerhaften Next-Zeigern und unrealistisch große Listenelemente leicht erkannt werden. Da, wie im letzten Abschnitt beschrieben, allerdings auch nicht sofort erkennbare Veränderungen auftreten können, wurde sich zur Fehlererkennung in dieser Arbeit für die Verwendung von Parität-Bits entschieden. Die in Abbildung 2.2 eingezeichneten gestrichelten Pfeile deuten die Speicherung der jeweiligen Parität des Feldes im zugehörigen Bit des Next-Zeigers an. Die Implementierung wird in Kapitel 3 beschrieben.

2.4.2 | Liste lokaler Referenzen (LLRefs)

Um nicht mehr genutzten Speicher freigeben zu können muss der GC alle Objekte finden, die nicht mehr von anderen Objekten referenziert sind. Nur diese Speicher-

bereiche können sicher erneut zur Verfügung gestellt werden. Dafür werden in der ersten GC-Phase, der Scan-Phase, alle lebendigen Referenzen im System gesucht und auf einen internen Working-Stack gelegt. Es gibt verschiedene Orte, an denen Objektreferenzen gespeichert sein können. Hierzu zählen statische Felder und lokale Variablen. Erstere werden vom KESO-Übersetzer in einem speziellen Array in jeder Domäne gespeichert und sind somit leicht auffindbar.

Objektreferenzen, die sich in lokalen Variablen befinden, liegen beim Aufruf der enthaltenden Funktion auf dem Stack des Ziel-Systems. Da sich der Aufbau des Stacks je nach Zielarchitektur und C-Übersetzer unterscheiden kann und es für JINO unmöglich ist die Stack-Struktur vorherzusagen, wurde in KESO eine Methode implementiert, die es unabhängig vom Aufbau des Stacks ermöglicht alle Objektreferenzen aufzufinden, die sich gerade auf jenem befinden.

Hierfür führt jeder Task eine *Liste lokaler Referenzen* (LLRefs). Abbildung 2.3 zeigt den schematischen Aufbau einer solchen Liste. Innerhalb einer Funktion werden die lokalen Objektreferenzen nicht in jeweils eigene lokale Variablen gespeichert. Stattdessen generiert JINO ein spezielles passend großes Array zu Beginn der Funktion. Dies zwingt den C-Übersetzer dazu die Objektreferenzen gruppiert auf dem Stack abzulegen. Zusätzlich wird am Ende des Arrays ein Platzhalter-Eintrag erzeugt. Dieser wird zur Verkettung der Objektreferenz-Arrays verwendet. Zunächst wird in der Haupt-Funktion eines Tasks, die einmalig in der zugehörigen, von JINO erzeugten, Initialisierungs-Funktion aufgerufen wird, der Beginn der Liste in einem dafür vorgesehenen Feld des Tasks gespeichert. Der Kopf der Liste zeigt damit auf das Objektreferenz-Array der Haupt-Funktion des Tasks. Bei jedem Aufruf einer weiteren Funktion wird ein ihr übergebener Zeiger auf den Platzhalter-Eintrag im Objekt-Array der aufrufenden Funktion auf den ersten Eintrag des eigenen Arrays gesetzt. Dadurch entsteht letztendlich eine lange Liste von Objektreferenzen, die sich alle in lokalen Variablen von Funktionen befinden. Das Ende List wird durch einen speziellen Eintrag (EOLL) markiert.

Mögliche Auswirkungen von Bitkippern

Die Liste der lokalen Referenzen wird in der Scan-Phase des GCs durchlaufen. Dieser führt ein Array mit allen blockierenden Tasks, das zunächst abgearbeitet wird. Da der GC-Task die niedrigste Priorität im System besitzt, kann er andere Tasks nicht an beliebigen Stellen unterbrechen. Nur wenn sich ein Task im wartenden Zustand befinden kann, da eine seiner Methoden beispielsweise auf ein Ereignis wartet und somit blockiert, muss dieser berücksichtigt werden. Alle anderen nicht blockierenden Tasks besitzen beim Start eines GC-Laufs keine lokalen Variablen mehr, da sie sich im suspendierten Zustand befinden müssen, um unterbrochen werden zu können.

Jeder Task besitzt neben dem bereits erwähnten Kopf der Liste auch ein Feld mit der zugehörigen Domäne. Zunächst wird daher im GC geprüft, ob die Domäne des Tasks mit derjenigen übereinstimmt, die der GC gerade säubert. Nur in

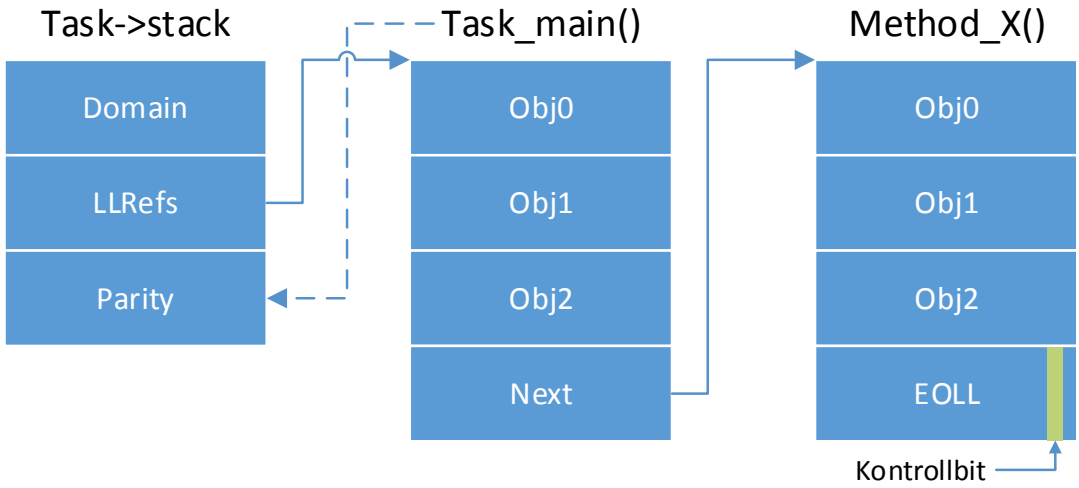


Abbildung 2.3: Schematische Darstellung des Aufbaus und der Absicherung der Liste lokaler Referenzen. Der gestrichelte Pfeil symbolisiert die einmalige Berechnung und Speicherung der Parität des Kopf-Zeigers der Liste.

diesem Fall wird die LLRefs bearbeitet. Kippt die im Feld gespeicherte Domänen-Information, kann der Task keiner bekannten Domäne zugeordnet werden. Die Speicherbereinigung wird daher nie für diese Domäne ausgeführt werden, was letztendlich zu einem Out-Of-Memory-Fehler führen kann.

Das Auftreten eines Bitkippers im LLRef-Zeiger des Tasks, der auf den Kopf der Liste zeigt, kann dazu führen, dass beim Zugriff darauf eine ungültige Adresse verwendet wird. Ebenfalls kritisch ist die Markierung des Ende der Liste. Beim Durchlaufen dieser wird bei jeder Iteration geprüft ob es sich bei dem aktuellen Eintrag um das Endsymbol handelt. Ist dieses beschädigt, kann dies dazu führen, dass die Schleife solange weiter ausgeführt wird bis eine Adresse erreicht wird, die jenseits des Endes des letzten Objekt-Arrays liegt. Dies führt je nach Konfiguration des KESO-Systems zu einer Ausnahme, da es sich bei den Daten im abgefragten Speicherbereich um kein gültiges Objekt handelt (Objekt-Header-Check) oder unter Umständen irgendwann im weiteren Verlauf zu undefinierbarem Verhalten aufgrund Auswertung der Daten ungültiger Speicheradressen.

Die Objektreferenzen selbst, welche in den Arrays abgelegt sind, sind nicht durch die in dieser Arbeit implementierten Mechanismen geschützt. KESO bietet allerdings bereits den systemweiten Mechanismus Dereference Check (DRC)[1], der alle Objektreferenzen bei Zugriff durch das Laufzeitsystem auf ihre Integrität prüft. Zur Absicherung bei weiterer Verwendung der Objektreferenz in der Anwendung kann auch der Load Reference Check (LRC)[2] aktiviert werden, der die Referenz unmittelbar nach dem Laden dieser aus dem Speicher prüft.

[1] JINO-Option: coded_ref_cxxp_parity
[2] JINO-Option: use_coded_refs_on_get_set

Maßnahmen zur Absicherung

Sowohl das Domänen-Feld, als auch der Kopf der Liste kann durch eine Parität abgesichert werden. Um Platz zu sparen, teilen sich beide Felder ein Parität-Byte, das ebenfalls als Task-Feld definiert ist. Details zur Implementierung dieser Bitmasken werden im nächsten Kapitel 3 erläutert. Ist die im letzten Abschnitt erwähnte DRC-Option aktiviert, wird für den Kopf der Liste keine zusätzliche Parität berechnet, da es sich hierbei in diesem Fall bereits um einen geschützten Objekt-Zeiger handelt. Ist dies nicht der Fall, wird die Parität des Kopfes der Liste einmalig beim Aufruf der Haupt-Methode des Tasks berechnet und gespeichert. In Abbildung 2.3 ist dieser Schritt durch den gestrichelten Pfeil dargestellt. Die Domäne wird hingegen immer gesichert.

Als Endmarkierung der Liste wurde ~1 (0xFFFFFFFE bzw. [...]1111 1110)[3] gewählt. Grund hierfür ist, dass das niedrigste Bit der ersten lokalen Objekt-Referenz jedes Objekt-Arrays dazu verwendet wird, um einen neuen Funktions-Abschnitt zu kennzeichnen. Dieses Bit ist bei der gewählten Markierungsmaske nicht gesetzt, wodurch einer Verwechslung vorgebeugt wird. Die restlichen Bits sind dagegen auf Eins gesetzt. Dadurch kann ein Bitkipper leicht erkannt werden. Bei jeder Iteration der Liste wird geprüft, ob die höheren N-2 Bits dem Wert 0xFFFFFFFC bzw. [...]1111 1100^2 entsprechen. Trifft dies zu und ist zusätzlich das zweitniedrigste Bit auf Eins gesetzt, handelt es sich um das Ende der Liste. Wird nur eine der beiden Bedingungen erfüllt, handelt es sich um das Endsymbol, das jedoch durch einen Bitkipper verändert wurde. Sind beide Vergleiche negativ, kann es sich entweder um eine gültige Objektreferenz, und somit nicht um das Ende der Liste, handeln oder es sind zwei Bitfehler aufgetreten. Letzteres genügt dem gewählten Fehlermodell dieser Arbeit nicht und wird daher nicht erkannt. Im Fall eines 1-Bit-Fehlers kann dieser ohne zusätzlichen Aufwand korrigiert werden, indem der Marker neu geschrieben wird.

2.4.3 | Working-Stack

Während der im letzten Abschnitt beschriebenen Scan-Phase des GC werden alle gefundenen Objektreferenzen auf einen internen Working-Stack gelegt. Dabei handelt es sich um ein Array, dessen Größe in der KESO-Konfiguration vor der Übersetzung festgelegt werden kann. Ist dies nicht der Fall, entspricht die Größe der Anzahl der Speicher-Slots, die der Heap verwaltet.

In der Markierungs-Phase läuft der GC alle auf dem Working-Stack abgelegten Objekte ab und markiert diejenigen in einer Bitmap, welche von anderen Objekten referenziert werden. Die referenzierenden Objekte werden daraufhin ebenfalls auf den Working-Stack gelegt und in einer späteren Iteration selbst überprüft.

[3] Die Bits der höheren 3 Byte sind alle auf Eins gesetzt und wurden der Lesbarkeit halber hier nicht in voller Form aufgeführt

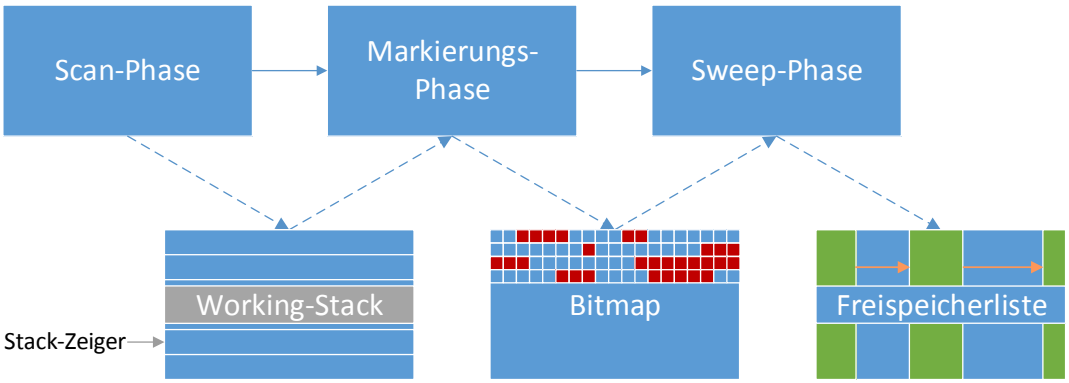


Abbildung 2.4: Schematische Darstellung der GC-Phasen und der jeweils verwendeten Datenstrukturen.

Der GC stellt dabei sicher, dass jedes Objekt nur maximal einmal auf den Stack gelegt werden kann, indem er ein Objekt einfärbt, nachdem es auf den Stack gelegt wurde.

Mögliche Auswirkungen von Bitkippern

Um die aktuelle Position beim Abarbeiten des Working-Stacks zu speichern, wird ein Zeiger verwendet. Dabei handelt es sich um den Index des nächsten freien Eintrags im Working-Stack-Array. Wird dieser durch einen Bitkipper beeinflusst, kann dies zu vier Resultaten führen.

Wird der Wert des Zeigers verringert, werden im weiteren Verlauf der Markierungs-Phase nicht alle Objekte bearbeitet, da der Zeiger beginnend ab einem niedrigeren Index dekrementiert wird, obwohl an höheren Indizes noch auf Objekte verwiesen wird. Werden beim Analysieren von Objekten auf niedrigeren Indizes weitere Objekte auf den Stack gelegt, was zu einer Inkrementierung des Zeigers führt, werden unter Umständen fälschlicherweise noch nicht bearbeitete Objekte überschrieben. Dadurch würden Objekte, die noch lebendig sind, später nicht als solche markiert werden. Der GC würde diese anschließend fälschlicherweise freigeben und zukünftigen Allokationen zur Verfügung stellen.

Wird der Zeiger so verändert, dass er auf einen höheren Index verweist, kann dies zu drei Ergebnissen führen. Ist der Index weiterhin gültig, kann dieser auf einen Eintrag im Stack-Array zeigen, der in einem früheren GC-Lauf für ein anderes Objekt verwendet wurde. Dadurch würde das Objekt als lebendig eingestuft und in der Bitmap entsprechend markiert werden, obwohl dies unter Umständen nicht mehr der Fall ist. Ebenfalls könnte der Zeiger auf einen leeren Eintrag zeigen. Dies würde zu einer Ausnahme führen, da sich hinter dem Eintrag kein gültiger Objekt-Header befinden würde. Als letzte Möglichkeit könnte der Index einen Wert annehmen, der größer als der Maximal-Index des Stack-Arrays

ist. Dabei würde auf eine ungültige Speicheradresse zugegriffen werden, was zu undefinierbarem Verhalten führen kann.

Maßnahmen zur Absicherung

Neben einer trivialen Überprüfung, ob der Stack-Zeiger einen Wert zwischen Null und dem Maximal-Index des Stack-Arrays besitzt, kann zur Absicherung alternativ abermals eine Parität verwendet werden. Diese wird bei jedem Laden des Stack-Zeigers aus dem Speicher in eine lokale Variable überprüft. Eine Überprüfung vor jeder Verwendung dieser lokalen Variable als Index beim Zugriff auf das Array ist nicht notwendig, da der Wert in einem Register gespeichert ist (siehe Auflistung 2.1) und diese nach dem Fehlermodell dieser Arbeit als sicher gelten.

```
movzwl 0x13c340, %eax    ; Lade Stack-Index aus dem Speicher
test   %ax, %ax          ; Schleifen-Bedingung
je     101bb0 <keso_ft_coffee_sweep_begin>
sub    $0x1, %eax        ; Dekrementiere Stack-Index im Register
mov    %ax, 0x13c340     ; Speichere dekrementierten Stack-Index
movzwl %ax, %eax
mov    0x13c360(, %eax, 4), %eax ; Greife auf das Stack-Array zu
```

Auflistung 2.1: Zugriff auf das Working-Stack-Array über den Stack-Zeiger

Die Objektreferenzen, die auf dem Stack abgelegt werden, können ebenfalls durch eine Parität gesichert werden. Dieser Mechanismus wird bei der Implementierung dieser Arbeit nur aktiviert, wenn die systemweite Referenzüberprüfung mittels DRC deaktiviert ist. Ist diese aktiviert, ist eine zusätzliche Parität überflüssig.

2.4.4 | Bitmap

Die in der Scan-Phase des GC gefundenen lebendigen Objekte werden in einer darauf folgenden Phase, der Markierungs-Phase, in einer Bitmap markiert. Dabei wird nicht für jedes Byte des Heaps ein Bit in der Bitmap zur Verfügung gestellt. Dies wäre Verschwendung von Speicherplatz. Stattdessen ist der Heap in Slots unterteilt. Ein Bit in dieser Maske entspricht dabei einem Slot, den ein Objekt im Heap belegt. Die Slot-Größe kann in der Konfiguration vor der Übersetzung festgelegt werden. JINO überprüft dabei, dass eine minimale Slot-Größe eingehalten wird, denn die Header der Objekte besitzen schon eine Größe von vier Byte. Auch die Datenstruktur der in einem vorherigen Abschnitt beschriebenen Freispeicherliste belegt bereits mindestens vier Byte. Kleinere Slots sind aus diesem Grund nicht konfigurierbar.

Die Bitmaske ist vor der Markierungs-Phase mit Null initialisiert. Alle Slots der gefundenen lebendigen Objekte werden in dieser Phase mit Eins in der Bitmap markiert. In einer anschließenden letzten Sweep-Phase kann der GC effizient die Bitmaske schrittweise durchlaufen und alle zusammenhängenden mit Eins markierten Bereiche in Adressen im Heap-Array umrechnen und als freie Speicherblöcke an die Freispeicherliste anhängen.

Mögliche Auswirkungen von Bitkippern

Bitkipper in der Bitmap können zu zwei Resultaten in der Sweep-Phase führen. Zum einen kann ein fälschlicherweise auf Eins gesetztes Bit dazu führen, dass ein eigentlich nicht mehr gebrauchter Slot im Heap-Array nicht freigegeben wird. Das führt zur Reduzierung des durch den GC wieder zur Verfügung gestellten Speichers. Da nur einzelne Slots betroffen sind, ist die unmittelbare Auswirkung gering, zumal die übergangenen Slots beim nächsten GC-Lauf freigegeben werden. Die eventuell dadurch erzeugte Lücke in einem größeren markierten Bereich erhöht allerdings die Fragmentierung des Speichers vorübergehend bis zum nächsten GC-Lauf, da nun zwei Elemente statt einem zur Freispeicherliste hinzugefügt werden.

Kritischer ist die Änderung eines Eins-Bits. Dadurch wird ein Slot als nicht mehr lebendig markiert. Die Sweep-Phase würde in diesem Fall den zugehörigen Speicher in die Freispeicherliste einhängen, was im weiteren Verlauf der Anwendung dazu führen kann, dass dieser Speicherbereich für eine neue Allokation vergeben wird. Dadurch würden Bereiche eines Objekts überschrieben werden, das noch in Verwendung ist. Dies kann zu einem unvorhersagbaren Verhalten der Anwendung führen.

Maßnahmen zur Absicherung

Vor der Sweep-Phase muss sichergestellt sein, dass sich die Bitmap in einem konsistenten Zustand befindet. Hierfür kann nach der Markierungs-Phase eine Parität der Bitmap berechnet und abgespeichert werden. Die Berechnung kann bei einem mehrere Byte großen Bitmap-Array aufwendig sein (Details siehe Kapitel 4). Daher wird aus Effizienzgründen stattdessen die Parität der Anzahl der Slots, die in der Markierungs-Phase auf Eins gesetzt werden, verwendet. Dies funktioniert, da die Bitmap vor jedem GC-lauf mit Null initialisiert wird und somit vor der Markierungs-Phase stets denselben Zustand besitzt.

Da der Coffee-Break nicht unterbrochen werden kann, ist es nicht möglich, dass durch Bitkipper in der Bitmap fälschlicherweise freigegebener Speicher an eine unterbrechende Allokation übergeben wurde. Somit ist die Anwendung von Bitkippern während der Sweep-Phase nicht betroffen. Es genügt daher, einmalig nach der Sweep-Phase die Parität der Bitmap zu berechnen. Stimmt diese mit der zuvor berechneten Parität überein, wurden genauso viele Slots freigegeben,

wie in der Markierungs-Phase zuvor markiert wurden. Ist dies nicht der Fall, sind Bitkipper während der Sweep-Phase aufgetreten und das System reagiert mit einer Ausnahme. Ein aufwendiges Prüfen der Bitmap bei jedem Zugriff wird dadurch vermieden.

2.4.5 | Domänenzeiger

Jeder Domäne kann eine andere Variante der Speicherverwaltung zugewiesen werden. Es ist beispielsweise eine Kombination aus Restricted-Domain-Scope und Coffee-Break möglich, um den Aufwand einfacher Tasks durch Verzicht auf eine automatische Speicherbereinigung gering zu halten, während in komplexeren Programmteilen weiterhin ein GC zum Einsatz kommt.

Der GC-Lauf wird mit einem Index-Parameter gestartet, welcher in einem internen Zeiger gespeichert wird. Dieser Index gehört zu einem intern im GC hinterlegten Array, welches Verweise auf alle Domänen beinhaltet, die von dieser GC-Variante verwaltet werden. Dadurch ist es an vielen Stellen im Programm-Code des GC möglich, schnell anhand dieses Index auf Informationen wie beispielsweise die Adresse des Heap-Arrays oder die Slot-Größe der gerade zu untersuchenden Domäne zuzugreifen.

Mögliche Auswirkungen von Bitkippern

Sowohl im Domänen-Array als auch im Zeiger auf die aktuelle Domäne können Bitkipper auftreten. Ein Fehler in den Einträgen des Arrays, also den Zeigern auf die jeweiligen Domänen-Strukturen, kann zur Verarbeitung ungültiger Daten und beliebigem undefinierten Verhalten führen, sobald der GC auf Informationen einer Domäne zugreifen will.

Ein beschädigter Domänen-Zeiger kann einerseits ebenfalls zu einem undefinierten Verhalten führen, wenn der Wert des Index den maximal erlaubten Index des Arrays überschreitet und dadurch auf eine ungültige Adresse zugegriffen wird. Der Zeiger kann während einer GC-Phase ebenfalls so beeinflusst werden, dass der Wert weiterhin einen gültigen Index des Domänen-Arrays darstellt. Auch hier kann ein unvorhersagbares Verhalten auftreten. Beispielsweise könnte in der Sweep-Phase Speicherbereiche des Heaps einer anderen Domäne freigegeben werden, obwohl die darin befindlichen Objekte zuvor nicht analysiert wurden.

Maßnahmen zur Absicherung

Zur Absicherung der Array-Einträge können diese mit einem Parität-Bit versehen werden. Da die Domänen-Strukturen auf 4-Byte-ausgerichteten (Alignment) Adressen abgelegt sind, kann hierfür das niedrigste Bit der Zeiger verwendet werden.

Beim Domänen-Index handelt es sich um eine Ganzzahl, deren Bits nicht für die Speicherung einer Parität zweckentfremdet werden können. Daher wird diese in einem eigenen speziellen separat geführten Parität-Byte gespeichert. Details werden im folgenden Kapitel 3 beschrieben.

2.5 | Latenzgewahre Speicherbereinigung (Idle-Round-Robin)

Neben der im letzten Abschnitt vorgestellten durchsatzstarken Speicherbereinigung (Coffee-Break) bietet KESO darüber hinaus noch eine dritte Implementierung einer Speicherverwaltung. Diese *Idle-Round-Robin* (IRR) genannte Variante ist mehr auf die Anforderungen von Echtzeitsystemen ausgerichtet. Bei eingebetteten Systemen ist es oft maßgeblich, dass feste Zeitschranken bei der Ausführung von Aufgaben oder Reaktion auf Ereignissen eingehalten werden. Die verwendete Speicherverwaltung darf dieses Verhalten nicht verfälschen. Der IRR arbeitet genauso wie der in Abschnitt 2.4 vorgestellte Coffee-Break nur dann, wenn kein anderer Task in der Domäne aktiv ist. Allerdings kann er im Gegenzug zusätzlich bei Bedarf in manchen Phasen unterbrochen werden. Aufgrund der auftretenden Fragmentierung der Freispeicherliste ist die maximale Laufzeit(WCET) allerdings hoch, weswegen er nicht gänzlich für Echtzeitsysteme geeignet ist.

Um die Unterbrechungen zu ermöglichen muss der GC mit den anderen Tasks der Domäne synchronisiert werden. In den beiden folgenden Abschnitten werden die Unterschiede zum Coffee-Break beschrieben und bezüglich ihrer Fehleranfälligkeit analysiert.

2.5.1 | Freispeicherliste

Die Freispeicherliste des Idle-Round-Robin ist ähnlich aufgebaut wie die in Abschnitt 2.4.1 beschriebene Implementierung des Coffee-Break. Der Unterschied besteht darin, dass das Durchlaufen der Liste unterbrochen werden kann. Beispielsweise kann eine Allokation während der Suche nach einem geeignet großen Speicherblock von einer weiteren Allokation pausiert werden. Es wäre in diesem Fall unter Umständen denkbar, dass ein Listenelement von der zweiten Allokation aus der Liste entfernt werden würde, wenn die Größe des Elements des angeforderten Speichers entspricht. Wäre die erste Allokation aber genau dann unterbrochen worden, als sie gerade ebenfalls dieses Element auf seine Eignung geprüft hat, würde sie nach Abschluss der zweiten Operation eine fehlerhafte Datenstruktur vorfinden.

Es ist daher essentiell sicherzustellen, dass die Konsistenz der Liste trotz Unterbrechbarkeit gewährleistet ist. Um dies zu realisieren, besitzt jedes Listenelement neben des Next-Zeigers und der Größe ein weiteres *Flags-Feld*. Ein Bit dieses

Feldes wird dazu verwendet, um Elemente zu sperren. Diese gesperrten Listenelemente können nicht aus der Liste entfernt, wohl aber verkleinert werden.

Mögliche Auswirkungen von Bitkippern

Neben den in Abschnitt 2.4.1 beschriebenen möglichen Auswirkungen bei Beeinflussung des Next-Zeigers und der hinterlegten Element-Größe ist das Sperrbit des IRR ebenfalls als kritische Datenstruktur hinsichtlich Bitkipper zu werten. Ein Kippen dieses Bits kann zu Problemen bei der Synchronisation der Listen-Traversierung führen. Um das obige Beispiel aufzugreifen: Ohne vorhandene Sperrung des Elements, wäre es denkbar, dass beide Allokationen denselben Speicherbereich an die Anwendung zurückgeben. Dies würde im Endeffekt dazu führen, dass ein Objekt überschrieben wird. Ein fehlerhaftes Verhalten der Applikation ist dadurch sehr wahrscheinlich.

Maßnahmen zur Absicherung

Für die Absicherung des Flag-Bytes, welches das Sperrbit beinhaltet, kann eine Parität verwendet werden. Beim Coffee-Break wurden die beiden niedrigsten Bits des Next-Zeigers verwendet, um die Paritäten des Next-Zeigers selbst und die der Element-Größe zu speichern. Da in diesem Fall aber noch ein drittes Feld hinzukommt und aufgrund des Alignments nur zwei ungenutzte Bits zur Verfügung stehen, kann diese Methode nicht angewandt werden. Stattdessen können allerdings die weiteren unbenutzten Bits des Flags-Feldes herangezogen werden. Dadurch wird ebenfalls eine unnötige Vergrößerung der Datenstruktur vermieden. Eine detailliertere Beschreibung der Verwendung von Bitmasken folgt in Kapitel 3.

2.5.2 | Weitere Unterschiede zur durchsatzstarken Speicherbereinigung

Der Idle-Round-Robin verwendet größtenteils dieselben Datenstrukturen, die auch beim Coffee-Break zum Einsatz kommen. Neben der im letzten Abschnitt beschriebenen Unterschiede der Freispeicherliste, müssen zwei weitere Datenstrukturen gesondert betrachtet werden.

Bitmap

Beim Coffee-Break genügte es, die Integrität der Bitmap einmal nach der Sweep-Phase zu prüfen. Da diese Phase dort nicht durch Allokationen unterbrochen werden kann, besteht keine Gefahr, dass fälschlicherweise in die Freispeicherliste

angehängte Speicherbereiche an die Anwendung vergeben werden. Denn vor der ersten neuen Allokation wird die Bitmap erneut überprüft.

Beim IRR kann die Sweep-Phase jedoch durch Interrupts unterbrochen werden. Tritt während dieser Phase ein Bitkipper auf, der eine Eins in eine Null ändert, würde das dazu führen, dass ein Speicherbereich, der zu einem noch lebendigen Objekt gehört freigegeben wird. Eine danach während der Sweep-Phase unterbrechende Allokation könnte genau diesen Speicherbereich wählen, was zu einem unvorhersagbaren Verhalten führen kann. Als Gegenmaßnahme muss daher vor jeder Allokation geprüft werden, ob diese die Sweep-Phase des GC verdrängt hat und in diesem Fall die Bitmap erneut prüfen. Die Sweep-Phase des IRR initialisiert die freigegebenen Speicherbereiche im Gegensatz zum Coffee-Break, bei dem dies erst bei der Allokation selbst geschieht, sofort. Daher muss die Überprüfung der Bitmap ausgeweitet werden. Diese ist nicht nur vor jeder Allokation notwendig, sondern generell beim Aufruf des Interrupt-Handlers, was ebenfalls Allokationen mit einschließt.

Colorbit

Aufgrund der Unterbrechbarkeit des IRR muss sichergestellt werden, dass nur nicht mehr verwendete Objekte freigegeben werden, die vor Beginn des aktuellen GC-Laufs allokiert wurden. Objekte, die von einer Allokation erzeugt wurden, die den aktuellen GC-Lauf unterbrochen haben, dürfen in diesem Lauf nicht bearbeitet werden. Um eine Unterscheidung zu ermöglichen, werden die Objekte eingefärbt, indem ein spezielles Bit im Objekt-Header gesetzt wird oder nicht. Die Farbe der jeweiligen Domäne wird zu Beginn des GC-Laufs in eine interne Variable kopiert. Anschließend wird der Domäne eine neue Farbe zugewiesen, indem das Bit invertiert wird. Alle bis zum nächsten GC-Lauf allokierten Objekte erhalten dadurch eine andere Farbe als bisherige Objekte. Der IRR kann dadurch prüfen, ob die Farbe eines Objekts mit der zuvor kopierten Farbe übereinstimmt und dieses nur dann behandeln.

Ein Bitkipper in der Farb-Kopie könnte dazu führen, dass sich die Farbe vorzeitig ändert. Das kann dazu führen, dass Objekte fälschlicherweise freigegeben und deren Speicherbereiche mit Null initialisiert werden. Im weiteren Verlauf der Anwendung kann der Zugriff auf solche Objekte in Null-Pointer-Exceptions oder anderes undefinierbares Verhalten resultieren.

Zur Absicherung der Farb-Information kann diese durch eine Parität geschützt werden. Das Parität-Byte, das bereits zur Absicherung des Stacks, der Bitmap und des Domänen-Zeigers verwendet wird, kann dafür als Speicherort herangezogen werden.

2.6 | Zusammenfassung

Die KESO-Multi-JVM bietet aufgrund der typsicheren Ausgangssprache (Java) ihren Anwendungen eine automatische Speicherverwaltung und -bereinigung an. Der Entwickler kann zwischen drei Varianten wählen. Jede der in diesem Kapitel vorgestellten Implementierungen verwendet intern diverse Datenstrukturen. Während die Anwendung und andere Teile der KESO-Laufzeitumgebung wie Objektreferenzen bereits durch Mechanismen wie LRC und DRC abgesichert sind, fehlte bisher ein Schutz der GC-Strukturen. Es wurde analysiert, zu welchen Auswirkungen einzelne Bitkipper in den verwendeten Datenstrukturen führen können. Des Weiteren wurde für jeden kritischen Bereich eine mögliche Lösung zur Absicherung vorgestellt.

Im folgenden Kapitel 3 werden technische Details zur Implementierung der Schutzmechanismen beschrieben.

3 | Implementierung

Im letzten Kapitel wurden die verschiedenen in KESO implementieren Speicherverwaltungs-Varianten vorgestellt. Des Weiteren wurde analysiert, wie die zugehörigen Datenstrukturen erweitert werden müssen, um sie gegen transiente Fehler abzusichern. Außerdem wurde diskutiert, welche potentiellen Auswirkungen nicht erkannte Fehler haben können. Dieses Kapitel befasst sich mit den im Rahmen dieser Arbeit implementierten Schutzmechanismen und deren Konfiguration.

3.1 | Vorüberlegungen

Generell gibt es bei der software-basierten Fehlertoleranz drei Ansätze. Zum einen können zur Fehlererkennung zusätzlich Paritäten oder Prüfsummen berechnet und gespeichert werden. Des Weiteren besteht die Möglichkeit, Fehler nicht nur zu erkennen, sondern diese in gewissem Umfang auch für die Applikation transparent zu korrigieren. Hierfür kann man zum einen auf Replikation bzw. redundante Speicherung zurückgreifen und zum anderen auf spezielle Datenstrukturen, die schon aufgrund ihres Designs tolerant gegenüber einer festgelegten Anzahl an Bit-Fehlern sind. In den folgenden Abschnitten werden diese drei Methoden detailliert anhand von Beispielen vorgestellt und diskutiert.

3.2 | Feingranular konfigurierbare Fehlertoleranz

Die im Laufe dieser Arbeit in das KESO-System integrierten Schutzmechanismen für die GCs wurden so entworfen, dass sie pro Domäne gezielt konfiguriert und einzeln aktiviert bzw. deaktiviert werden können. Zusätzlich ist es für einige Datenstrukturen möglich, die Stufe der Fehlertoleranz zu definieren. Dabei kann festgelegt werden, ob gar keine Überprüfung auf Fehler stattfinden soll, ob diese nur erkannt und mit einer Exception behandelt werden sollen oder ob versucht werden soll, auftretende Fehler zu korrigieren.

Zusätzlich erlaubt die Konfiguration die Festlegung einer Gesamtstufe (FTLevel) für die jeweilige Domäne. Dies ermöglicht es dem Entwickler schnell alle Daten-

strukturen abzusichern, und nur gezielt einzelne mit einer geringeren Stufe zu versehen.

Je nach Zielsystem, Applikation und weiteren Anforderungen wie möglichst geringer Laufzeit oder Programmgröße kann der Entwickler bereits vor der Übersetzung entscheiden, welche Features er für die jeweiligen Domänen seiner Anwendung einsetzen möchte. Das hat den Vorteil, dass JINO intelligent entscheiden kann, welche Code-Teile benötigt werden und mit Hilfe von C-Makros entsprechend effizienten ANSI-C-Code erzeugen kann, was sich sowohl auf die Größe der am Ende erzeugten Binärdatei als auch auf die Laufzeit des Systems auswirken kann. Die genauen Unterschiede werden anhand von Beispielen im folgenden Kapitel 4 diskutiert.

KESO unterstützt dabei die folgenden Konfigurations-Optionen, die in der Heap-Definition jeder Domäne justiert werden können und die Stufe der Fehlererkennung bestimmen:
FTLevel, FTLevelFreeMemList, FTLevelWorkingStack, FTLevelDomainPointer, FTLevelBitmap, FTLevelLocalRefs, FTLevelColoredObjectBits

Der Wert `0` schaltet die Erkennung dabei ab, während der Wert `1` die Verwendung von Paritäten zur Absicherung aktiviert. Ein höherer Wert bewirkt bei den Datenstrukturen Heap-Zeiger(RDS) und End-Zeiger(RDS), sowie Working-Stack, Domänenzeiger und Color-Bit den Einsatz von Replikation und bestimmt dabei die Anzahl der korrigierbaren Fehler.

3.3 | Fehlererkennung

Um eine reine Fehlererkennung zu ermöglichen wurden in dieser Arbeit verschiedene Methoden zur Speicherung von Parität-Bits implementiert.

Paritäten

Paritäten stellen die einfachste Form von Prüfsummen dar. Sie sind einfach und effizient zu berechnen und viele Architekturen bieten bereits native Instruktionen dafür an. Andere schon in KESO implementierte Schutzmechanismen wie Header- und Referenzüberprüfungen (siehe Abschnitt 1.2) verwenden bereits Paritäten um Typ- und Meta-Informationen der Laufzeitumgebung abzusichern.

Alle in dieser Arbeit gesicherten Datenstrukturen werden automatisch vor dem Zugriff auf ihre Integrität hin überprüft. Stellt der Mechanismus eine Inkonsistenz zwischen zu über prüfendem Wert und hinterlegter Parität fest, wird eine Ausnahme erzeugt.

Auflistung 3.1 zeigt die in KESO verwendeten Parität-Funktionen. Falls verfügbar, wird die spezielle GCC[1]-Funktion `__builtin_parity` verwendet. Diese wird

[1] GNU Compiler Collection

in eine native Prozessor-Instruktion übersetzt, falls die Architektur eine zur Verfügung stellt. Auf den in Kapitel 4 verwendeten Architekturen TriCore und x86 wird diese unterstützt. Sollte kein GCC-Übersetzer verwendet werden, kommen selbst implementierte Funktionen zum Einsatz.

Zur Speicherung der Paritäten der einzelnen im letzten Kapitel beschriebenen Datenstrukturen wurden im Rahmen dieser Arbeit verschiedene Möglichkeiten verwendet. Die naive Methode wäre für jede Struktur ein eigenes Parität-Byte an geeigneter Stelle zu definieren. Allerdings kann das zu einer Verschwendung von Ressourcen führen. Da in diesem Fall nur ein Bit, meist das niedrigste, verwendet wird, sind die anderen sieben Bit überflüssig. Wird dieses Byte zusätzlich an einer Stelle definiert (z.B. in einem Struct), kann der Speicherbedarf aufgrund der Speicherausrichtung der Adressen an 4-Byte-Offsets (Alignment) weiter ansteigen. In diesem Beispiel würden 31 Bit ungenutzt bleiben.

```
#if defined(__GNUC__)
    #define KESO_GET_PARITY(x) __builtin_parity((unsigned int) x)
#else
    #define KESO_GET_PARITY4(x)  ((0x6996 >> ((x) & 0xf)) & 1)
    #define KESO_GET_PARITY8(x)  keso_get_parity8(x)
    #define KESO_GET_PARITY16(x) keso_get_parity16(x)
    #define KESO_GET_PARITY(x)   keso_get_parity32(x)

    static inline unsigned int keso_get_parity8(unsigned int x) {
        return KESO_GET_PARITY4((x) ^ ((x) >> 4));
    }

    static inline unsigned int keso_get_parity16(unsigned int x) {
        return KESO_GET_PARITY8((x) ^ ((x) >> 8));
    }

    static inline unsigned int keso_get_parity32(unsigned int x) {
        return KESO_GET_PARITY16((x) ^ ((x) >> 16));
    }
#endif
```

Auflistung 3.1: Parität-Funktionen in KESO

Bitmasken

Bei einigen Datenstrukturen wie dem Domänenzeiger, dem Working-Stack-Zeiger oder der Anzahl der Einsen in der Bitmap handelt es sich um simple Ganzzahlen. Statt drei einzelner Parität-Bytes werden die Werte kombiniert in einem einzigen Byte gespeichert. Pro Parität wird ein bestimmtes Bit auf Eins oder Null gesetzt. Dieses Verfahren wird als Bitmaske bezeichnet.

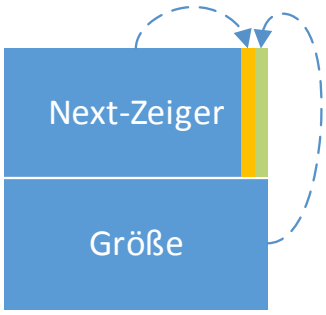


Abbildung 3.1: Schematische Darstellung des Aufbaus eines Elements der Freispeicherliste. Die gestrichelten Pfeile symbolisieren den Speicherort der Parität des jeweiligen Felds.

Zeigermanipulation

Die Parität einiger Zeiger muss nicht in einem separaten Byte gespeichert werden. Die Zieladressen der Next-Zeiger der Freispeicherliste des Coffee-Break beispielsweise sind immer ein Vielfaches von Vier. Das ergibt sich aufgrund des Alignment und der Vorschrift, dass die Heap- und Slot-Größe eine Zweierpotenz sein muss. Des Weiteren ist die minimale Slot-Größe auf Vier Byte festgelegt. Die niedrigsten zwei Bit der Adressen sind dadurch immer mit Null belegt. Diesen Umstand macht sich diese Implementierung zu Nutze und verwendet die beiden Bits um die Paritäten der Listenelementgröße und des Next-Zeigers in diesen selbst zu speichern. Abbildung 2.2 auf Seite 13 zeigt die Gesamtstruktur der Freispeicherliste, Abbildung 3.1 den Aufbau eines Listenelements im Detail. Die eingezeichneten gestrichelten Pfeile stellen den Speicherort der Parität der jeweiligen Felder dar. So wird Bit Null für die Parität des Next-Zeigers verwendet, während Bit Eins für die der Größe reserviert ist. Eine unnötige Vergrößerung der Datenstruktur kann somit vermieden werden.

Bei jedem Aufruf des Zeigers, wird das Extraktions-Makro `((uintptr_t)pointer & ~3)` aufgerufen, das die beiden niedrigsten Bits vor der Verwendung auf Null zurücksetzt.

3.4 | Fehlerkorrektur

Neben der reinen Erkennung von Fehlern wurden in dieser Arbeit zusätzlich ein Mechanismus zur Korrektur implementiert.

3.4.1 | Replikation

Für einfache Strukturen wie Ganzzahlen kann Replikation verwendet werden. Dabei wird zusätzlich eine festgelegte Anzahl an Kopien des Werts gespeichert. Sie ergibt sich aus der einfachen Formel F_k*2+1, wobei F_k für die gewünschte Anzahl korrigierbarer Fehler steht. Beim Setzen eines neues Wertes einer replizierten Variable müssen auch alle Kopien gleichermaßen geändert werden. Beim Lesen des Wertes wird anhand eines Mehrheits-Algorithmus [BM82] unter Betrachtung aller Replikate der Variablen der korrekte Wert ermittelt.

Zur Vereinfachung wurden in dieser Arbeit die sogenannten *Resilient-Variablen* eingeführt. Diese erlauben durch ihre Getter- und Setter-Funktionen einen einfachen Lese- und Schreib-Zugriff auf die Replikate einer Variable. Auflistung 3.2 zeigt den Getter einer Resilient-Variable des Typs `int` mit dem Namen `foo`. Der zweite Funktions-Parameter `ft_level` gibt die Stufe der Fehlertoleranz an. Ist dieser Wert Null, werden keine Replikate der Variablen geschrieben oder gelesen. Bei Stufe Eins werden nach obiger Formel drei Replikate verwaltet.

```
#define NUMBER_OF_COPIES(f) (2 * ((f > 0) ? (f) : 0) + 1)

int keso_resilient_foo_get(keso_resilient_foo_t *var,
    keso_ft_level_t ft_level){
    unsigned char counter = 1;
    int v = var->value[0];

    for(unsigned int i=1; i < NUMBER_OF_COPIES(ft_level - 1); i++){
        counter = counter + (v == var->value[i] ? 1 : -1);

        if(counter == 0){
            v = var->value[i];
            counter++;
        }
    }

    return v;
}
```

Auflistung 3.2: Funktion zum Auslesen des Werts einer replizierten Variable

Nachteil dieses Mechanismus ist die durch die Replikate erzeugte größere Angriffsfläche für transiente Fehler. Die Wahrscheinlichkeit von Bitkippern ist dadurch höher als bei einfachen Variablen.

3.5 | Zusammenfassung

In diesem Kapitel wurden die technischen Details der verschiedenen Absicherungsmechanismen beschrieben. Dabei wurde darauf geachtet, dass diese sowohl effizient als auch ressourcensparend implementiert sind. Ein Vergleich zur bisherigen Implementierung der Speicherverwaltungen wird anhand ausgewählter Experimente und Messungen im nächsten Kapitel 4 diskutiert.

4 | Evaluation

In den beiden vorherigen Kapiteln wurde das Design und die Implementierung einer Fehlererkennung für die drei verfügbaren GC-Varianten in KESO detailliert beschrieben. Dieses Kapitel evaluiert und diskutiert die Effektivität und Effizienz der neu implementierten Mechanismen anhand verschiedene Experimente und Messungen.

4.1 | Die Testanwendung CD_x

Testanwendungen (Benchmarks) sind eine gängige Herangehensweise um Optimierungen und Erweiterungen einer Software mit dem vorherigen Zustand dieser zu vergleichen. Hierbei ist es wünschenswert eine Anwendung zu verwenden, die einer in der Realität eingesetzten Applikation nahe kommt um aussagekräftige Ergebnisse erzielen zu können. Frühere Veröffentlichungen zu KESO [SSE+13, Sti12] verwendeten hierfür den Benchmark CD_j, der als Teil der CD_x-Sammlung erstmals in [KHP+09] veröffentlicht wurde. Diese wird beschrieben als eine "open-source real-time Java benchmark family that models a hard real-time aircraft

	CD_x on-the-go
CPU:	Infineon TriCore TC1796
Speicher:	2 MiB Flash, 1 MiB SRAM
Betriebssystem:	CiAO be3999c
Übersetzer:	GCC 4.5.2, Binutils 2.20
KESO:	r3866

Tabelle 4.1: Bei den Messungen verwendete Hard- und Software-Konfiguration.

collision detection application."

Die Anwendung besteht aus zwei Kernkomponenten. Der *Flugverkehrssimulator* (air traffic simulator, ATS) erzeugt dabei periodisch Radar-Abschnitte, die Positionen von Flugzeugen beinhalten. Diese werden der anderen Komponente, dem *Kollisionsdetektor* (collision detector, CD), anschließend in einer Warteschlange zur Verfügung gestellt, der darin potentielle Kollisionen erkennt.

Die Messungen wurden anhand zweier Konfigurationen durchgeführt:

1. "on-the-go": Bei dieser Variante werden die Radar-Abschnitte im selben Faden generiert, in dem auch der CD ausgeführt wird. Die Erzeugung erfolgt auf Bedarf vor der Detektion. Dadurch entfällt die Verwendung der Warteschlange, was allerdings ein weniger realistisches Szenario darstellt.

2. "simulated": Im Gegenzug zur ersten Variante wird der ATS hier in einem eigenen Faden ausgeführt, der in eine eigene KESO-Domäne eingebunden ist. Die Übergabe der generierten Radar-Abschnitte findet unter Verwendung der oben erwähnten Warteschlange statt.

4.2 | Vergleich zu KESO ohne GC-Schutz

Um zu bestimmen wie sich die in dieser Arbeit implementieren Schutzmechanismen für die GCs auf die Größe der Anwendungsdatei und die Ausführungszeit auswirken, wurde die *on-the-go*-Variante des CD_x-Benchmarks herangezogen. Dieser wurde auf Basis des CiAO-Betriebssystems [LHSP+09] für die TriCore-Architektur compiliert. Das KESO-System wurde dabei mit einer Heap-Größe von 256 KiB für den Coffee-Break(CB) und 512 KiB für den Idle-Round-Robin(IRR) konfiguriert. Die Anzahl der zu berechnenden Radar-Abschnitte wurde auf 50 gesetzt. Die jeweiligen Versionen der verwendeten Komponenten ist aus Tabelle 4.1 ersichtlich.

Die in diesem Abschnitt folgenden Tabellen und Abbildungen zeigen die ermittelten Messwerte bei verschiedenen Konfigurationen. Als Basis wurde stets ein KESO-System ohne GC-Schutz gewählt. Für jede zu schützende Datenstruktur wurde eine weitere Messung durchgeführt, bei der nur die jeweils zugehörige Option in der GC-Konfiguration aktiviert war. Abrundend wurden zusätzlich noch die Werte für eine Konfiguration mit allen aktivierten Optionen berechnet.

Um Platz zu sparen wurden für die Legenden und Beschriftungen Abkürzungen verwendet. *Vanilla* bedeutet in diesem Zusammenhang, dass sämtliche Schutz-Mechanismen von KESO deaktiviert sind. *H+DRC* stellt die Kombination aus Header-Only-Check[1] und Dereference-Checks[2] dar. Ferner steht *FSP* für *Freispeicherliste*, *LLRefs* für die *Liste lokaler Referenzen*, und *Color* für das vom IRR verwendete Color-Bit.

[1] JINO-Optionen: use_coded_refs_check_arraylength, use_coded_refs_header_check
[2] JINO-Optionen: coded_ref_cxxp_parity, llrefs_no_volatile

CB	H+DRC	FSP	Stack	Bitmap	LLRefs	Color	Alle
Text:	75.177	75.705	75.881	76.177	75.959	-	76.599
OH:	0,00	0,70	0,94	1,33	1,04	-	1,89
Daten:	1.461	1.461	1.461	1.461	1.461	-	1.461
OH:	0,00	0,00	0,00	0,00	0,00	-	0,00
BSS:	309.298	309.306	309.310	309.314	309.306	-	309.310
OH:	0,00	0,00	0,00	0,01	0,00	-	0,00
IRR	H+DRC	FSP	Stack	Bitmap	LLRefs	Color	Alle
Text:	80.009	81.677	80.465	80.553	80.479	80.293	82.555
OH:	0,00	2,08	0,57	0,68	0,59	0,35	3,18
Daten:	1.462	1.465	1.465	1.465	1.465	1.465	1.465
OH:	0,00	0,21	0,21	0,21	0,21	0,21	0,21
BSS:	309.306	309.314	309.318	309.322	309.314	309.318	309.322
OH:	0,00	0,00	0,00	0,01	0,00	0,00	0,01

Tabelle 4.2: Größe der Segmente der Binärdatei in Byte und Overhead(OH) in Prozent zur H+DRC-Variante in Prozent bei verschiedenen Konfigurationen.

4.2.1 | Anwendungsgröße

Zunächst soll analysiert werden, wie sich die Erweiterungen dieser Arbeit auf die Größe der Segmente der erzeugten Binärdateien auswirkt. Für den Vergleich wurde die Applikation mit folgenden, zusätzlich zu den in den Konfigurationen hinterlegten, JINO-Optionen übersetzt:
ssa, const_arg_prop, ssa_alias_prop, use_coded_refs_inline, ft_gc_use_parity, production

Tabelle 4.2 zeigt die Größe der Segmente der erzeugten Dateien bei verschiedenen Konfigurationen. Zusätzlich wird deren Overhead im Vergleich zu einem mit H+DRC kompilierten System dargestellt.

Coffee-Break

Die in Abschnitt 3.2 beschriebene feingranulare Konfiguration der Fehlertoleranzstufe pro Domäne benötigt zusätzliche Felder in den Domänen-Structs. Pro

verfügbarer Option wird 1 Byte zur Speicherung der Stufe im BSS-Segment verwendet. Zusätzlich wird eine für die jeweilige Domäne festgelegte Gesamt-Stufe abgelegt. Für den Coffee-Break ergibt sich aufgrund des 4-Byte-Alignments daher ein zusätzlicher Speicherbedarf pro Domäne von $6 * 1 = 5 \Rightarrow 8$ Byte. Um die zur Absicherung verwendeten Paritäten zu speichern, sind unter Umständen weitere Datenstrukturen notwendig. Stack- und Bitmap-Absicherung teilen sich hierbei ein 1 Byte großes Feld, welches das BSS-Segment aufgrund des Alignments um 4 Byte vergrößert. Die Bitmap benötigt zur Zählung der bereits markierten Slots eine zusätzliche 4 Byte große Variable. Durch geschickte Platzierung des Parität-Bytes im Struct der Tasks, fällt für die LLRef-Absicherung kein zusätzlicher Speicherbedarf an. Die Freispeicherliste benötigt keine zusätzlichen Felder, weswegen hier ebenfalls nur der Zuwachs durch die Domänen-Konfiguration zu verzeichnen ist. Der Overhead des BSS-Segment ist insgesamt sehr gering und kann daher vernachlässigt werden.

Tabelle 4.2 beinhaltet die gemessenen absoluten Zahlen, sowie den Overhead in Prozent, während Abbildung 4.1 den Zuwachs des Text-Segments als Balkendiagramm darstellt.

Im Daten-Segment ist keine Änderung festzustellen. Das Text-Segment wächst je nach Konfiguration gegenüber der H+DRC-Variante an. Der Zuwachs beträgt zwischen 0,7 und 1,89 Prozent. Die Absicherung der Bitmap besitzt dabei mit 1,33 Prozent die größten Anteil.

Idle-Round-Robin

Wie in Tabelle 4.2 zu erkennen ist, verhält sich die Größe des BSS-Segments analog zu der des Coffee-Break. Die zusätzliche Möglichkeit zur Absicherung des Color-Bits resultiert dabei im gleichen Overhead wie die Sicherung des Stacks, da zur Speicherung der Parität dasselbe Byte verwendet wird. Das Daten-Segment zeigt wie beim Coffee-Break keine Veränderung und ist bei allen Konfigurationen mit GC-Fehlererkennung durchgehend konstant. Deren Größe unterscheidet sich von der des Daten-Segments der H+DRC-Variante um 2 Byte.

Beim Text-Segment ist gegenüber dem Coffee-Break ein deutlicherer Zuwachs von 2,08 Prozent bei der Absicherung der Freispeicherliste erkennbar (Abbildung 4.2). Aufgrund der Unterbrechbarkeit des Idle-Round-Robin ist dessen Code zur Verwaltung der Liste wesentlich komplexer. Es wird dabei öfter sowohl lesen als auch schreibend auf die Next-Zeiger und die Größe der Listen-Elemente zugegriffen. Zusätzlich wird das Flags-Feld ebenfalls oft gelesen und geschrieben. An all diesen Stellen im Code, werden die Funktionen zum Berechnen, Überprüfen und Neusetzen der Paritäten vom Übersetzer an Ort und Stelle eingebunden (Inlining). Dies führt zu einer Vergrößerung des Programm-Codes.

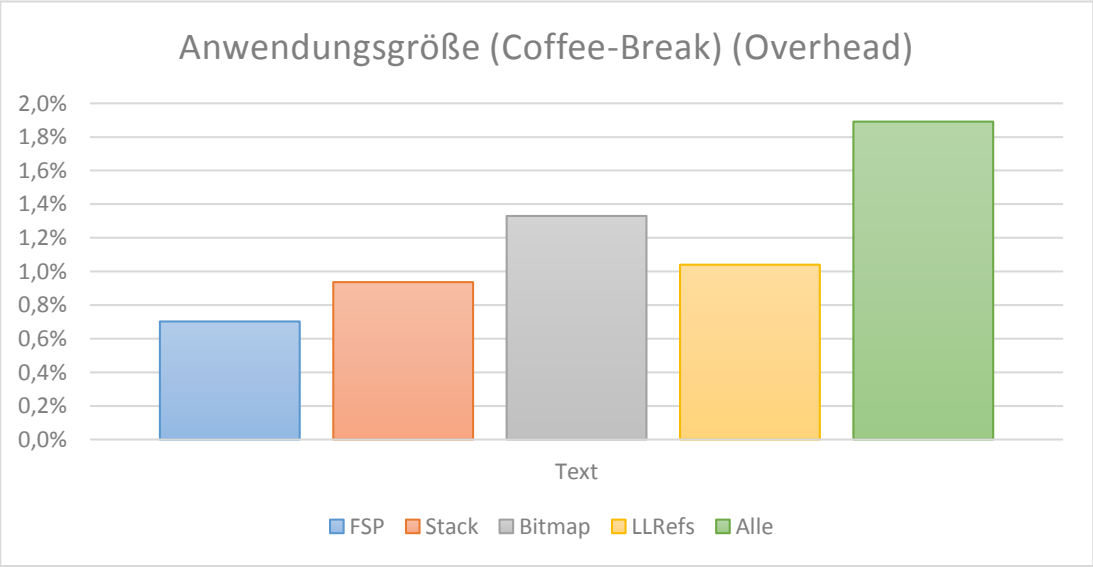


Abbildung 4.1: Coffee-Break: Vergrößerung der Binärdatei (Overhead) im Vergleich zur H+DRC-Variante in Prozent bei verschiedenen Konfigurationen. Die Absicherung der Bitmap stellt mit 1,33 Prozent den höchsten Overhead dar.

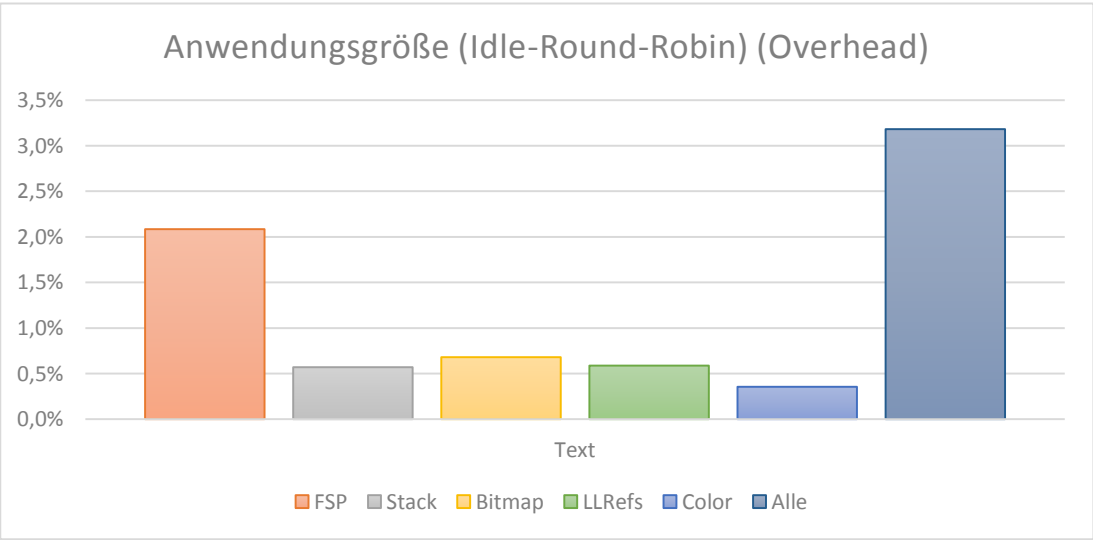


Abbildung 4.2: Idle-Round-Robin: Vergrößerung der Binärdatei (Overhead) im Vergleich zur H+DRC-Variante in Prozent bei verschiedenen Konfigurationen. Erkennbar ist ein deutlicher Zuwachs um 2,08 Prozent bei Absicherung der Freispeicherliste.

Fazit

Insgesamt führen die im Rahmen dieser Arbeit implementierten Mechanismen zur Absicherung der Datenstrukturen bei allen in KESO verfügbaren Speicherverwaltungen zu einem akzeptablen Zuwachs der Anwendungsgröße von maximal 1,89 Prozent (Coffee-Break) bzw. 3,18 Prozent (Idle-Round-Robin).

4.2.2 | Ausführungszeit

Die "on-the-go"-Konfiguration wurde auf einem TriCore TC1796 Mikrocontroller auf Basis des CiAO-Betriebssystems evaluiert. Für die "simulated"-Version existiert keine Konfiguration für die TriCore-Architektur, da die Anwendung zu viel Heap-Speicher benötigt, die das verwendete TriCore-Board nicht zur Verfügung stellen kann. Die Hardware- und Software-Konfiguration ist in Tabelle 4.1 aufgeführt.

Die im Kapitel 2 beschriebenen Scan- und Markierungs-Phasen wurden bei den Messungen zusammengelegt, da diese im Programm-Code stark verzahnt sind. Eine strikte Trennung bei der Messung wurde daher im Rahmen dieser Arbeit nicht durchgeführt.

Coffee-Break

Die bei den Laufzeitmessungen des Coffee-Break ermittelten Ergebnisse sind in Tabelle 4.3 gelistet. Der Overhead(OH) gegenüber der H+DRC-Variante ist in Abbildung 4.3 dargestellt.

Es ist zu erkennen, dass der Schutz der Freispeicherliste mit 97,33 Prozent Overhead, die Laufzeit der Allokationen von 0,011 auf gerundet 0,022 Millisekunden im Schnitt fast verdoppelt. Die Reduzierung der Laufzeit bei den übrigen Konfigurationen ist vermutlich auf Optimierungen des verwendeten GCC-Übersetzers zurückzuführen. Die Schwankungen bewegen sich dabei im einstelligen Mikrosekunden-Bereich.

Bei der Scan- und Markierungs-Phase erzeugt die Absicherung des Working-Stack den größten Overhead von 41,99 Prozent. Sowohl beim Scannen nach lebendigen Objekten als auch bei der Markierung dieser in der Bitmap wird der Working-Stack intensiv verwendet. Bei Aktivierung aller Absicherungen beläuft sich der Overhead auf 60,55 Prozent, was einer Steigerung von 1,521 auf 2,442 Millisekunden entspricht.

Bei der Sweep-Phase ist erneut eine Reduzierung der Laufzeit gegenüber der H+DRC-Variante erkennbar, die vermutlich durch Übersetzer-Optimierungen verursacht werden.

CB	H+DRC	FSP	Stack	Bitmap	LLRefs	Color	Alle
alloc:	0,011	0,022	0,009	0,011	0,010	-	0,019
OH:	0,00	97,33	-19,17	-5,76	-13,87	-	67,70
scan + mark:	1,521	1,633	2,160	1,670	1,666	-	2,442
OH:	0,00	7,36	41,99	9,80	9,50	-	60,55
sweep:	0,645	0,613	0,585	0,571	0,577	-	0,622
OH:	0,00	-5,05	-9,43	-11,57	-10,61	-	-3,65
IRR	H+DRC	FSP	Stack	Bitmap	LLRefs	Color	Alle
alloc:	0,021	0,056	0,022	0,020	0,022	0,022	0,056
OH:	0,00	170,04	5,00	-1,87	5,32	7,59	173,40
scan + mark:	1,834	2,087	2,234	1,901	1,991	2,183	2,644
OH:	0,00	13,78	21,77	3,62	8,57	19,01	44,16
sweep:	50,874	41,552	38,950	50,741	44,832	17,621	48,827
OH:	0,00	-18,32	-23,44	-0,26	-11,88	-65,36	-4,02

Tabelle 4.3: Laufzeit in Millisekunden und Overhead(OH) zur H+DRC-Variante in Prozent bei verschiedenen Konfigurationen.

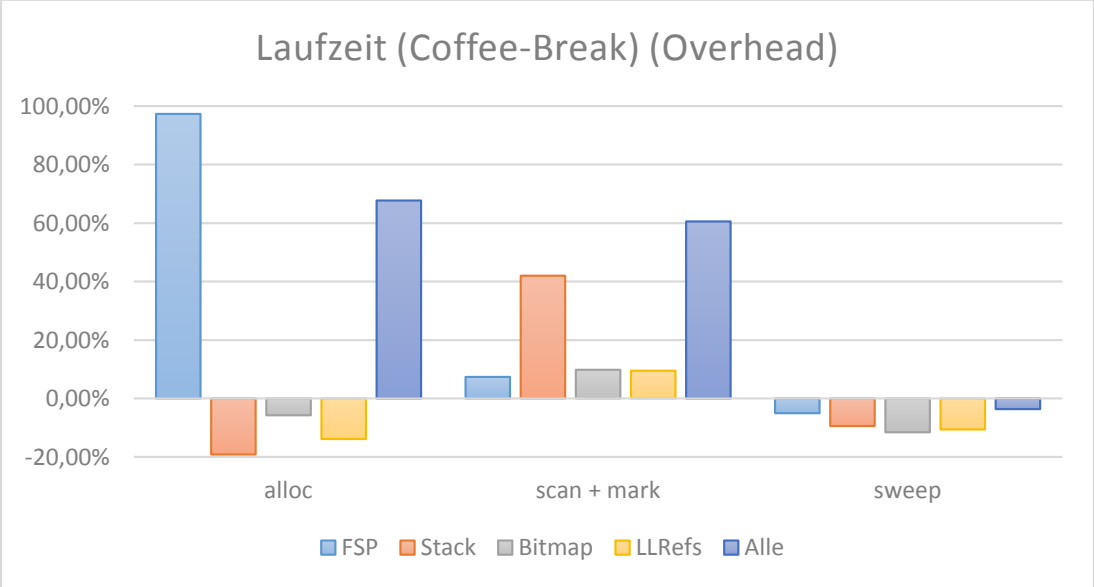


Abbildung 4.3: Coffee-Break: Erhöhung der Laufzeit (Overhead) in Prozent bei verschiedenen Konfigurationen. Die Absicherung der Freispeicherliste erzeugt bei den Allokationen einen Overhead von 97,33 Prozent. Bei allen aktivierten Sicherungen, beträgt der Overhead in der Scan- und Markierungs-Phase 60,55 Prozent.

Idle-Round-Robin

Die bei den Laufzeitmessungen des Idle-Round-Robin ermittelten Ergebnisse sind in Tabelle 4.3 gelistet. Der Overhead(OH) gegenüber der H+DRC-Variante ist in Abbildung 4.4 dargestellt.

Die Laufzeit der Allokationen steigt bei Absicherung er Freispeicherliste im Schnitt von 0,021 auf 0,056 Millisekunden an, was einem Overhead von 170,04 Prozent entspricht. Im Vergleich zum Coffee-Break wirkt sich die Absicherung stärker auf die Laufzeit aus, da aufgrund der höheren Komplexität der Verwaltung der Liste durch die Synchronisation mit dem System, mehr Überprüfungen der Integrität der Next-Zeiger und Elementgrößen notwendig sind.

Die Scan- und Markierungs-Phase verhält sich ähnlich zu der des Coffee-Break. Die Absicherung des Working-Stacks erzeugt hierbei den größten Overhead von 21,77 Prozent bei einer Steigerung von 1,834 auf 2,234 Millisekunden. Hinzu kommt die Absicherung des Color-Bit, die eine Steigerung von 19,01 Prozent von 1,834 auf 2,183 Millisekunden bewirkt. Insgesamt ist bei allen aktivierten Schutz-Maßnahmen eine Steigerung von 44,16 Prozent auf 2,644 Millisekunden zu verzeichnen.

Ebenfalls ist bei der Sweep-Phase eine Reduzierung der Laufzeit zu beobachten, die durch Optimierungen des C-Übersetzers verursacht werden.

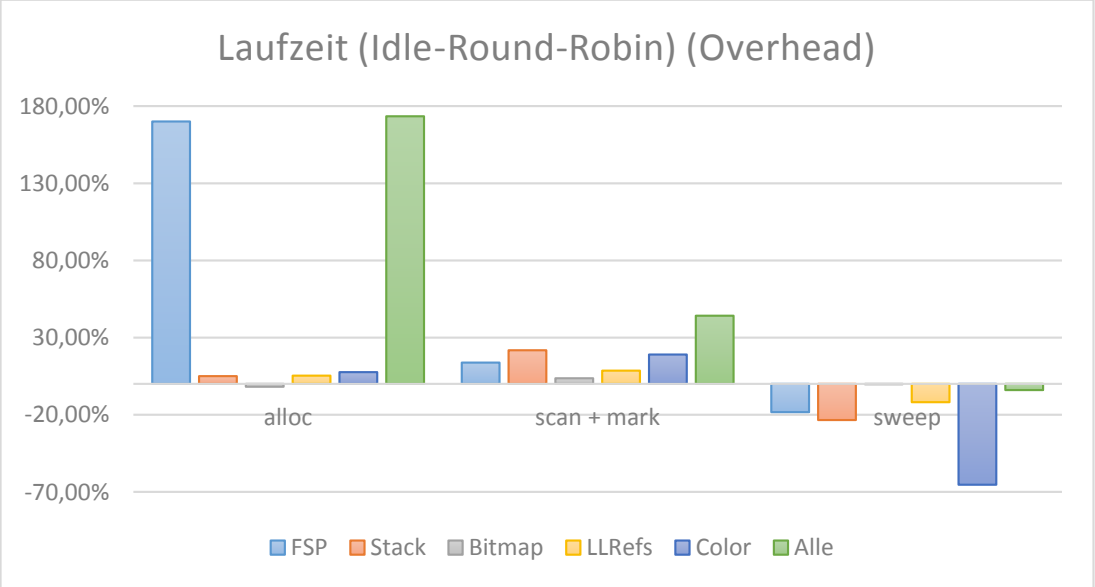


Abbildung 4.4: Idle-Round-Robin: Erhöhung der Laufzeit (Overhead) in Prozent bei verschiedenen Konfigurationen. Die Absicherung der Freispeicherliste erzeugt bei den Allokationen einen Overhead von 170,04 Prozent. Bei allen aktivierten Sicherungen, beträgt der Overhead in der Scan- und Markierungs-Phase 44,16 Prozent

Fazit

Der durch die Absicherung erzeugte Overhead liegt bei den Allokationen im Schnitt zwischen 100 und 175 Prozent, bei der Scan- und Markierungs-Phase nur zwischen 45 und 60 Prozent. Die Sweep-Phase zeigt aufgrund von Optimierungen des GCC-Übersetzers sogar eine Reduzierung um bis zu 12 Prozent auf.

Insgesamt bewegt sich die Erhöhung der Laufzeit in einem durchaus akzeptablen Bereich. Aufgrund der feingranular konfigurierbaren Absicherungen kann es daher sinnvoll sein, je nach Einsatzzweck gezielt Datenstrukturen abzusichern.

4.3 | Fehlerinjektion

Um die implementierten Schutzmechanismen zur Fehlererkennung auf Funktionalität zu testen, wurden im Rahmen der Evaluation dieser Arbeit eine Reihe an Fehlerinjektions-Experimenten durchgeführt. Als Testapplikation kam ebenfalls der Benchmark CD_x zum Einsatz. In diesem Abschnitt wird zunächst die hierfür verwendete Fehlerinjektions-Software vorgestellt. Nach einer kurzen Einführung in die verwendeten Konfigurationen, werden die Ergebnisse tabellarisch und graphisch vorgestellt und abschließend diskutiert.

4.3.1 | Die FAIL*-Fehlerinjektionsumgebung

Als Basis diente die FAIL*-Fehlerinjektionsumgebung [SHK+12], die auf dem Open-Source-Simulator Bochs [Law96] für die x86-Architektur aufsetzt. Dabei wurde zunächst ein sogenannter *Golden Run* durchgeführt, in welchem ein Tracing-Programm gestartet wird, das alle Instruktionen, die einen Speicherzugriff verursachen, und deren Aufrufe protokolliert. Um die Menge auf die wesentlichen relevanten Abschnitte im Speicher zu reduzieren, besteht die Möglichkeit die in Frage kommenden Instruktionen anhand einer Liste von Speicherabschnitten (memory map) zu beschränken. Des Weiteren führt FAIL eine intelligente Säuberung (Pruning) der ermittelten Instruktions- uns Speicheradressmenge durch. Dabei werden diejenigen Instruktionen entfernt, die schreibend auf den Speicher zugreifen. Außerdem werden alle Speicheradressen der in der Liste definierten Speicherabschnitte entfernt, auf die nie zugegriffen wird. Injektionen in diese Speicheradressen sind sinnlos, da sie keine Auswirkung auf den Lauf des Programms haben. Letztendlich wird durch diese Reduzierung die Anzahl der Injektions-Experimente nicht unnötig vergrößert.

Für die Experimente dieser Arbeit wurden zwei Einschränkungen vorgenommen. Zum einen wurde die Liste der Speicherbereiche auf die für den GC relevanten Abschnitte beschränkt. Diese sind zum einen der Heap selbst, welcher die Freispeicherliste beinhaltet, und zum anderen alle weiteren Speicherstrukturen wie der Working-Stack, die Bitmap, die LLRefs und das Color-Bit. Zusätzlich wurde die Menge weiter auf die Instruktions-Adressen reduziert, welche innerhalb der vom GC verwendeten Funktionen liegen.

Für jeden Instruktion-Aufruf wurden pro gelesener Speicheradresse nacheinander 8 einzelne Bitkipper injiziert. Dafür wird das simulierte System zunächst pausiert. Nach erfolgter Injektion des Bitkippers an der gewünschten Bit-Position der Speicheradresse, wird die Ausführung des Systems fortgesetzt. Das FAIL-Experiment kann anschließend auf etwaige Fehler reagieren. Hierzu gehören geworfene Ausnahmen (Exceptions), Traps und Zugriffe auf ungültige Speicheradressen. Um eine endlose Ausführung der Anwendung zu verhindern, bricht das Experiment zusätzlich nach einer vorher definierten maximalen Zeitspanne (hier 5 Sekunden) ab.

4.3.2 | Experimente

Es wurden diverse Experimente auf Basis unterschiedlicher Konfigurationen durchgeführt. Die Größe des Heaps betrug bei allen Durchführungen 48 KiB. Die Slot-Größe war auf 16 Bytes festgelegt. Der Benchmark CD_x wurde mit einer Radar-Abschnitt-Anzahl von 2 übersetzt. Diese Wahl der Konfigurationsparameter resultierte in 2305 Allokationen, wobei insgesamt 50,768 KiB (CB) bzw. 50,848 KiB (IRR) auf dem Heap vergeben wurden. Die GC-Phase wurde dabei einmal

durchlaufen, wobei 41,527 KiB (CB) bzw. 44,480 KiB (IRR) Speicher freigegeben werden konnte. Um sinnvolle Werte zu ermitteln, kam eine eigens für diese Arbeit implementierte Statistik zum Einsatz, die während der Laufzeit Daten über die in den Domänen verwendeten GCs sammelt und diese kumuliert kurz vor Beendigung des Betriebssystems auf der Standardausgabe wiedergibt. Hierfür wurde der von der OSEK-Spezifikation zur Verfügung gestellte *ShutdownHook* verwendet.

Als Referenz und Ausgangssituation wurde ein Experiment ohne aktivierte Fehlertoleranzmechanismen (Vanilla), sowie eines mit aktiviertem H+DRC-Schutz durchgeführt. Da der in dieser Arbeit implementierte GC die Möglichkeit bietet, die Stufe der Fehlererkennung feingranular nach verwendeten Datenstrukturen zu konfigurieren, wurde pro verfügbarer Option ein weiteres Experiment durchgeführt. Abrundend wurde eine Konfiguration erstellt, in der alle Schutzmechanismen aktiviert waren. Bei allen verwendeten Konfigurationen mit GC-Schutz war H+DRC ebenfalls aktiviert, um den Gewinn an Fehlererkennung gegenüber bereits vorhandenen Schutzmechanismen ermitteln zu können.

Insgesamt wurden 3.567.048 Experimente durchgeführt.

4.3.3 | Fehlerraten

Die Ergebnisse der im letzten Abschnitt beschriebenen Experimente sind in den folgenden Tabellen 4.4 und 4.5, sowie in den Abbildungen 4.5 und 4.7 dargestellt. Zu sehen sind die Ergebnisse der Anwendungsläufe gruppiert nach Konfiguration. *Speicherzugriff* steht hier für einen ungültigen Zugriff ins Text-Segment oder auf eine Adresse außerhalb des von der Anwendung verwendeten Adressraums. Die Abkürzung *E.* steht jeweils für Exception (Ausnahme). Der Begriff *GC-E.* bezeichnet einen im Rahmen dieser Arbeit neu eingeführten Exception-Typ, die GC-Exception. Dabei handelt es sich um Ausnahmen, die durch Fehler verursacht werden, die der GC erkannt hat. Parität-Exceptions werden dagegen von den H+DRC-Mechanismen geworfen.

Die Ergebniszahlen ergeben sich aus folgender Formel:

$$N(e) = \sum_{k=1}^{Max(I(e))} Time_R(I(e,k)) - Time_W(I(e,k)) \tag{4.1}$$

Die Anzahl(N) der Ergebnisse des Typs e ergibt sich aus der Summe der relevanten Zeitspannen aller Injektionen mit dem Ergebnis e. Eine Zeitspanne definiert sich durch den jeweils letzten schreibenden Zeitpunkt($Time_W$) auf die zur Injektion gehörenden Speicheradresse und dem Lesezeitpunkt der Injektion ($Time_R$).

Unter Berücksichtigung, dass die Wahrscheinlichkeit für einen Bitkipper größer wird, je länger die Daten an der jeweiligen Speicheradresse nicht geschrieben wurden, folgt, dass die Anzahl der Ergebnisse einer Injektion der Länge des Zeitintervalls entspricht.

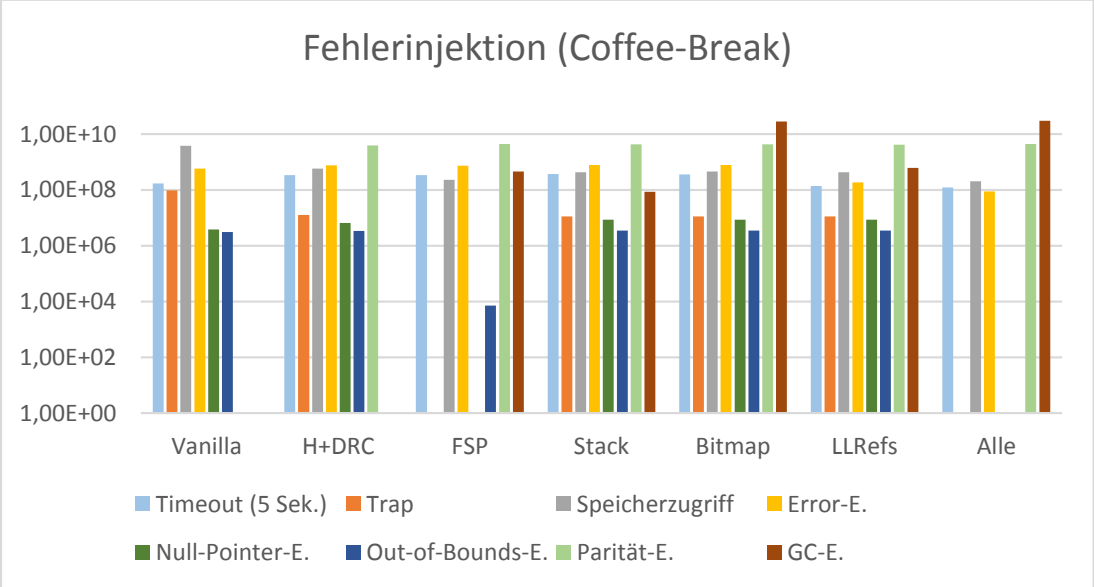


Abbildung 4.5: Coffee-Break: Ergebnisse der Fehlerinjektion bei verschiedenen Konfigurationen. Logarithmische Y-Achse.

Coffee-Break

Wie aus Tabelle 4.4 und Abbildung 4.5 ersichtlich, kann bei Aktivierung aller Schutzmechanismen dieser Arbeit die Anzahl der Timeouts (von 3,36E+08 auf 1,23E+08), Error-Exceptions (von 7,62E+08 auf 8,76E+07) und Zugriffe auf ungültige Speicheradressen (von 5,73E+08 auf 2,01E+08) gegenüber der H+DRC-Variante reduziert werden. Traps, Null-Pointer und Out-of-Bounds-Exceptions werden sogar gänzlich eliminiert.

Die Absicherung der Freispeicherliste (FSP) trägt im Vergleich zu den Absicherungen des Stacks, der Bitmap und den LLrefs am meisten zu einer Verringerung der ungültigen Speicherzugriffe bei. Sie verhindert außerdem alle Traps und Null-Pointer-Exceptions.

Die prozentuale Verteilung, der durch den GC erkannten Fehler, ist gruppiert nach Datenstruktur in Abbildung 4.6 dargestellt. Hier wird sichtbar, dass die meisten Fehler (96,03 Prozent) bei der Bitmap auftreten können. Dies liegt zum einen an der großen Angriffsfläche (1 Byte pro Slot) und zum anderen daran, dass die Bitmap erst gegen Ende dieses Experiments verwendet wird. Ihre Speicheradressen werden daher lange nicht gelesen, was die Wahrscheinlichkeit von Bitkippern erhöht und damit laut Gleichung (4.1) die Anzahl der Ergebnisse.

Ergebnis	Vanilla	H+DRC	V+FSP	FSP
Timeout (5 Sek.):	1,72E+08	3,36E+08	1,72E+08	3,42E+08
Trap:	9,48E+07	1,26E+07	9,30E+07	0
Speicherzugriff:	3,79E+09	5,73E+08	4,53E+09	2,28E+08
Error-E.:	5,87E+08	7,62E+08	6,00E+08	7,37E+08
Null-Pointer-E.:	3,82E+06	6,42E+06	0	0
Out-of-Bounds-E.:	3,06E+06	3,41E+06	2,17E+07	7,12E+03
Parität-E.:	0	3,96E+09	0	4,42E+09
GC-E.:	0	0	3,40E+08	4,61E+08
Gesamt:	4,65E+09	5,65E+09	5,76E+09	6,18E+09
Ergebnis	Stack	Bitmap	LLRefs	Alle
Timeout (5 Sek.):	3,65E+08	3,54E+08	1,39E+08	1,23E+08
Trap:	1,12E+07	1,11E+07	1,11E+07	0
Speicherzugriff:	4,28E+08	4,55E+08	4,26E+08	2,01E+08
Error-E.:	7,87E+08	7,74E+08	1,88E+08	8,76E+07
Null-Pointer-E.:	8,44E+06	8,50E+06	8,50E+06	0
Out-of-Bounds-E.:	3,52E+06	3,52E+06	3,52E+06	0
Parität-E.:	4,29E+09	4,32E+09	4,21E+09	4,39E+09
GC-E.:	8,53E+07	2,82E+10	6,17E+08	2,98E+10
Gesamt:	5,98E+09	3,41E+10	5,60E+09	3,46E+10

Tabelle 4.4: Coffee-Break: Ergebnisse der Fehlerinjektion bei verschiedenen Konfigurationen.

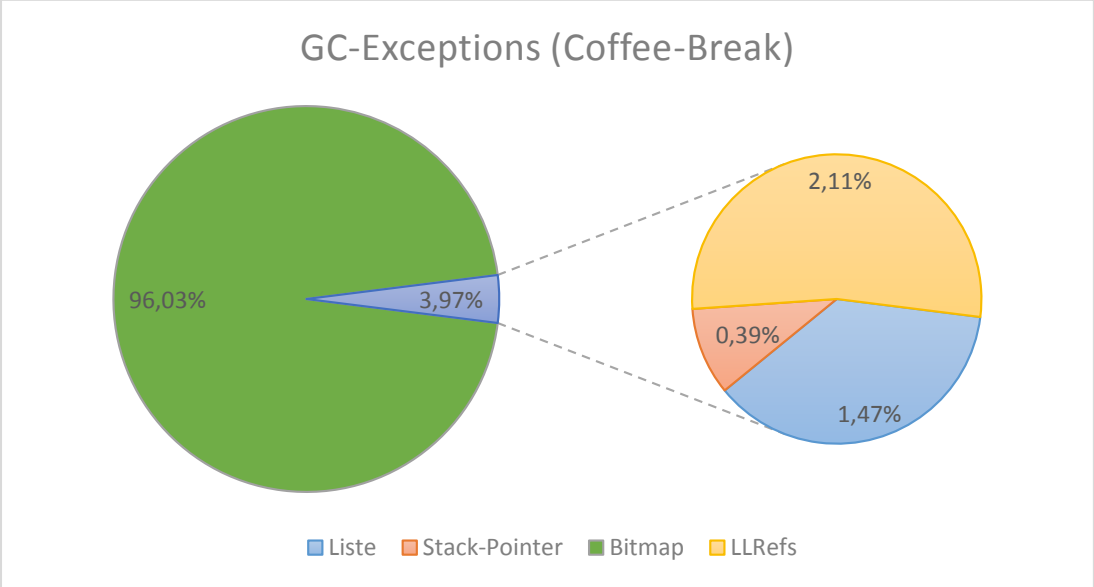


Abbildung 4.6: Coffee-Break: Verteilung der GC-Exceptions bei Aktivierung alle GC-Schutzmechanismen.

Idle-Round-Robin

Tabelle 4.5 und Abbildung 4.7 zeigen, dass die Aktivierung aller Schutzmechanismen dieser Arbeit im Vergleich zur H+DRC-Variante alle Timeouts und Traps verhindern kann. Außerdem wird die Anzahl der Speicherzugriffe (von 8,77E+08 auf 6,46E+07) reduziert. Die Anzahl der Error-Exceptions verringert sich von 1,04E+09 auf 2,41E+08. Auch die Out-of-Bounds-Exceptions können von 4,99E+07 auf 4,20E+07 gesenkt werden.

Die Absicherung der Freispeicherliste (FSP) trägt dabei erneut wie bereits beim Coffee-Break im Vergleich zu den Absicherungen des Stacks, der Bitmap und den LLrefs am meisten zu einer Verringerung der ungültigen Speicherzugriffe bei. Sie reduziert außerdem erheblich die Anzahl der Timeouts (von 3,19E+07 auf 7,10E+01) und Traps (von 2,04E+07 auf 4,09E+03).

Die prozentuale Verteilung der GC-Exceptions ist gruppiert nach Datenstruktur in Abbildung 4.8 dargestellt. Wie zuvor beim Coffee-Break wird auch hier sichtbar, dass die meisten Fehler (92,56 Prozent) bei der Bitmap auftreten können. Die Freispeicherliste besitzt mit 3,05 Prozent einen höheren Anteil als beim Coffee-Break (1,47 Prozent). Dies liegt an der höheren Anzahl an Zugriffen auf die Datenstruktur aufgrund der größerem Komplexität durch die Synchronisierung mit dem System.

Ergebnis	Vanilla	H+DRC	V+FSP	FSP	-
Timeout (5 Sek.)	2,31E+07	3,19E+07	0	7,10E+01	-
Trap	1,07E+08	2,04E+07	8,94E+07	4085	-
Speicherzugriff	3,91E+08	8,77E+08	7,14E+08	3,86E+08	-
Error-E.	7,12E+08	1,04E+09	8,42E+08	1,18E+09	-
Null-Pointer-E.	1,10E+07	6,26E+07	49030725	63213091	-
Out-of-Bounds-E.	1,01E+07	4,99E+07	4,89E+07	4,21E+07	-
Parität-E.	0	6,07E+08	0	6,28E+08	-
GC-E.	0	0	8,86E+08	1,17E+09	-
Gesamt	1,25E+09	2,69E+09	2,63E+09	3,47E+09	-
Ergebnis	Stack	Bitmap	LLRefs	Color	Alle
Timeout (5 Sek.)	1,49E+07	1,40E+07	1,49E+07	4,04E+07	0
Trap	2,99E+07	2,99E+07	3,00E+07	2,99E+07	0
Speicherzugriff	7,04E+08	7,84E+08	8,17E+08	5,17E+08	6,46E+07
Error-E.	1,04E+09	1,08E+09	2,16E+08	1,03E+09	2,41E+08
Null-Pointer-E.	6,26E+07	6,25E+07	6,26E+07	6,26E+07	6,31E+07
Out-of-Bounds-E.	5,01E+07	5,00E+07	5,01E+07	5,02E+07	4,20E+07
Parität-E.	5,53E+08	6,58E+08	6,44E+08	4,82E+08	4,27E+08
GC-E.	3,18E+08	2,64E+10	6,12E+08	6,33E+08	3,21E+10
Gesamt	2,77E+09	2,91E+10	2,45E+09	2,85E+09	3,29E+10

Tabelle 4.5: Idle-Round-Robin: Ergebnisse der Fehlerinjektion bei verschiedenen Konfigurationen.

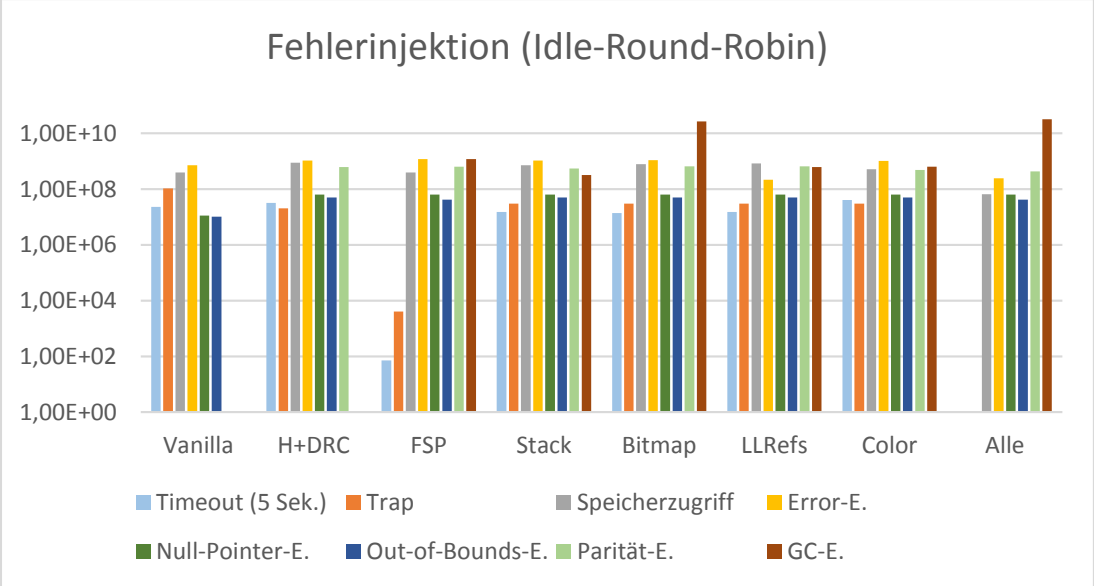


Abbildung 4.7: Idle-Round-Robin: Ergebnisse der Fehlerinjektion bei verschiedenen Konfigurationen. Logarithmische Y-Achse.

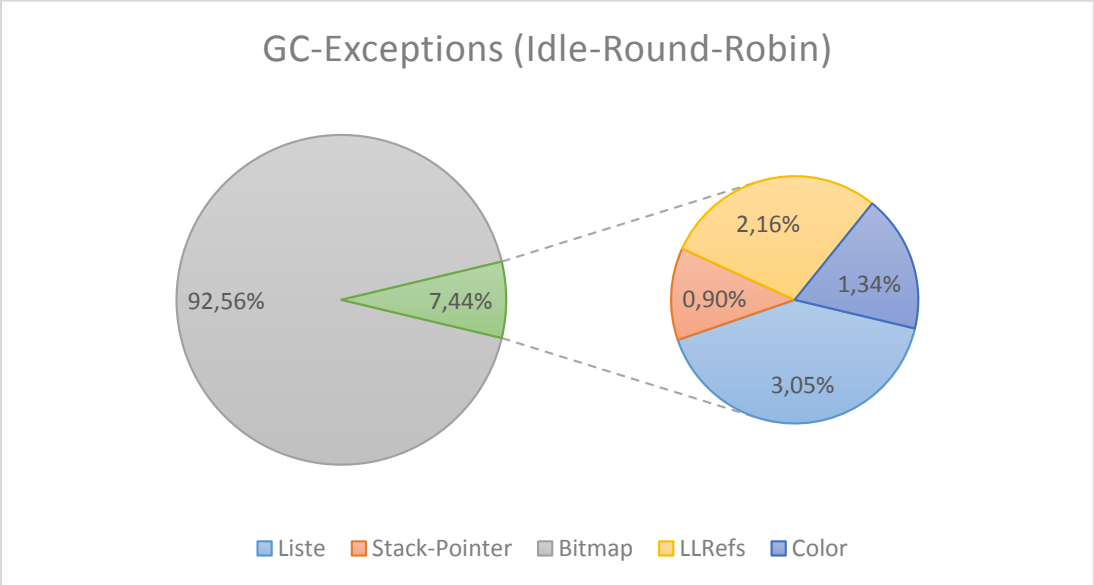


Abbildung 4.8: Idle-Round-Robin: Verteilung der GC-Exceptions bei Aktivierung alle GC-Schutzmechanismen.

4.4 | Zusammenfassung

Die Evaluation hat gezeigt, dass die im Rahmen dieser Arbeit entworfenen und implementierten Schutzmechanismen für die Datenstrukturen der beiden GC-Varianten in KESO, effektiv sind. Viele der potentiellen Bitkipper können durch die verschiedenen Absicherungen erkannt werden, wodurch das System gezielt darauf reagieren kann. Das Auftreten von undefinierbarem Verhalten der Anwendung und Verletzung der Typsicherheit der Laufzeitumgebung kann damit reduziert werden.

Die verursachte Erhöhung der Laufzeit bei den Allokationen beträgt im Schnitt 67,70 Prozent (Coffee-Break) bzw. 173,40 Prozent (Idle-Round-Robin). Der Overhead der Scan- und Markierungs-Phase liegt zwischen 45 Prozent (Idle-Round-Robin) und 68 Prozent (Coffee-Break). Je nach Einsatzzweck kann diese Erhöhung zugunsten des Gewinns an Zuverlässigkeit des Systems durch die Fehlererkennung durchaus vernachlässigbar sein.

Die Vergrößerung der Programmdateien beträgt je nach Konfiguration der Fehlererkennung maximal 1,89 Prozent (Coffee-Break) bzw. 3,18 Prozent (Idle-Round-Robin). Dies stellt einen akzeptablen Zuwachs dar.

5 | Abschluss

Im Rahmen dieser Arbeit wurde das Design und die Implementierung einer fehlertoleranten Speicherbereinigung vorgestellt, evaluiert und diskutiert. Dazu wurden drei Speicherverwaltungs-Varianten der KESO Multi-JVM, einer virtuellen Maschine für Java, die für statisch konfigurierte eingebettete Systeme konzipiert ist, analysiert und erweitert.

Die jeweils verwendeten Datenstrukturen der Restricted-Domain-Scope-, Coffee-Break- und Idle-Round-Robin-Variante wurden hinsichtlich ihrer Anfälligkeit für transiente Fehler (Bitkipper) untersucht. Anhand der Verwendung von Paritäten wurde gezeigt, wie diese Datenstrukturen effektiv und ressourcensparend abgesichert werden können.

Ziel war es, eine Fehlererkennung zu entwerfen, die vom Programmierer feingranular pro Domäne konfigurierbar ist, um die Anwendung je nach Zielsystem und Einsatzzweck flexibel anpassen zu können. Dadurch ist es möglich gezielt ausgewählte Datenstrukturen abzusichern.

In der abschließenden Evaluation wurde anhand von Laufzeitmessungen gezeigt, dass der durch die Absicherungen entstandene Overhead in einem durchaus vertretbaren Bereich liegt. Es konnte durch Vergleich der Dateigrößen des Programms gezeigt werden, dass JINO aufgrund der feinen Konfiguration den generierten C-Code effizient reduzieren kann und die Programmgröße durch die Absicherungsmaßnahmen nicht signifikant ansteigt.

Ferne wurde in diversen Fehlerinjektions-Experimenten gezeigt, dass die implementierten Schutzmechanismen effektiv sind und eine Vielzahl von Fehlerszenarien reduzieren oder gar verhindern können.

5.1 | Ausblick

In dieser Arbeit wurde die Evaluation auf die "on-the-go"-Variante des Benchmarks CD_x beschränkt. Interessant wären durchaus auch Laufzeitmessungen und Fehlerinjektions-Ergebnisse bei Verwendung einer Applikation mit mehreren Domänen (bsp. "simulated-multidomain").

Hierzu müsste zusätzlich die JINO-Klasse *IMLocalPortalMethod* untersucht werden, um verkettete Task-Stacks über Domänengrenzen hinweg zur Auffindung lokaler Objektreferenzen (LLRefs) abzusichern.

KESO bietet neben der Bitmap noch zwei weitere Datenstrukturen, die stattdessen zur Markierung von lebendigen Objekten verwendet werden können. Dabei handelt es sich um die *Patchmap* und die *Testmap*, die beide ebenfalls abgesichert werden könnten.

Des Weiteren besteht bei der Erkennung von Bitkippern durchaus noch Potential um beispielsweise die Anzahl der fehlerhaften Speicherzugriffe weiter reduzieren zu können.

Abbildungsverzeichnis

Auflistungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AUT11] AUTOSAR: Specification of Operating System (Version 5.0.0) / Automotive Open System Architecture GbR. 2011. – Forschungsbericht

[BM82] Boyer, Robert S. ; Moore, J S.: *MJRTY - A Fast Majority Vote Algorithm*. 1982

[Bor05] Borkar, Shekhar: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. In: *IEEE Micro* 25 (2005), November, Nr. 6, 10–16. `http://dx.doi.org/10.1109/MM.2005.110`. – DOI 10.1109/MM.2005.110. – ISSN 0272–1732

[KHP+09] Kalibera, Tomas ; Hagelberg, Jeff ; Pizlo, Filip ; Plsek, Ales ; Titzer, Ben ; Vitek, Jan: CDx: A Family of Real-time Java Benchmarks. In: *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA : ACM, 2009 (JTRES '09). – ISBN 978–1–60558–732–5, 41–50

[Law96] Lawton, Kevin P.: Bochs: A Portable PC Emulator for Unix/X. In: *Linux J.* 1996 (1996), September, Nr. 29es. `http://dl.acm.org/citation.cfm?id=326350.326357`. – ISSN 1075–3583

[LHSP+09] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: An Aspect-oriented Operating-system Family for Resource-constrained Embedded Systems. In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2009 (USENIX'09), 16–16

[OSE05] OSEK/VDX Group: Operating System Specification 2.2.3 / OSEK/VDX Group. 2005. – Forschungsbericht. – `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2014-02-06

[SHK+12] Schirmeier, H. ; Hoffmann, M. ; Kapitza, R. ; Lohmann, D. ; Spinczyk, O.: Fail*: Towards a versatile fault-injection experiment framework. In: *ARCS Workshops (ARCS), 2012*, 2012, S. 1–5

[SSE+13] Stilkerich, Isabella ; Strotz, Michael ; Erhardt, Christoph ; Hoffmann, Martin ; Lohmann, Daniel ; Scheler, Fabian ; Schröder-Preikschat, Wolfgang: A JVM for Soft-Error-Prone Embedded Systems. In: ACM (Hrsg.): *Proceedings of the 14th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2013. – ISBN 978–1–4503–2085–6, 21–32

[SSWSP12] Stilkerich, Michael ; Stilkerich, Isabella ; Wawersich, Christian ; Schröder-Preikschat, Wolfgang: Tailor-made JVMs for statically configured embedded systems. In: *Concurrency and Computation: Practice and Experience* 24 (2012), Nr. 8, 789–812. `http://dx.doi.org/10.1002/cpe.1755`. – DOI 10.1002/cpe.1755

[Sti12] Stilkerich, Michael: *Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2012. `http://www.opus.ub.uni-erlangen.de/opus/volltexte/2012/3969/pdf/MichaelStilkerichDissertation.pdf`

[TN93] Taber, A. ; Normand, E.: Single event upset in avionics. In: *Nuclear Science, IEEE Transactions on* 40 (1993), Nr. 2, S. 120–126. `http://dx.doi.org/10.1109/23.212327`. – DOI 10.1109/23.212327. – ISSN 0018–9499

[TSK+11] Thomm, Isabella ; Stilkerich, Michael ; Kapitza, Rüdiger ; Lohmann, Daniel ; Schröder-Preikschat, Wolfgang: Automated Application of Fault Tolerance Mechanisms in a Component-based System. In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA : ACM, 2011 (JTRES '11). – ISBN 978–1–4503–0731–4, 87–95