Eine fragmentierungstolerante Speicherbereinigung für die KESO Java Virtual Machine

Michael Strotz

31.03.2014

Ich versichere, dass ich die Arbeit ohne fremde Hilfe 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 Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 4 (Verteilte Systeme und Betriebssysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den

Masterarbeit

Umfeld: KESO (http://www4.cs.fau.de/Research/KESO) ist eine Java-Laufzeitumgebung für eingebettete Systeme mit dem Fokus auf software-basierten Speicherschutz. KESO ermöglicht die isolierte Ausführung mehrerer Anwendungen auf einem Mikrokontroller. Um eine möglichst ressourcenschonende Laufzeitumgebung bereitzustellen, wird auf viele teure Java-Mechanismen (Reflection, Just-in-Time-Übersetzung, dynamisches Nachladen von Klassen, Exceptions) verzichtet und das komplette System wird vor der Ausführung in C-Code übersetzt.

Aufgabenstellung: Der Einsatz einer automatischen Speicherbereinigung (engl. garbage collection (GC)) kann dazu führen, dass der verwaltete Speicherbereich bei längerer Programmlaufzeit immer stärker fragmentiert wird. Viele Algorithmen zur Defragmentierung kopieren in regelmäßigen Abständen Nutzdaten um, so dass ungenutzte Speicherblöcke wieder miteinander verschmolzen werden können. Dieses Vorgehen stellt in Systemen, in denen Echtzeitanforderungen keine Rolle spielen, kein Problem dar. In Echtzeitsystemen muss jedoch darauf geachtet werden, dass man garantierte Aussagen über die Laufzeit und den Speicherverbrauch der Defragmentierungsphase treffen kann. In dieser Arbeit soll eine Speicherbereinigung für die KESO Multi-JVM implementiert werden, die in der Lage ist mit dem Fragmentierungsproblem umzugehen. Hierzu soll ein Algorithmus entwickelt werden, durch den es möglich ist, sowohl die maximale Ausführungszeit (engl. worst-case execution time (WCET)) zu berechnen also auch eine obere Grenze für den Speicherverbrauch anzugeben, was den Einsatz einer automatischen Speicherbereinigung in Echtzeitsystemen erlaubt.

Betreuung: Dipl.-Inf. Isabella Stilkerich, Dipl.-Inf. Christoph Erhardt

Bearbeiter: Michael Strotz

Abstract

Die Fragmentierung der Halde stellt für automatische Speicherbereiniger in Echtzeitsystemen ein Problem dar. So kann diese dazu führen, dass Speicheranfragen nicht erfüllt werden können, obwohl insgesamt genügend freier Speicher vorhanden ist. Desweiteren erschwert die Fragmentierung die Bestimmung der maximalen Ausführungszeit, die zur Suche eines passenden Freispeicherblocks notwendig ist. Im ungünstigsten Fall müsste die gesamte Halde durchsucht werden.

Ziel dieser Arbeit ist die Implementierung eines fragmentierungstoleranten, echtzeitfähigen, automatischen Speicherbereinigung für die KESO Java Virtual Machine. Diese nennt sich „FragGC“. FragGC verwendet Teile des in KESO bereits vorhandenen „mark-and-sweep“-Speicherbereinigers IdleRoundRobin wieder. Zusätzlich wird Fragmentierungstoleranz erreicht, indem fragmentierte Allokation verwendet wird. Um verlässliche Schranken für die Ausführungszeit von Vektorzugriffen („arrays“) zu erreichen, werden Rückgrate verwendet, um deren Fragmente zu verwalten. Zur Verwaltung der Rückgrate werden replizierende Speicherbereinigungstechniken verwendet. Gegenüber dem IdleRoundRobin entstehen dadurch zusätzliche Kosten in Laufzeit und Codegröße. Die durchschnittliche Ausführungsgeschwindigkeit der Testanwendung *CDj* [8] sinkt dabei auf ca. 76%. Die Codegröße steigt um ca. 22%. Dieser Abfall in der durchschnittlichen Leistung, wird durch die Möglichkeit eine geringere maximale Ausführungszeit zu bestimmen gerechtfertigt.

Auch die maximale Ausführungzeit des Speicherbereinigers wird untersucht. Diese steigt vor allem mit der Anzahl der überlebenden Objekte.

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Arbeit ist die Implementierung einer fragmentierungstoleranten, echtzeitfähigen Speicherbereinigung für die „KESO Multi Java Virtual Machine“. Im folgenden werden die Begriffe „echtzeitfähig“ und „fragmentierungstolerant“ erläutert. Selbstverständlich wird auch die KESO Multi-JVM vorgestellt und begründet, warum ein solches System eine fragmentierungstolerante Speicherbereinigung benötigt.

1.1 Die KESO Multi-JVM

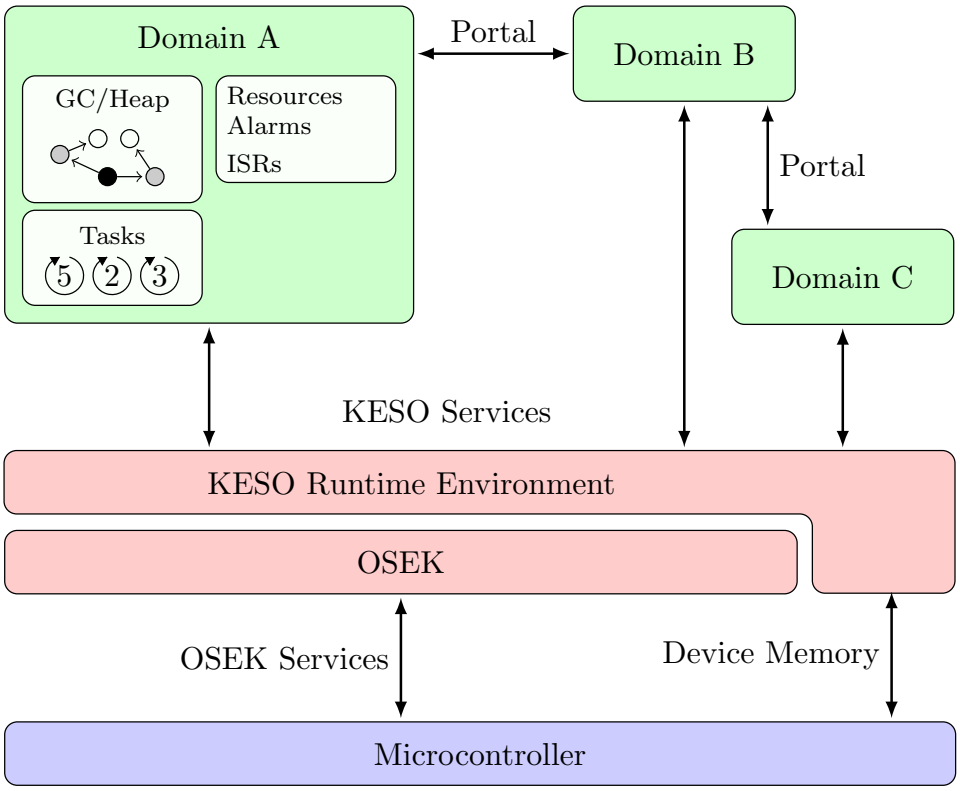

Abbildung 1.1: Übersicht über ein mit KESO gestaltetes System [9]

Bei KESO handelt es sich um eine virtuelle Maschine zur Ausführung von Java-Anwendungen in statischen, eingebetteten Systemen und Echtzeitsystemen.

„Statisch“ bedeutet hier, dass keine Klassen zur Laufzeit nachgeladen werden können. Alle Anwendungen sind beim Beschreiben des Mikrocontrollers bekannt. Alle wichtigen Informationen über die Anwendung, werden in einer Konfigurationsdatei angegeben. Diese enthält unter anderem Informationen über die Nutzung von Betriebssystemressourcen, über die beteiligten Programmfäden (in OSEK: Task [10]) und die Größe des Haldenspeichers sowie dessen Bereinigungsstrategie. Es ist daher nicht nötig, den Bytecode einer Anwendung zu interpretieren, oder während der Laufzeit zu nativem Code zu übersetzen (engl. „just in time compilation“). Anwendungen werden vor der Laufzeit analysiert und zu C-Code übersetzt (engl. „ahead of time“). Dabei kann eine statische Analyse der gesamten Anwendung vorgenommen werden (engl. „whole program optimizations“), welche tiefgreifende Optimierungen erlaubt. So werden Ausführungszeiten, Codegrößen und Speicherverbrauch erreicht, die sich mit Anwendungen vergleichen lassen, die in C-Code verfasst wurden, während gleichzeitig die Robustheit einer typsicheren Sprache genutzt wird.

KESO baut auf OSEK/VDX-kompatiblen [10] Betriebssystemen auf. Abbildung 1.1 zeigt ein „System“ KESOs. Ein System besteht aus einer oder mehreren „Domains“. Diese Domänen bieten Schutz, der mit logischen Adressräumen, die von Mehrzweckbetriebssystemen wie Linux bekannt sind, vergleichbar ist. Dieser Schutz wird dadurch sichergestellt, dass Referenzen auf Objekte einer Domäne nicht an andere Domänen weitergegeben werden können. Javas Typsicherheit verhindert so jeden domänenübergreifenden Zugriff. Um das Weitergeben von Referenzen zu verhindern, besitzt jede Domäne einen eigenen Haldenspeicher (engl. „heap“) und eventuell auch eine eigene Speicherbereinigungsstrategie. Zusätzlich besitzt jede Domäne eine eigene Instanzen der statischen Klassenfelder. Jede Domäne kann daher als eigenständige virtuelle Maschine gesehen werden, weshalb es sich bei KESO um eine Multi-JVM handelt. Die Kommunikation zwischen den Domänen geschieht über sogenannte „Portale“, die den „Java Remote Method Invocations“ oder „Remote Procedure Calls“ ähneln.

Neben dem Verzicht auf das dynamische Nachladen von Klassen, weicht KESO auch in anderen Aspekten von der JVM-Spezifikation ab, um Java-Anwendungen auch auf schwachen Mikrocontrollern ausführen zu können. So werden teure Mechanismen wie Reflexion oder eine komplexe Ausnahmebehandlung nicht unterstützt. An Stelle der Ausnahmebehandlung kann der Aufruf einer einfachen Fehlerbehandlungsroutine und der anschließende Neustart des Systems stehen. Auch „Finalizer“, die beim Freigeben von Objekten aufgerufen werden, werden teilweise nicht unterstützt, da sie zusätzliche Laufzeitkosten verursachen und indeterministisch auftreten.

1.2 Motivation und Ziel der Arbeit

Das Ziel dieser Arbeit ist die Implementierung einer fragmentierungstoleranten, echtzeitfähigen, automatischen Speicherbereinigung. Hierzu werden die Eigenschaften eines Echtzeitsystems und die Probleme, die sich für ein solches System durch Speicherfrag-

mentierung ergeben, erläutert.

1.2.1 Echtzeitsysteme

Unter einem Echtzeitsystem versteht man ein System, das Aufgaben rechtzeitig erledigt. Rechtzeitigkeit bedeutet, dass auf die Eingabe eines Ereignisses vor dem Ablauf eines bestimmten Termins mit einem korrekten Ergebnis geantwortet wird.

Echtzeitsysteme können danach unterschieden werden, welche Konsequenzen das Überschreiten dieser Frist hat.

Verliert ein Ergebnis durch die verzögerte Bereitstellung lediglich an Qualität, handelt es sich um ein weiches Echtzeitsystem. Ein Beispiel hierfür sind Bildqualität und -rate von Multimediasysteme.

Ist nach dem Verstreichen des Termins zwar das errechnete Ergebnis hinfällig, das Gesamtsystem aber weiter ausführbar, handelt es sich um ein festes Echtzeitsystem.

Führt die Terminüberschreitung zu einer „Katastrophe“, also einer Ausnahmesituation, die von der Echtzeitanwendung behandelt werden muss, spricht man von einem harten Echtzeitsystem. Ein Beispiel hierfür ist das rechtzeitige Auslösen von Airbags.

Um die Einhaltung dieser Fristen zu garantieren, muss die maximale Ausführungszeit (WCET, engl. „worst case execution time“) entscheidender Codeabschnitte abschätzbar sein.

1.2.2 Fragmentierung

Unter Fragmentierung versteht man die Zerstückelung oder den Verschnitt des Hauptspeichers. Verschnitt, auch interne Fragmentierung genannt, entsteht dann, wenn einer Anwendung mehr Speicherplatz zugeteilt wird, als sie anfordert und nutzen kann. Von externer Fragmentierung oder Zerstückelung spricht man, wenn der freie Speicher in mehreren unzusammenhängenden Teilstücken vorliegt. Diese Fragmentierung kann dazu führen, dass eine Speicheranforderung nicht erfüllt werden kann, obwohl insgesamt genügend freier Speicher vorhanden ist, aber kein genügend großes Fragment. Des Weiteren erschwert sie die Suche nach einem passendem Fragment, wenn mehrere Freispeicherfragmente untersucht werden müssen, die die Anforderung nicht erfüllen können. Eine Speicherverwaltung heißt dann fragmentierungstolerant, wenn die Effekte der externen Fragmentierung nicht zum Tragen kommen.

1.2.3 Fragmentierung in Echtzeitsystemen

Echtzeitsysteme verfügen typischerweise über begrenzte Ressourcen und sind über einen längeren Zeitraum im Einsatz. Dies steigert das Fragmentierungsrisiko, vor allem in Kombination mit dem charakteristischem Speichernutzungsverhalten von Java-Anwendungen, in Folge dessen häufig Objekte angelegt und wieder freigegeben werden.

Jedoch gilt es gerade in Echtzeitsystemen diese Zerstückelung des Speichers zu vermeiden. So steigt die WCET, die zum Anlegen eines Objektes benötigt wird, ins Unermessliche, wenn nicht vorhersagbar ist, wie viele Freispeicherblöcke untersucht werden müssen, bis ein passender Block gefunden wurde. Des Weiteren muss sicher gestellt werden, dass - entsprechend der Rate an Speicheranforderungen - immer ausreichend Speicherplatz zur Verfügung steht. Fragmentierung, kann zur Folge haben, dass vorhandener freier Speicher nicht genutzt werden kann. Dadurch kann der Fehler „Speicherkapazität erschöpft“ (Java: „OutOfMemoryError“) auftreten, von dem sich die Anwendung nur schwer erholen kann und der meist einen Neustart zur Folge hat.

1.2.4 Automatische Speicherbereinigung in Echtzeitsystemen

Automatische Speicherbereiniger in Echtzeitsystemen müssen zusätzlich Nebenläufigkeit zulassen, da sie von höherprioren Aufgaben (engl. „tasks“) unterbrochen werden können müssen. Verdrängungssperren dürfen nur für möglichst kleine Zeiträume verwendet werden, um die Antwortzeiten anderer Aufgaben nicht unnötig zu erhöhen.

Zusätzlich zu diesen Bedingungen sollte die Speicherbereinigung die Ausführungsgeschwindigkeit der Anwendung so wenig wie möglich verlangsamen und den Speicherverbrauch so wenig wie möglich erhöhen. Geringere Rechenlast und Speicherverbrauch könnten die Ausführung auf günstigeren Mikrocontrollern ermöglichen.

Des Weiteren muss die Speicherbereinigung wie jede andere Aufgabe in den Ablauf des Echtzeitsystems eingeplant werden. Um Speicheranforderungen immer erfüllen zu können, ist darauf zu achten, dass der Speicherbereinigung genügend Zeit bleibt, um unerreichbare Objekte („Müll“) zu löschen. Bei fragmentierungstolerenten Speicherbereinigern reicht es aus, sicher zu stellen, dass genügend freier Speicher vorhanden ist. Spielt Fragmentierung eine Rolle, muss zusätzlich darauf geachtet werden, dass die Fragmente groß genug sind, um die Anfrage zu bedienen.

Um sicherstellen zu können, dass genügend freier Speicher vorhanden ist, muss der Speicherbereiniger entsprechend eingeplant werden. Hierzu ist es notwendig seine maximale Ausführungszeit (WCET) berechnen zu können.

1.3 Aufbau dieser Arbeit

Nachdem die Notwendigkeit einer fragmentierungstoleranten Speicherbereinigung für die KESO Multi-JVM erläutert wurde, werden nun die Gegebenheiten und mögliche Lösungsansätze genauer analysiert. Dazu werden die Speicherbereinigungstechniken beleuchtet, die in KESO bereits vorhanden sind. Des Weiteren werden mögliche Lösungen des Fragmentierungsproblems in Erwägung gezogen. Besonderes Augenmerk liegt hierbei auf dem fragmentierungstoleranten Speicherbereiniger „SCHISM“ [11].

Im Kapitel „Implementierung“ wird die tatsächliche Umsetzung des fragmentierungstoleranten Speicherbereinigers beschrieben. Anschließend folgt die Evaluation dieser Implementierung. Hierzu werden Ausführungszeiten und Codegröße der neuen Speicherbereinigungstechnik mit denen der bereits vorhandenen verglichen, um die Kosten abzuschätzen mit denen die Fragmentierungstoleranz erkauft wurde.

2 Analyse

2.1 KESOs bisherige Haldenverwaltung

KESO verfügt bereits über drei Strategien zur Verwaltung der Halde (engl. „heap“): „RestrictedDomainScope“ (RDS), „IdleRoundRobin“ (IRR) und „CoffeeBreak“. Diese werden im Folgenden bezüglich ihrer Fragmentierungstoleranz und ihres Einflusses auf die Antwortzeit der Anwendung bewertet. Dabei werden Aspekte herausgearbeitet, die zur Implementierung des fragmentierungstoleranten Speicherbereinigers wiederverwendet werden können.

2.1.1 „RestrictedDomainScope“

Bei „RestrictedDomainScope“ (RDS) ist eine Bereinigung des Speichers nicht möglich. Ein einfacher Zeiger markiert die Grenze zwischen dem bereits belegten und dem noch verfügbaren Speicher. Soll ein Objekt angelegt werden, wird dieser Zeiger um die Größe des Objektes in den verfügbaren Speicherbereich verschoben. Das Anlegen verfügt über eine verlässliche WCET, da diese von der Größe des Objektes unabhängig ist. Das Verschieben des Zeigers muss atomar geschehen. Da weiter nicht auf Nebenläufigkeit geachtet werden muss, hat dieses Verfahren nur minimalen Einfluss auf die Antwortzeiten der Anwendung. Außerdem tritt weder nennenswerte interne oder externe Fragmentierung auf. Ein verwandtes Verfahren ist die replizierende Speicherbereinigung (siehe Abschnitt 2.3.2).

Da eine Freigabe des Speichers im Allgemeinen nicht möglich ist, eignet sich diese Strategie nur dann, wenn vorher bekannt ist, wie viel Speicher während der Ausführung angefordert werden wird. In speziellen Fällen kann der RDS im Modus „Reset“ betrieben werden [17]. Hierbei wird nach dem Terminieren aller Anwendungsfäden die gesamte Domäne zurückgesetzt.

2.1.2 „CoffeeBreak“

„CoffeeBreak“ ist ein präziser, nicht-verschiebender Speicherbereiniger, der nach dem Prinzip „Markieren und Löschen“ (engl. „mark and sweep“) verfährt. Während der Bereinigungsphase hält dieser das restliche System unter Verwendung einer Unterbrechungssperre an. Somit muss der Bereiniger nicht auf nebenläufige Modifikationen achten, wodurch eine hohe Ausführungsgeschwindigkeit und damit ein hoher Durchsatz

an freigeräumtem Speicher möglich wird. Jedoch verhält sich die Dauer dieser Unterbrechungssperre nicht asymptotisch konstant, wodurch von einer sehr hohen Beeinträchtigung der Antwortzeiten der Anwendung ausgegangen werden muss.

Markieren und Löschen

Ausgehend von der Wurzelmenge (statische und lokale Referenzen), werden alle erreichbaren Objekte per Tiefensuche besucht und markiert. Alle nicht markierten Objekte werden anschließend gelöscht. Die Markierung der Objekte geschieht durch ein dreifarbiges Schema [1]. Objekte, die noch nicht erreicht wurden, sind weiß gekennzeichnet. Schwarze Objekte wurden bereits untersucht. Objekte, die durch die Referenzen eines schwarzen Objektes erreichbar sind, aber selbst noch nicht auf Referenzen untersucht wurden, werden grau gekennzeichnet.

Der Arbeitsstapel Um die Objekte in Tiefensuche zu besuchen, wird ein Arbeitsstapel verwendet. Dieser Enthält die Referenzen auf alle grauen Objekte. Am Anfang der Markierungsphase wird auf diesem die Wurzelmenge abgelegt. Anschließend werden die Referenzen nacheinander vom Stapel genommen und das zugehörige Objekte nach weiteren Referenzen durchsucht. Diese werden dann auf dem Arbeitsstapel abgelegt. Der Stapel ignoriert dabei alle Referenzen, die auf kein Objekt (*null*) oder ein bereits erreichtes Objekt (schwarz oder grau) zeigen. Nur Referenzen auf bisher unerreichte Objekte (weiß) werden abgelegt. Somit wird jedes Objekt nur einmal auf dem Stapel abgelegt. Ob ein Objekt bereits erreicht wurde wird mit einem Bit pro Objekt gespeichert. Ob das Objekt noch durchsucht werden muss (grau) oder nicht (schwarz), erkennt man daran, ob es noch auf dem Arbeitsstapel liegt.

Präzise Speicherbereinigung

Ein präziser Speicherbereiniger kann Referenzen von Nichtreferenzen unterscheiden. Dieses Verfahren wird durch Javas Typsicherheit ermöglicht. Referenzen können als statische Felder, lokale Variablen in Methoden, oder als Instanzfelder auftreten. Während ein klassischer Speicherbereiniger jedes erreichbare Datum als potentielle Referenz betrachten muss, werden in KESO spezielle Datenstrukturen verwendet. Die statischen Referenzen werden für jede Domäne in einem Vektor gespeichert und sind somit leicht auffindbar. Zur Erkennung der methodenlokalen Referenzen, wird eine sogenannte Henderson-Liste [4] durch den Stapelspeicher des Programmfadens gefädelt. Diese Liste beinhaltet alle Referenzen, die sich derzeit auf dem Stapelspeicher des Fadens befinden.

Um die Referenzen zu finden, die in einem Objekt gespeichert sind, wird ein bidirektionales Objektformat verwendet (siehe Abs. 3.1.2). Dieses Format erlaubt es, Referenzen im Objekt zusammenhängend abzulegen.

Nicht-Verschiebende Speicherbereinigung

Wurde ein Objekt einmal angelegt, bleibt seine Position im Speicher fest. Verschiebende Speicherbereiniger können den Haldenspeicher defragmentieren. In KESO steht diese Möglichkeit nicht zur Verfügung. Dadurch wird der Aufwand vermieden, der mit dem verschieben von Objekten einher geht (siehe Abschnitt 2.3.2), jedoch kann der Speicherbereiniger nicht aktiv gegen Fragmentierung vorgehen. Des Weiteren können Objekte bisher nicht fragmentiert angelegt werden (siehe Abschnitt 2.3.1). Dadurch sind „CoffeeBreak“ und auch „IdleRoundRobin“ nicht fragmentierungstolerant. Somit muss bei beiden von einer hohen WCET beim Anlegen von Objekten ausgegangen werden.

2.1.3 „IdleRoundRobin“ – inkrementelle Speicherbereinigung

Wie „CoffeeBreak“ ist „IdleRoundRobin“ (IRR) ein präziser, nicht-verschiebender Speicherbereiniger, der nach dem Prinzip „Markieren und Löschen“ verfährt [14]. Zusätzlich arbeitet IRR inkrementell. Das heißt er ist jederzeit unterbrechbar – mit Ausnahme einiger kurzer Abschnitte und der Durchsuchung der Stapelspeicher. Um die Konsistenz seiner Datenstrukturen zu garantieren, werden kritische Abschnitte durch Unterbrechungssperren geschützt. Diese Abschnitte sind von konstanter Komplexität und möglichst kurz gehalten, um die Antwortzeiten der Anwendung nicht unnötig zu erhöhen.

Einplanung

Der Speicherbereiniger erhält die niedrigste Priorität im System. Andere Aufgaben können ihn verdrängen. Er selbst kann nur dann eingelastet werden, wenn kein anderer Programmfaden lauffähig ist. IRR muss dadurch nur die Stapelspeicher der Programmfäden untersuchen, die in ihrer Ausführung blockiert werden können, weil sie auf ein bestimmtes Ereignis warten. Dadurch muss auch die Henderson-Liste nur in Methoden verwendet werden, die blockieren können. Des Weiteren vereinfacht es die Synchronisation der Speicherverwaltung, da ausgeschlossen werden kann, dass eine Speicheranforderung von der Bereinigung unterbrochen wird.

Die Freispeicherliste

Freie Speicherbereiche verwaltet IRR in einer verketteten Liste. Diese muss unterbrechungstolerant implementiert werden. Die Speicheranfragen der verschiedenen Anwendungsfäden entnehmen Elemente aus dieser Liste. Während der Löschphase werden freie Speicherbereiche der Liste hinzugefügt. Diese Listenmanipulationen werden durch kurze Unterbrechungssperren geschützt, um die Konsistenz der Liste zu wahren. Sowohl Anfragen als auch der Bereiniger iterieren über die Liste. Hierbei wird eine spezielle Schnittstelle verwendet, die verhindert, dass die Iteration durch die Entnahme von Elementen „entgleist“ (siehe Abschnitt 3.2.2).

Nebenläufige Manipulation des Erreichbarkeitsgraphen

Schreibbarrieren Anwendungsfäden können IRRs Markierungsphase jederzeit verdrängen – mit Ausnahme einiger kritischer Abschnitte. Während dessen kann die Anwendung die Referenzfelder der Objekte beliebig manipulieren. Dadurch können Objekte vom Speicherbereiniger „übersehen“ werden.

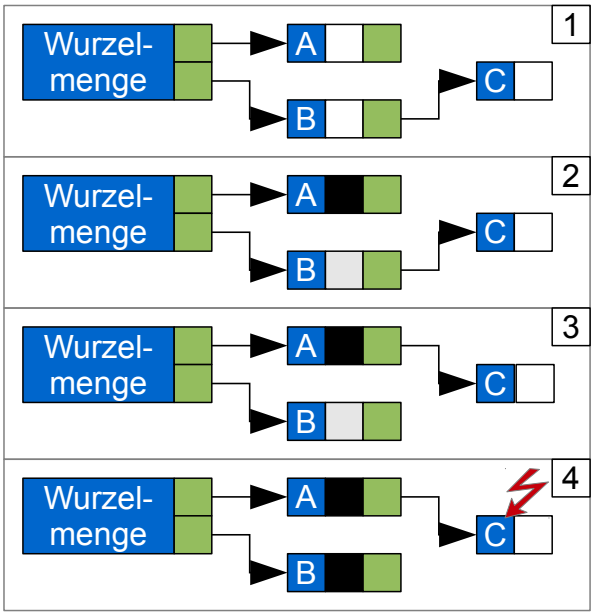

Abbildung 2.1: Objekt C entkommt der Markierung

Abbildung 2.1 verdeutlicht diesen Vorgang. In der Ausgangssituation (1) existieren drei Objekte A, B und C. A und B sind über die Wurzelmenge erreichbar. C ist über B erreichbar. Die Markierungsphase wird begonnen (2) und markiert Objekt A. Jetzt (3) wird diese von der Anwendung unterbrochen, die eine Referenz von A nach C einrichtet und die Referenz von B nach C entfernt. Die Markierungsphase wird anschließend fortgesetzt (4). Nach Abschluss dieser Phase, wurde C nicht markiert. Es wird gelöscht, obwohl es noch verwendet wird. Dieses Problem entsteht immer dann, wenn eine Referenz auf ein weißes Objekt in ein schwarzes Objekt eingetragen wird und die ursprüngliche Referenz auf das weiße Objekt gelöscht wird.

Um dies zu vermeiden, verwendet IRR sogenannte „Schreibbarrieren“. Dazu wird an den Stellen, an denen die Anwendung Referenzfelder beschreibt, Code eingefügt, der verhindert, dass Objekte übersehen werden. In KESO wird die Variante von Yuasa [7] verwendet. Hierbei werden weiße Objekte, auf die die Referenzen zeigen, die überschrieben werden sollen, grau gefärbt werden.

Abbildung 2.2 verdeutlicht die Funktionsweise der Schreibbarrieren anhand von Schritt 2 des vorherigen Beispiels. Die Referenz des Objektes A zeigt auf kein Objekt

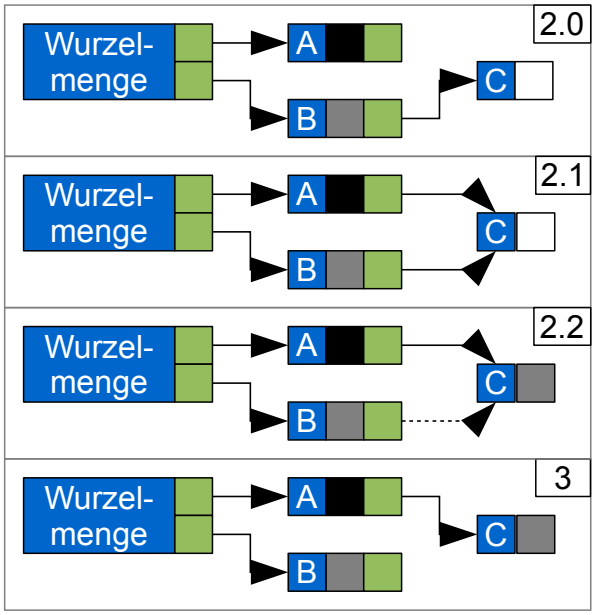

Abbildung 2.2: Schreibbarriere nach Yuasa

(*null*), daher ist beim Überschreiben (2.1) kein Graufärben nötig. Anschließend soll die in B gespeicherte Referenz mit *null* überschrieben werden (2.2). Da diese Referenz momentan auf das weiße C zeigt, wird dieses grau gefärbt. Mit dem Nullen der Referenz wird Schritt 3 erreicht. Wobei C nun grau statt weiß ist und vor Beendigung der Markierungsphase geschwärzt werden wird.

Der Speicherbereiniger schiebt die Adressen aller noch zu untersuchenden Objekte auf den Arbeitsstapel. Graue Objekte unterscheiden sich von schwarzen dadurch, dass sie noch auf diesem Stapel liegen. Ein weißes Objekt grau zu färben bedeutet, dieses Objekt zu schwärzen und auf den Stapel zu legen. Daher wurden die nebenläufigen Zugriffe durch die Schreibbarrieren und dem Bereiniger auf den Stapel synchronisiert.

Durchsuchen der Stapelspeicher Da Schreibbarrieren einen zusätzlichen Aufwand bedeuten, werden für methodenlokale Referenzen keine Schreibbarrieren verwendet. Um das Übersehen von Referenzen zu vermeiden, darf der Stapelspeicher eines Anwendungsfadens während der Durchsuchung nicht verändert werden. Dazu wird der Faden blockiert. Dies vermeidet ebenfalls, dass Methodenrahmen ab- und wieder aufgebaut werden. Um eine Prioritätsumkehr zu vermeiden, müssen auch alle Fäden, die eine niedrigere Priorität als der durchsuchte Faden besitzen, blockiert werden.

Nur die Stapel der Fäden, die blockiert werden können, müssen durchsucht werden. Die Komplexität der Suche ist linear zur Tiefe des Stapelspeichers. Diese lässt sich vor der Laufzeit abschätzen, indem man analysiert auf welcher Aufrufebene der blockieren-

de Betriebssystemaufruf $WaitEvent()$ auftaucht. Jedoch dürfen blockierende Methoden dazu nicht rekursiv aufgerufen werden.

Der Stapel eines Fadens zählt zur Wurzelmenge und muss vor der Traversierung der Haldenobjekte untersucht werden. Weiter muss er nach der Aktivierung der Schreibbarrieren untersucht werden, da sonst Referenzen, über Objekte von einem Faden zum anderen fliehen könnten.

Auswahl der Domäne

Seinen Namen hat der IdleRoundRobin von der Art, wie er die Domäne auswählt, die als nächstes bereinigt wird. Der IRR wechselt während sich das System im Leerlauf befindet reihum durch die Domänen. Für jede Domäne wird ein sogenannter „need“-Wert berechnet [14]. Zeigt dieser Wert an, dass in der Domäne eine Speicherbereinigung durchgeführt werden sollte, wird dies getan. Der „need“-Wert berechnet sich wie folgt:

$$need = \text{FreieSlots} - \text{NeuAngeforderteSlots} - \frac{\text{Haldengröße}}{4}$$

„Slot“ bezeichnet die kleinste anforderbare Speichereinheit der Halde. „FreieSlots“ ist die Anzahl der Slots, die gegenwärtig nicht belegt sind. „NeuAngeforderteSlots“ bezeichnet die Anzahl der Slots, die seit der letzten Bereinigungsphase angelegt wurden. „Haldengröße“ ist die Größe der Halde in Slots. Wird ein negativer Wert berechnet, wird die Speicherbereinigung ausgelöst. Das passiert, wenn weniger freier Speicher verfügbar ist, als seit der letzten Bereinigung angefordert wurde. Zur Sicherheit wird vom freien Speicher noch 25% der Größe der Haldengröße abgezogen. Grund für diese Heuristik ist, dass eingebettete Systeme oft periodische Aufgaben erledigen. Wurde seit der letzten Allokation noch nicht die Hälfte des freien Speichers angefordert, wird vermutet, dass die Domäne noch bis zur nächsten Gelegenheit, den Speicher zu bereinigen, mit ihrem Speicher auskommt.

2.1.4 Bewertung

„RestrictedDomainScope“ bürdet der Anwendung starke Einschränkungen auf, da die Menge des gesamten Speichers, der während der Laufzeit angefordert werden könnte, vorher abgeschätzt werden können muss.

Diese Einschränkungen lockert „CoffeeBreak“, der die Freigabe von angefallenem „Müll“ ermöglicht. Jedoch blockiert dieser die gesamte Anwendung, während er den Speicher untersucht und säubert, wodurch Echtzeitkriterien verletzt werden können. Des Weiteren ist er nicht fragmentierungstolerant.

„IdleRoundRobin“ erlaubt es ebenfalls Speicher freizugeben, jedoch arbeitet er inkrementell und benutzt nur kurze Unterbrechungssperren konstanter Komplexität. Somit ist die mögliche Verzögerung der Antwort auf ein Ereignis, die durch die Speicherbereinigung entsteht, klar abschätzbar. Jedoch kommt auch hier Fragmentierung zum Tragen,

wodurch es enorm erschwert wird, abzuschätzen, ob jede Speicheranfrage bedient werden kann. In Abschnitt 2.3 werden deshalb fragmentierungstolerante Speicherbereinigungstechniken betrachtet.

2.2 Typisches Speichernutzungsverhalten in Java

Im Speichernutzungsverhalten einer Java-Anwendung lassen sich Charakteristika finden, die bei der Planung der Speicherverwaltung berücksichtigt werden sollten, oder nützlich sind. Dazu werden Vermutungen über die Größe von Objekten angestellt. Des Weiteren werden Objekte nach ihrer „Überlebensfähigkeit“ in Kategorien unterteilt. Diese Kategorien können genutzt werden, um die WCET zu bestimmen, die zur Bereinigung des Speichers notwendig ist. Die Lebensdauer und Größe von Objekten wird anschließend anhand der Testanwendung CDj [8] überprüft.

2.2.1 Größe von Objekten

Im Gegensatz zur Programmiersprache C sind Speicheranfragen in Java immer typsicher. So lässt sich unterscheiden, ob es sich bei einem anzulegenden Objekt um eine Instanz einer explizit definierten Klasse handelt, oder um einen Vektor (engl. „array“). Vektoren und Nicht-Vektoren („reguläre Objekte“) unterscheiden sich in der Art, wie auf ihre Felder zugegriffen wird. Auf Vektoren ist wahlfreier Zugriff (engl. „random access“) unter Verwendung von variablen Indices möglich, während die Position eines Instanzfeldes regulärer Objekte zur Übersetzungszeit bekannt ist.

Anders als in C++ ist es Java-Klassen nicht erlaubt, Instanzen einer Klasse als Wert in einem Feld zu speichern. In Instanzfeldern dürfen nur Referenzen auf Objekte gespeichert werden. Dadurch bleiben Instanzen benutzerdefinierter Klassen im Vergleich zu Vektoren verhältnismäßig klein, solange sie nicht - beispielsweise von einem Codegenerator - mit einer unüblich hohen Anzahl an Feldern ausgestattet werden.

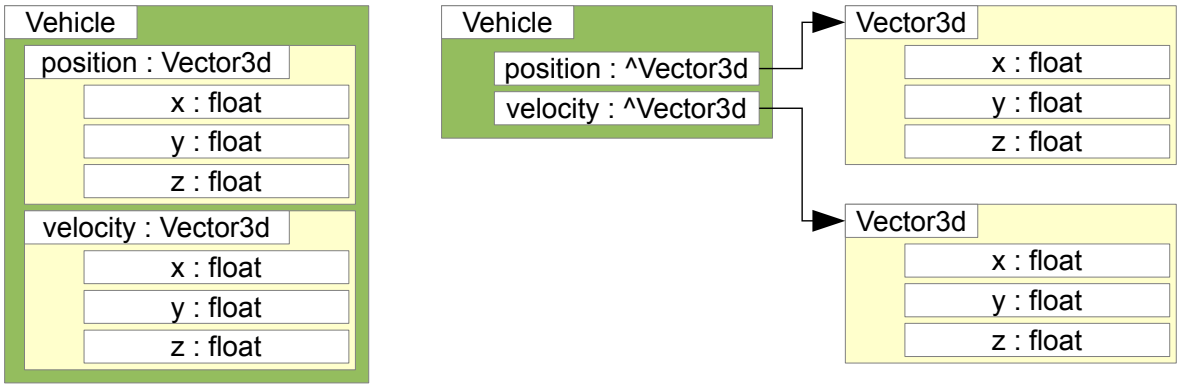

Abbildung 2.3: Gegenüberstellung von Wert- und Referenz-Instanzfeldern

Abbildung 2.3 verdeutlicht warum Objekte in C oder C++ schneller wachsen können als in Java. Links ist die Speicherung eines Objektes als Wert zu sehen, die durch die Programmiersprache Java nicht ausgedrückt werden kann: „Vehicle“ speichert Instanzen der Klasse „Vector3d“ als Wert. Beim rechten „Vehicle“ werden „position“ und „velocity“ durch Referenzen auf zwei getrennt angelegte „Vector3d“-Objekte dargestellt. Deshalb sind in Java – bei objektorientierter Programmierung – mehrere kleine Objekte wahrscheinlicher, als wenige große Objekte.

Obwohl es in der Sprache Java nicht explizit beschrieben werden kann, können sich Übersetzter dennoch entscheiden, Objekte doch als Instanzfelder zu speichern. Dies nennt man Objekteinbettung (engl. „object inlining“) [18].

2.2.2 Überlebensfähigkeit von Objekten

Der Aufwand der Speicherbereinigung hängt maßgeblich von der Anzahl der „überlebenden“ Objekte ab. Daher wird beleuchtet, in welchen Fällen ein Überleben der Speicherbereinigungsphase ausgeschlossen werden kann. Nach diesem Kriterium werden die Objekte in verschiedene Kategorien unterteilt. Um diese zu untermauern, werden alternative Speicherverwaltungsstrategien für die jeweiligen Muster in Betracht gezogen. Diese lassen Rückschlüsse auf die Lebensspanne der Objekte zu. Wird während dieser Lebensspanne eine Speicherbereinigung zugelassen überlebt das Objekt. Kann kein Bereinigung stattfinden, kann das Objekt keine Bereinigungsphase überleben.

Hierbei ist zu beachten, dass es dem Speicherbereiniger in KESO nicht gestattet ist, laufende Programmfäden zu unterbrechen (siehe Abschnitt 2.1.3). Des Weiteren müssen Objekte die während der Speicherbereinigungsphase angelegt wurden, nicht untersucht werden.

Viele Java-Anwendungen scheinen die sogenannte „schwache Generationenhypothese“ zu erfüllen. Die schwache Generationenhypothese [7] besagt, dass die meisten Objekte eine kurze Lebensspanne besitzen, also kurz nachdem sie angelegt wurden, wieder freigegeben werden.

Methodenlokale Objekte

Methodenlokale Objekte existieren nur für die Dauer einer Methode. Referenzen auf diese Objekte „fliehen“ nicht aus der Methode. Das heißt, dass sie nicht an Klassenfelder oder Instanzfelder von Objekten, die ebenfalls aus der Methode „fliehen“, zugewiesen werden. Zudem werden sie nicht als Rückgabewert der Methode ausgegeben. Diese Eigenschaft kann mit einer Fluchtanalyse (engl. „escape analysis“) überprüft werden [9]. Alternativ zur Haldenallokation können diese Objekte im Stapelrahmen der Methode angelegt werden. Auf Grund dessen können diese Objekte keine Bereinigungsphase überleben, wenn ihr Methodenrahmen bereits wieder abgebaut wurde. Also überleben methodenlokale Objekte keine Bereinigungsphase, solange sie sich nicht in einer blockierenden Methode befinden.

```
class Vector2d {
      float x, y;

      double magnitude() {
            return Math.sqrt(x*x + y*y);
      }
      static void subtract(Vector2d a, Vector2d b, Vector2d result) {
            result.x = a.x - b.x;
            result.y = a.y - b.y;
      }
      static double distance(Vector2d a, Vector2d b) {
            final Vector2d difference = new Vector2d();
            subtract(a,b,difference);
            return difference.magnitude();
      }
}
```

Abbildung 2.4: „difference“ – Beispiel eines Methodenlokalen Objektes

Abbildung 2.4 zeigt das methodenlokale Objekt „difference“, das nicht aus der umgebenden Methode „distance“ ausbricht. Daher könnte es auf dem Stapel der Methode angelegt werden. Da die Methode „distance“ während ihrer Laufzeit keine Speicherbereinigung zulässt, wird „difference“ in der nächsten Speicherbereinigungsphase freigegeben werden.

Regionenlokale Zwischenergebnisse

Regionenlokale Zwischenergebnisse umfassen Objekte, die aus dem Methodenrahmen in dem sie erzeugt werden ausbrechen. Trotzdem werden diese Objekte nicht für andere Programmfäden sichtbar. Des Weiteren lassen sie sich in stapelbaren Speicherregionen anlegen (engl. „region based memory management“ [16]). So stellt der typsichere C-Dialekt Cyclone [6] beispielsweise ein regionenbasiertes Typsystem zu Verfügung. Stapelbare Speicherregionen ähneln den Stapelrahmen von Methoden. Allerdings existieren diese Regionen parallel zu den Methodenrahmen. Somit kann eine Region mehrere Methodenrahmen umfassen. Genauso können in einem Methodenrahmen mehrere Regionen auf und abgebaut werden. Objekte einer Region können Referenzen auf jene der eigenen oder einer umgebenden Region besitzen. Sie dürfen jedoch keine Referenzen auf Objekte besitzen, die sich in einer Subregion befinden.

Ruft der Programmfaden in der Region, in der diese Objekte „leben“ eine blockierende Methode auf, können sie Bereinigungsphasen überleben. Ist dies nicht der Fall, überleben sie die nächste Bereinigungsphase nicht, da der Programmfaden nicht vom Bereiniger unterbrochen werden kann.

Weiter lassen sich diese Objekte dadurch unterscheiden, ob ihre Größe zur Übersetzungszeit bestimmbar ist.

```
static Vector2d subtract(Vector2d a, Vector2d b) {
      final Vector2d result = new Vector2d();
      result.x = a.x - b.x;
      result.y = a.y - b.y;
      return result;
}
static double distance(Vector2d a, Vector2d b) {
      final Vector2d difference = subtract(a,b);
      return difference.magnitude();
}
```

Abbildung 2.5: Regionenlokale Zwischenergebnisse fester Größe

Regionenlokale Zwischenergebnisse fester Größe Bei Abbildung 2.5 handelt es sich um eine abgewandelte Version von Abbildung 2.4. Nun flieht das Objekt „result“ aus der Methode „subtract“. Es flieht jedoch nicht aus der Methode „distance“. Das Objekt „result“ kann weiter im Stapelrahmen von „distance“ angelegt werden, wenn der Übersetzer in der Lage ist, dieses Beispiel in das vorherige zu überführen.

Regionenlokale Zwischenergebnisse dynamischer Größe Abbildung 2.6 zeigt regionenlokale Objekte. In „processVectors“ wird anfangs eine Region eröffnet. Innerhalb dieser Region wird eine Liste von „Vector2d“-Objekten erstellt. Anschließend wird „reduce“ diese Liste übergeben, welches ein Ergebnis errechnet. Zuletzt wird die Region zerstört und das Ergebnis von „reduce“ zurückgegeben. Beim zerstören der Region, werden auch alle Objekte der Liste freigegeben. Es ist zu sehen, dass die Region die Rahmen der Funktionen „collectVectors“ und „reduce“ umfasst. Mit dem Schlüsselwort „region“ annotierte Referenzen sollen in die aktuelle Region zeigen. „region(this)“ soll bedeuten, dass die Referenz auf ein Objekt zeigt, das sich in der selben Region wie dieses Objekt befindet (oder einer Elternregion). „region.new“ meint eine regionlokale Allokation.

Fadenlokale Objekte

„Fadenlokale Objekte“ meint Objekte, die einerseits anderen Ausführungssträngen nicht zugänglich werden, aber andererseits nicht in Regionen gehalten werden können. Diese Objekte müssen mit einer automatischen Speicherbereinigung verwaltet werden. Blockiert der zugehörige Faden nicht, kann davon ausgegangen werden, das diese Objekte keine Bereinigungsphase überleben. Ansonsten muss davon ausgegangen werden, dass das Objekt überlebt. Gäbe „reduce“ in Abbildung 2.6 statt „double“ ein Objekt dynamischer Größe zurück, wäre das ein Beispiel für ein fadenlokales Objekt. Dieses Objekt könnte nicht in der aktuellen Region angelegt werden, da es sonst bei deren Zerstörung gelöscht werden würde.

```
static class VectorList {
      VectorList region(this) next;
      Vector2d region(this) value;
      VectorList() { next = this; }
      static void append(VectorList region head, Vector2d region value) {
            VectorList element = region.new VectorList();
            element.value = value;
            element.next = head.next;
            head.next = element;
      }
}
static VectorList region collectVectors() {
      VectorList region list = region.new VectorList();
      /*erschaffe variable Anzahl an Listenelementen*/
      while(hasVectors()) {
            Vector2d region vec = region.new Vector2d();
            readNextVector(vec);
            VectorList.append(list,vec);
      }
      return list;
}
static double reduce(VectorList list) {
      /*...*/
}
static double processVectors() {
      double result;
      region {
            VectorList region list = collectVectors();
            result = reduce(list);
      }
      /*hier wird „list“ gelöscht*/
      return result;
}
```

Abbildung 2.6: Regionenlokale Objekte

Fadenübergreifende Kommunikation

Im Folgenden werden Objekte betrachtet, die mehreren Ausführungssträngen zugänglich sind. Die an der Fadenübergreifende Kommunikation beteiligten Objekte lassen sich in konsumierbare und unkonsumierbare unterteilen.

Konsumierbare Objekte Konsumierbare Objekte können freigegeben werden, sobald sie konsumiert wurden. Kann der Speicherbereiniger zwischen dem Erstellen des Objektes und dem Konsumieren nicht anlaufen, sterben diese Objekte in der nächsten Bereinigungsphase. Kann der Ausführungsstrang während der Konsumierung eines Objektes blockieren, können alle unkonsumierten Objekte dieser Art überleben.

Beispielsweise könnte ein höherpriorer Faden Arbeitsaufträge an einen niederprioren

Faden schicken. Ob die Auftragsobjekte selbst überleben, hängt davon ab, ob der Faden während der Abarbeitung eines Auftrags blockieren kann. Blockiert der Faden nicht, werden alle Aufträge abgearbeitet und die entsprechenden Objekte können freigegeben werden.

Hierbei ist auf die minimale Zwischenankunftszeit der konsumierbaren Objekte zu achten. Dem niederprioren Faden muss genug Zeit bleiben, um die ankommenden Objekte zu verarbeiten. Falls die Objekte überleben könnten, muss auf diese Art auch die maximale Anzahl an Überlebenden abgeschätzt werden.

Unkonsumierbare Objekte Bei unkonsumierbaren Kommunikationsmitteln geschieht der Informationsaustausch über die Eigenschaften des Objektes. Diese Objekte werden selten oder niemals freigegeben werden.

Im einfachsten Fall handelt es sich um ein gemeinsam genutztes Objekt, dass einen Zustand ausdrückt. Die Warteschlange, in die die oben genannten Aufträge eingereiht werden, ist ebenfalls nicht konsumierbar. Ebenso könnten in diese Warteschlange wiederverwertbare Auftragsobjekte eingereiht werden. Diese würden einmal angelegt und immer wieder eingereiht werden. Die Auftragsobjekte wären dann keine „konsumierbaren Objekte“ mehr.

Sind diese Objekte von fester Größe, können sie statisch angelegt werden.

Chronikobjekte

Chronikobjekte ermöglichen es der Anwendung Informationen über ihre Vergangenheit zu speichern. Diese Objekte überleben mehrere Bereinigungszyklen. Da keine Anwendung über die Zeit Objekte anhäufen darf, muss es eine obere Schranke für die Anzahl der Chronikobjekte geben.

Beispielsweise könnte eine Anwendung zu Diagnosezwecken die 20 letzten wichtigen Ereignisse protokollieren. Die Testanwendung CDj [8] simuliert dieses Verhalten, indem sie Byte-Vektoren in einem Ringpuffer anlegt.

Unkategorisierte Objekte

Unter Verwendung der genannten Kategorien sollte es möglich sein, die meisten Anwendungen aufzubauen. Damit ist auch eine obere Schranke für die Anzahl der Objekte, die Bereinigungsphasen überleben, bestimmbar. Werden Objekte verwendet, die keiner der genannten Kategorien entsprechen, ist darauf zu achten, dass ihre Überlebenschancen vorhersagbar sind.

Des Weiteren sind „untote Objekte“ zu vermeiden. Hierbei handelt es sich um Objekte, die von der Anwendung zwar referenziert, aber nicht mehr verwendet werden. Sie entstehen, wenn Referenzen auf Objekte, die nicht mehr benötigt werden, nicht mit *null* überschrieben werden.

2.2.3 Analyse der Testanwendung CDj

Um die obigen Aussagen zu überprüfen, wurde bei der Testanwendung CDj [8] während der Ausführungszeit protokolliert, wie viele Objekte welchen Typs auf der Halde (engl. „heap“) angelegt wurden und wann ihre Löschung erfolgte.

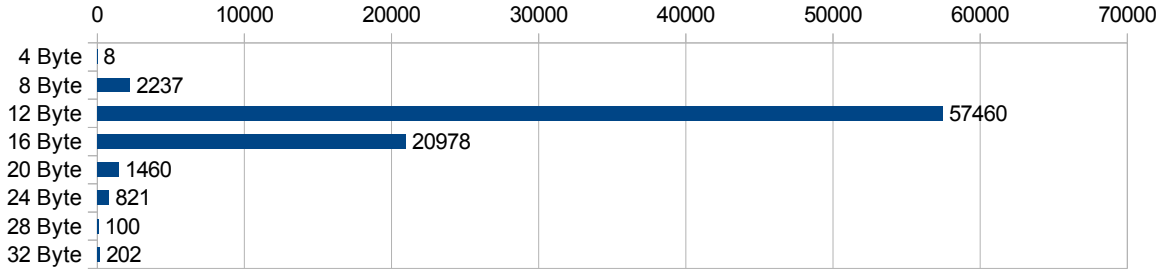

Abbildung 2.7: Häufigkeit aller Größen regulärer Objekte (CDj)

Das Histogramm in Abbildung 2.7 zeigt die Häufigkeit, mit der bestimmte Objektgrößen auftauchen. Horizontal wird die Anzahl der angelegten Instanzen aufgetragen, vertikal die Größe. Es ist zu erkennen, dass Objekte im Median 12 Byte groß sind. Dabei nimmt der Objektkopf 4 Byte und die Instanzfelder 8 Byte ein.

	Anzahl	Bezeichner	Größe [Byte]
1	56.736	immortal/persistentScope/transientScope/Vector2d	12
2	11.262	javacp/util/HashMap$HashEntry	16
3	5.682	javacp/util/ArrayList	16
4	2.402	immortal/persistentScope/transientScope/Vector3d	16
5	1.073	java/lang/String	8
6	1.020	javacp/util/LinkedList$Entry	16
7	992	javacp/util/AbstractList$1	20
8	511	java/lang/StringBuffer	12
9	464	javacp/util/LinkedList	20
10	464	javacp/util/LinkedList$LinkedListItr	24

Abbildung 2.8: Am häufigsten angelegte Klassen (CDj)

Abbildung 2.8 zeigt die am häufigsten angelegten Klassen. Die häufigste Klasse „Vector2d“ scheint Möglichkeiten zur Optimierung zu bieten. Diese zwölf Byte große Klasse könnte möglicherweise als Wert-Instanzfeld gespeichert werden (z.B. durch „object inlining“ [18]). Eventuell können ihre Instanzen in Zukunft auch auf dem Stapelspeicher angelegt werden – z.B. durch eine Fluchtanalyse (engl. „escape analysis“). Einträge von Streutabellen („HashMap.Entry“) oder Listen werden vermutlich immer auf der Halde platziert werden müssen. Das ist Spekulation, trotzdem sollte im Median pessimistisch von einer Objektgröße von 12 bis 16 Byte ausgegangen werden.

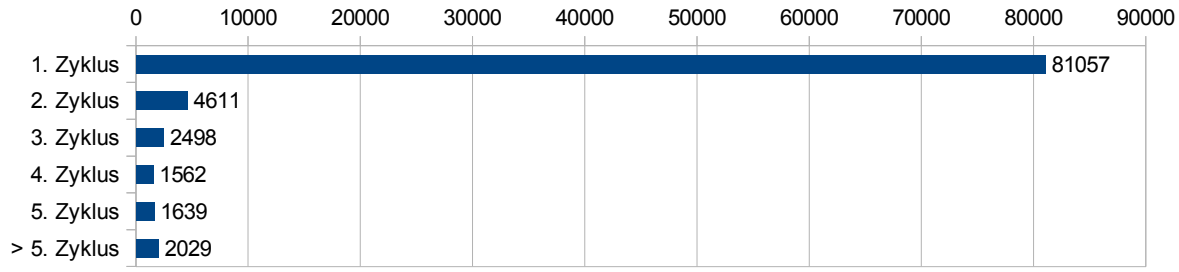

Abbildung 2.9: Häufigkeit der Lebensdauer von Objekte (CDj)

Das Histogramm in Abbildung 2.9 zeigt die Häufigkeit bestimmter Lebensspannen von Objekten. Vertikal lässt sich ablesen in welchem Bereinigungszyklus ein Objekt nach seiner Erschaffung freigegeben wurde. Horizontal ist die Anzahl der Objekte angegeben, die über eine entsprechende Lebensdauer verfügen. Insgesamt wurden 20 Bereinigungszyklen durchgeführt. Es ist zu erkennen, dass die meisten Objekte im ersten Zyklus nach ihrer Erschaffung freigegeben werden. Damit bestätigt die Testanwendung die „schwache Generationenhypothese“ [7].

2.2.4 Einplanung des Speicherbereinigers

Der Aufwand der Speicherbereinigung hängt maßgeblich von der Anzahl der „überlebenden“ Objekte ab (siehe Abschnitt 3.4.5). Zusätzlich fällt die Größe der Halde und der freigegebenen Objekte ins Gewicht.

Der Speicher sollte dann bereinigt werden, wenn möglichst wenige Objekte überleben. Dadurch sinkt die Ausführungszeit der Bereinigungsphase. Zudem sind weniger Bereinigungsphasen nötig. Wann wenige Objekte überleben, lässt sich durch die verschiedenen Objektkategorien (Abschnitt 2.2.2) bestimmen.

Zugleich sollte dem Speicherbereiniger oft genug und lange genug Zeit gegeben werden, um die von der Anwendung erstellten Objekte wieder freizugeben. Die Anzahl und Größe der Objekte wird durch die „Allokationsrate“ ausgedrückt. Diese ergibt sich aus der minimalen Zwischenankunftszeit von Ereignissen und aus der Größe und Menge von Objekten, die angelegt werden könnten, um auf dieses Ereignis zu antworten. Als Ereignis wird auch das Ausführen von periodischen Aufgaben gesehen.

Wird bei bestimmten Aufgaben zu viel Speicher angefordert, muss die Größe der Halde erhöht werden. Alternativ kann die Aufgabe in mehrere Teilaufgaben zerlegt werden, um zwischen den Teilaufgaben den Speicher zu bereinigen. Dies erhöht jedoch die Anzahl der überlebenden Objekte und die Antwortzeit.

Im Vergleich zur Programmiersprache C wird die Freigabe von dynamisch angelegten Objekten auf die Schlupfzeit (engl. „slack time“) verschoben. In der Programmiersprache C wird dynamisch angelegter Speicher mit der Funktion „free“ freigegeben. Sie findet

also mit der Priorität des Fadens statt.

Kommt Fragmentierung zum Tragen, kann es vorkommen, dass Speicheranforderungen trotz der richtigen Einplanung des Speicherbereinigers nicht erfüllt werden können. In diesem Fall ist insgesamt genügend freier Speicherplatz verfügbar. Langlebige Objekte zerteilen die freien Blöcke des Speichers jedoch so, dass kein zusammenhängender Speicherbereich ausreichender Größe gefunden werden kann.

2.3 Fragmentierungstolerante Speicherbereinigungstechniken

Fragmentierung erschwert es enorm, abzuschätzen, ob alle Speicheranforderungen der Anwendung erfüllt werden können. Kommt Fragmentierung nicht zum Tragen, kann der Speicherbereiniger, wie in Abschnitt 2.2.4 beschrieben wurde, eingeplant werden.

Im Folgenden werden fragmentierungstolerante Speicherbereinigungstechniken vorgestellt.

2.3.1 Fragmentierte Allokation

Fragmentierung erschwert es, genügend große zusammenhängende Speicherbereiche zu finden. Fragmentierte Allokation erlaubt es Objekte auf mehrere unzusammenhängende Bereiche verteilt anzulegen [12]. Dazu werden Objekte in Fragmente fester Größe unterteilt. Diese Fragmente sind ebenfalls die kleinste Einheit, in der der freie Speicher verwaltet wird. So kann freier Speicherplatz immer genutzt werden.

Die Fragmente eines regulären Objektes werden verkettet. Dazu wird in jedes Fragment ein Zeiger auf das jeweils nächste Fragment eingefügt. Instanzfeldzugriffe sind von konstanter Komplexität, da für jedes Feld das Fragment in dem es sich befindet zur Übersetzungszeit bekannt ist. Weil Objekte in der Regel aus wenigen Fragmenten bestehen (siehe Abschnitt 2.2), ist die Anzahl der Indirektionen, die durch die Verkettung aufgelöst werden müssen vertretbar. Die Zeit, die zum Anlegen einer Instanz benötigt wird, verhält sich im ungünstigsten Fall linear zur Anzahl der benötigten Fragmente. Da die Anzahl der Fragmente von der Klasse des Objekts abhängt, existiert für jede Klasse eine konstante WCET.

Vektoren erlauben wahlfreien Zugriff über Indices, die erst zur Laufzeit bekannt sind. Würden die Fragmente eines Vektors ebenfalls verkettet werden, verhielte sich die Komplexität eines Zugriffs linear zum Index. Daher werden diese durch eine Baumstruktur verknüpft, wodurch sich die Komplexität logarithmisch zur maximalen Vektorgröße verhält. Da die maximale Vektorgröße eine Konstante ist, ergibt sich theoretisch eine asymptotisch konstanter Aufwand. Allerdings müssten bei einer 32-Bit-Architektur mit einer Fragmentgröße von 32 Byte neun Baumebenen durchschritten werden, um zu dem Fragment zu gelangen, das die Nutzdaten enthält. Dadurch kann sich die WCET eines

Vektorzugriffes stark erhöhen. Die Zeit, die zum Anlegen eines Vektors benötigt wird, verhält sich schlimmstenfalls linear zur Anzahl der benötigten Fragmente.

Bei der Wahl der Fragmentgröße, ist die mittlere Größe der Objekte zu beachten. Wird sie zu groß gewählt, tritt interne Fragmentierung auf. Wird sie zu klein gewählt, muss zu viel Verwaltungsaufwand betrieben werden, um die Nutzdaten unterzubringen. So wären für Vektoren tiefere Baumstrukturen und für reguläre Klassen längere Verkettungen notwendig.

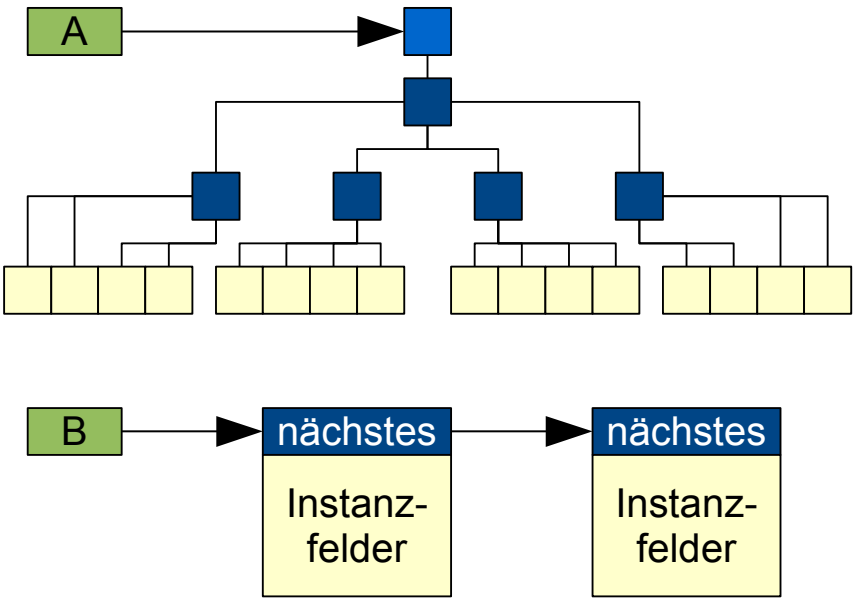

Abbildung 2.10: Fragmentierte Allokation

Abbildung 2.10 stellt dar, wie Fragmentiert angelegte Objekte aussehen könnten. *A* zeigt auf einen fragmentierten Vektor, der von eine Baumstruktur aus Fragmenten verwaltet wird. Jedes Fragment zeigt dabei auf vier weitere Fragmente. Am Ende besteht der Vektor aus 16 Nutzfragmenten. *B* zeigt auf ein fragmentiertes Objekt, das aus zwei Fragmenten besteht. Ein Verkettungszeiger zeigt auf das jeweils nächste Fragment.

2.3.2 Replizierende Speicherbereinigung

Replizierende Speicherbereiniger teilen den Haldenspeicher in eine inaktive und eine aktive Hälfte. Speicheranfragen werden nur von der aktiven Hälfte bedient. Abbildung 2.11 stellt dies dar. Wird die Speicherbereinigung ausgelöst, werden alle erreichbaren Objekte (schwarz) in die inaktive Hälfte kopiert. Damit tauschen die beiden Hälften ihre Rollen.

Alle Objekte werden hintereinander angelegt, indem ein einfacher Zeiger um den angeforderten Betrag erhöht wird (vgl. 2.1.1 „RestrictedDomainScope“). Daher verhält sich die Ausführungszeit, die zum Anlegen eines beliebig großen Objektes oder Vektors

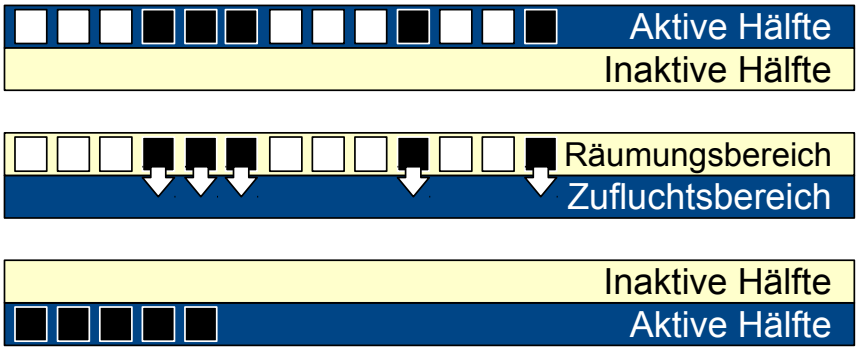

Abbildung 2.11: Replizierende Speicherbereinigung

benötigt wird, immer asymptotisch konstant. Ebenfalls tritt innerhalb der jeweiligen Hälfte keine Fragmentierung auf. Jedoch können beide Hälften nicht gleichzeitig genutzt werden. Diese Form der „Fragmentierung“ hat zwar keine Auswirkungen auf die Planbarkeit der Speicherbereinigung bezüglich Allokations- und Freigaberate, jedoch steigt der Platzbedarf der Halde auf das Doppelte.

Auch das Umkopieren, das während der Bereinigungsphase stattfindet, bringt Schwierigkeiten mit sich. Alle Referenzen, die auf Instanzen zeigen, die sich in der inaktiven Hälfte befinden, müssen auf die entsprechende Instanz in der aktiven Hälfte umgebogen werden. Des Weiteren müssen nebenläufige Schreibzugriffe auf inaktive Instanzen, die während der Anpassung der Referenzen geschehen könnten, auf die entsprechenden aktiven Instanzen übertragen werden.

2.3.3 Speicherbereinigung mit Generationenkonzept

Speicherbereiniger mit Generationenkonzept sind aus der Erkenntnis entstanden, dass der Aufwand der Bereinigungsphase hauptsächlich durch Objekte verursacht wird, die diese „überleben“. Markierende Speicherbereiniger müssen langlebige Objekte immer wieder markieren und nach Referenzen durchsuchen. Replizierende kopieren diese Objekte immer wieder zwischen den beiden Hälften hin und her. Außerdem ist zu erkennen, dass die meisten Objekte nur für einen kurzen Zeitraum existieren (siehe Abschnitt 2.2). Deshalb konzentrieren sich generationelle Speicherbereiniger auf „junge“ Objekte und versuchen den Aufwand für die Behandlung langlebiger Objekte zu mindern.

Um dies Umzusetzen wird die Halde in mehrere Generationen aufgeteilt. Meistens gibt es eine „junge“ und eine „alte“ Generation. Diese Generationen können von verschiedenen Speicherbereinigungstechniken verwaltet werden, üblich sind replizierende Verfahren. Die Bereinigungsphase wird für die junge Generation häufiger ausgelöst, als für die alte. „Überlebt“ ein junges Objekt genug Zyklen, wird es in den Bereich der alten Objekte verschoben.

Je nach dem mit welcher Technik die verschiedenen Generationen verwaltet werden, kann ein generationelles Verfahren verschiedene Vorteile bringen. Wird ein replizierender Speicherbereiniger verwendet, entfällt der Aufwand, der für das Kopieren langlebiger Objekte notwendig ist. Die Fragmentierungstoleranz bleibt erhalten. Wird ein „Markieren und Auskehren“-Verfahren verwendet, kann dessen Fragmentierungsverhalten verbessert werden, indem langlebige Objekte, die den Speicher eventuell fragmentieren, aus der jungen Generation entfernt werden.

2.4 Kombination der genannten Speicherverwaltungstechniken

Die Schwächen der fragmentierten Allokation in Bezug auf Vektoren können überwunden werden, indem sie mit einer replizierenden Speicherbereinigung kombiniert wird [11].

2.4.1 Rückgrate

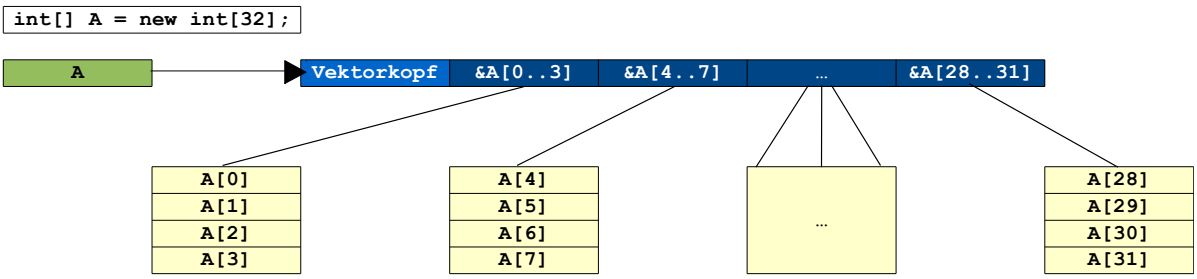

Abbildung 2.12: Verwaltung eines fragmentierten Vektors durch ein Rückgrat

Statt fragmentierte Vektoren über eine Baumstruktur aus Fragmenten zu verwalten, wird eine sogenanntes Rückgrat (engl. „spine“ [11]) verwendet. Ein Rückgrat ist ein zusammenhängender Vektor von Zeigern, die auf die Nutzdatenfragmente des verwalteten Vektors zeigen (siehe Abb. 2.12). Im Gegensatz zur Baumstruktur, die - je nach Tiefe des Baums - mehrere zusätzliche Indirektionen mit sich bringt, führt ein Rückgrat nur eine zusätzliche Indirektion ein. Die Zugriffszeit auf einen Vektor ist damit asymptotisch konstant.

2.4.2 Rückgratspeicher

Da die Größe der Rückgrate variiert, werden sie von einem replizierenden Speicherbereiniger verwaltet. Die im Rückgrat gespeicherten Zeiger sind nach der Initialisierung konstant, da die bedeuteten Fragmente nicht verschoben werden. Da sie für die Anwendung transparent sind, nimmt auch diese keine Änderungen an den Zeigern vor. Deshalb

muss beim Umkopieren der Rückgrate nicht auf die Konsistenz ihrer Werte geachtet werden.

Die Halde unterteilt sich in Rückgrat- und Fragmentspeicher. Im ungünstigsten Fall wird jedes Fragment von einem Rückgrat verwaltet. In diesem Fall muss in beiden Hälften des Rückgratspeichers Platz für einen Zeiger pro Fragment und einigen Metadaten geboten werden. Je nach Fragmentgröße kann ein Rückgratspeicher benötigt werden, der mehr als halb so groß ist, als der Fragmentspeicher. Kann der Anwendungsentwickler die Rate abschätzen, in der Vektoren angelegt werden, kann die Größe des Rückgratspeichers vermindert werden. Diese Rate kann wie die Allokationsrate bestimmt werden.

2.4.3 Generationeller Rückgratspeicher

Eine weitere Möglichkeit den Platzbedarf zu verringern ist die Verwendung einer Art generationellen Speicherbereinigung. In einigen Anwendungen werden die meisten Vektoren bereits in der ersten Bereinigungsphase gelöscht. In diesem Fall kann ein replizierender generationeller Speicherbereiniger benutzt werden, der ein Rückgrat, das eine einzige Bereinigungsphase überlebt hat, in die alte Generation verschiebt. Auf diese Weise kann in der jungen Generation die Hälfte des Speichers eingespart werden, da Rückgrate sofort altern. Während der Bereinigungsphase werden Speicheranforderungen von der inaktiven Hälfte der alten Generation bedient. Dies muss geschehen, da in der jungen Generation keine neuen Objekte angelegt werden können, während sie bereinigt wird.

Dieses Verfahren bürdet der Anwendung jedoch zusätzliche Einschränkungen auf. In der alten Generation muss Platz für alle Rückgrate geschaffen werden, die eine Bereinigungsphase überleben können (siehe Abschnitt 2.2.2). Zusätzlich muss der maximale Rückgratspeicher, der während der Bereinigungsphase angelegt werden kann, bestimmt werden. Diese Rate kann wie die Allokationsrate bestimmt werden.

2.4.4 Fragmentspeicher

Zur Verwaltung des Fragmentspeichers orientiert sich am IdleRoundRobin-Speicherbereiniger. Abgesehen von der Fragmentierungstoleranz erfüllt dieser bereits alle Kriterien, die an einen echtzeitfähigen Speicherbereiniger gestellt werden. Die kleinste anforderbare Einheit ist ein Fragment.

3 Implementierung

In diesem Kapitel wird die Umsetzung des fragmentierungstoleranten Speicherbereinigers beschrieben. Dieser trägt die Bezeichnung „FragGC“.

3.1 Das Objektformat

Das Objektformat beschreibt, wie Objekte im Speicher abgelegt werden. In Java besitzt jede Instanz einen Objektkopf, der Metadaten enthält. Das Objektformat beschreibt neben dem Objektkopf die Position und Reihenfolge, in der Instanzfelder abgelegt werden. Hierbei unterscheidet man Referenzfelder, die auf andere Objekte zeigen, und Wertfelder, wie Felder des Typs „int“ oder „char“, die primitive Daten enthalten. Da Speicherbereiniger Objekte nach Referenzen durchsuchen, ist es wichtig, dass diese Felder im Objekt leicht zu finden sind.

3.1.1 Der Objektkopf

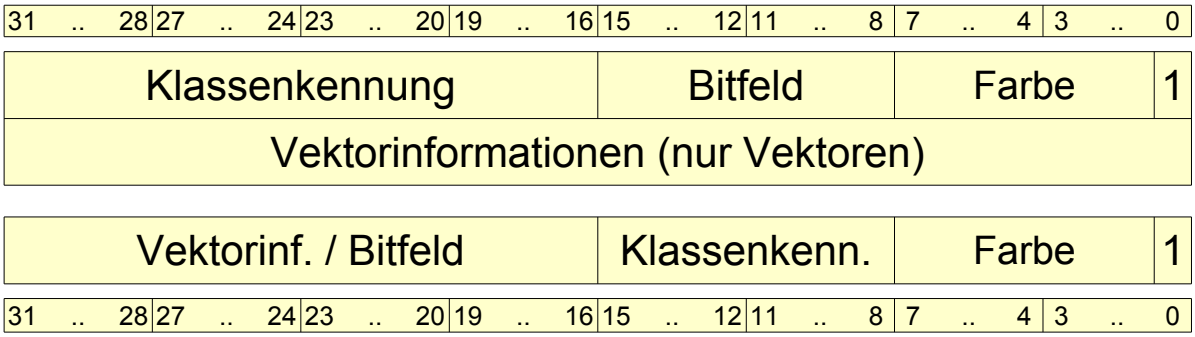

Abbildung 3.1: Objektköpfe auf einem 32-Bit-System

Der Objektkopf enthält vor allem Laufzeittypinformationen und Daten des Speicherbereinigers. Damit diese Informationen ohne weiteres Wissen verfügbar sind, zeigen Referenzen immer auf den Objektkopf der Instanz. Abbildung 3.1 zeigt den typischen Aufbau eines Objektkopfes auf einem 32-Bit-System (oben). Auf Wunsch können kompakte Objektköpfe (unten) erzeugt werden. Diese sind in der Anzahl der möglichen Klassen (8 Bit) und der Vektorlänge (2^{16} Elemente) beschränkt. Im Folgenden werden die einzelnen Bestandteile beschrieben.

Klassenkennung Die Klassenkennung repräsentiert den Typ der Instanz. Es handelt sich um eine 8 oder 16 Bit breite Ganzzahl. Anhand dieses Index' können weitere Informationen in einer Tabelle (genannt „ClassStore“) nachgeschlagen werden. Diese Tabelle kann zum Beispiel Informationen über Größe, Anzahl der Referenzen und Schnittstellen der Klasse enthalten.

Farbmarkierung Das niederwertigste Byte des Objektkopfes enthält Informationen der Speicherbereinigung.

Das niederwertigste Bit ist immer gesetzt, um in einem bidirektionalen Format (Abschnitt 3.1.2) den Objektkopf von Referenzen unterscheiden zu können. Da Objekte mindestens auf gerade Adressen ausgerichtet werden, ist das niederwertigstes Bit einer Referenz immer gelöscht.

Die restlichen sieben Bit enthalten die Farbmarkierung. Diese markiert die Objekte, welche bereits vom Speicherbereiniger besucht wurden, und damit erreichbar sind. Es stehen zwei Farben zur Verfügung, deren numerischer Wert 0 und 1 beträgt. Um vor der Markierungsphase nicht alle Objekte als unerreichbar markieren zu müssen, tauschen diese Farben nach jeder Phase ihre Rolle. Zusätzlich wurde eine dritte Farbe definiert, die anzeigt, dass es sich um ein schreibgeschütztes Objekt handelt. Diese Objekte können nicht freigegeben werden und müssen nicht nach Referenzen durchsucht werden.

Bitfeld Das Bitfeld entsteht auf 32-Bit-Architekturen dadurch, dass die 16 Bit breite Klassenkennung, auf gerade Adressen ausgerichtet wird. Das Byte könnte zukünftig für die Speicherung von „boolean“ Feldern genutzt werden. Bisher werden zwei Bit zur Paritätskodierung genutzt. Diese beiden Bits sollen transiente Fehler beim Lesen der Klassenkennung und der Vektorlänge erkennbar machen [13].

Vektorinformationen Vektoren enthalten zusätzlich zum regulären Objektkopf Informationen über die Anzahl ihrer Elemente. Je nach Konfiguration handelt es sich um eine 16 bis 32 Bit breite Ganzzahl. Vektoren, die Referenzen enthalten, speichern zusätzlich die Klassenkennung der Elemente (siehe Abschnitt 3.1.4).

3.1.2 Das bidirektionale Objektformat

Die Referenzfelder eines Objekts müssen für die Speicherbereinigung leicht zu finden sein. Es bietet sich daher an, die Referenzen zusammenhängend abzulegen.

Probleme treten bei der Vererbung von Klassen auf. Besitzen beiden Klassen Referenzen und primitive Daten, entstehen zwei Inseln mit Referenzfeldern. Abbildung 3.2 („unidirektional“) verdeutlicht dies. Der Speicherbereiniger muss hier zuerst die Referenzen von B (also r2) sammeln. Danach schlägt er die Typinformationen der Basisklasse A nach und sammelt deren Referenzen (r1 und r2). Zuletzt erkennt er, dass A lediglich von „Object“ abgeleitet wurde, das keine Referenzfelder besitzt, und beendet die Suche.

```
1  class A {
2     Object r0, r1;
3     int i0;
4  };
5  class B extends A {
6     Object r2;
7     int i1, i2;
8  };
```

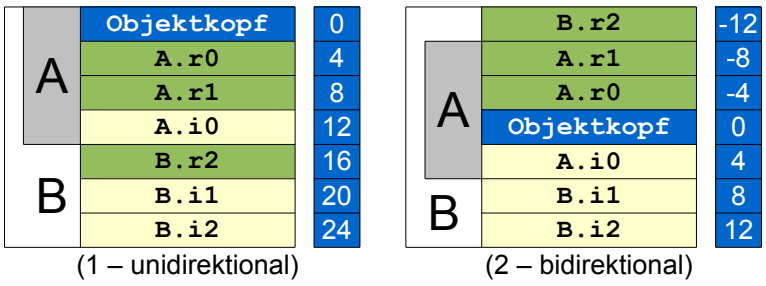

Abbildung 3.2: Unidirektionales und bidirektionales Objektformat

Der Speicherbereiniger führt also eine Schleife aus, um sich durch die Klassenhierarchie zu arbeiten. Dazu sind bestimmte Laufzeittypinformationen nötig. So muss die Kennung der Basisklasse nachzuschlagen sein. Außerdem muss für jede Klasse der Anfang und das Ende der „Referenzinsel“ gespeichert werden.

Um das Durchsuchen der Klassenhierarchie zu vermeiden, lehnt sich KESOs Objektformat an das der SableVM [3] an. Hierbei werden Referenzfelder vor dem Objektkopf abgelegt, während primitive Daten weiterhin hinter dem Objektkopf angehängt werden. Die Referenzfelder einer abgeleiteten Klasse, werden vor denen der Basisklasse abgelegt. Analog werden primitive Felder nach denen der Basisklasse angehängt. Dadurch wächst das Objekt sowohl in negativer Richtung (relativ zur Adresse des Objektkopfes) als auch in positiver Richtung, weshalb dieses Objektformat bidirektional genannt wird.

In Abbildung 3.2 („bidirektional“) ist zu erkennen, dass nun alle Referenzen zusammenhängend platziert werden. Der Speicherbereiniger muss nur noch die Anzahl der Referenzen des jeweiligen Typs nachschlagen, und kann diese dann in einer Schleife einsammeln. Die Kennung der Basisklasse und das Ende der Referenzen muss nicht in der „ClassStore“-Tabelle gespeichert werden. Das Ende der Referenzen wird durch die Referenz auf den Objektkopf markiert. In Situationen, in denen diese Referenz nicht zur Verfügung steht, wird der Objektkopf durch sein niederwertigstes Bit gekennzeichnet, beispielsweise wenn „Finalizer“ zu löschender Objekte aufgerufen werden sollen. Das niederwertigste Bit einer Referenz ist immer null, da Objekte auf Adressen ausgerichtet werden, die durch vier teilbar sind. Das niederwertigste Bit des Objektkopfes ist immer gesetzt, um so das Ende der Referenzen zu markieren.

Der blaue Balken, der sich in Abbildung 3.2 rechts neben den Objekten befindet, zeigt die relative Position der Felder zum Objektkopf an. Die Referenz auf ein Objekt zeigt immer auf dessen Objektkopf. Die relative Position eines bestimmten Feldes zum Objektkopf ist noch immer konstant und kann zur Übersetzungszeit berechnet werden. Dadurch entsteht kein zusätzlicher Aufwand bei Feldzugriffen.

3.1.3 Das fragmentierte bidirektionale Objektformat

KESO legt Objekte bisher zusammenhängend an. Um Objekte fragmentiert anlegen zu können, muss ein Objektformat entworfen werden, das dies unterstützt. Hierzu werden Objekte in Fragmente fester Größe unterteilt (siehe Abschnitt 2.3.1). Um Referenzfelder leichter finden zu können, werden die einzelnen Fragmente nach dem bidirektionalen Objektformat aufgebaut.

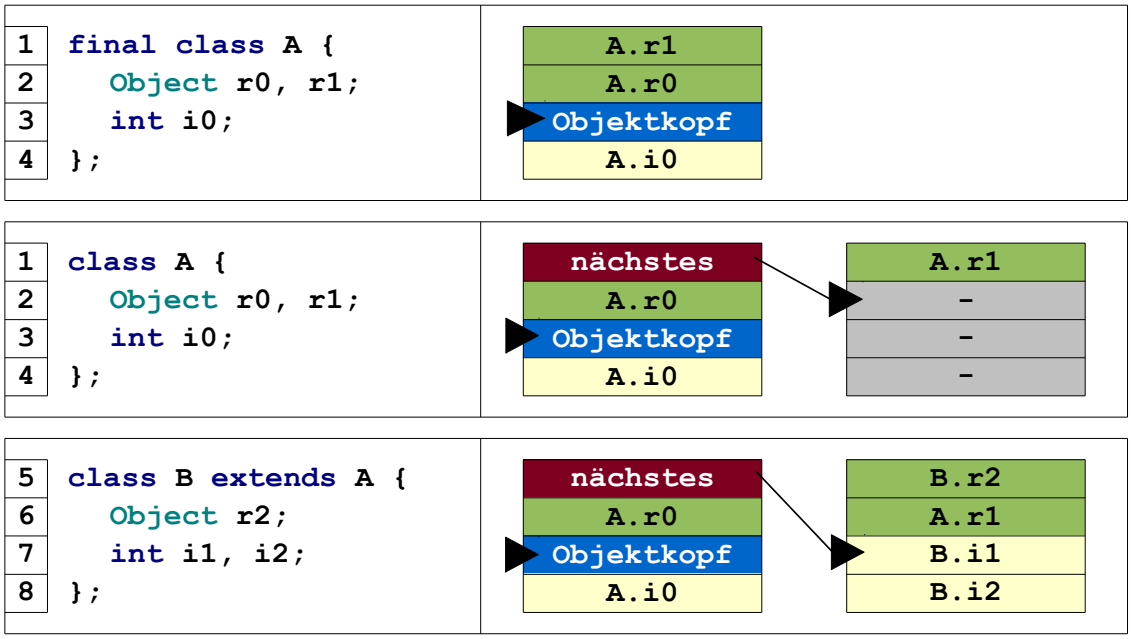

Abbildung 3.3: Das fragmentierte bidirektionale Objektformat

Abbildung 3.3 stellt Java-Klassen und die daraus gebildeten fragmentierten Objekte gegenüber. Es wird eine Fragmentgröße von 16 Byte auf einem 32-Bit-System angenommen.

Die erste Gegenüberstellung zeigt, dass Instanzen der Klasse A, die vollständig in ein Fragment passen, das gleiche Format besitzen, wie unter Verwendung des bidirektionalen Objektformats. Hierbei ist zu beachten, dass Klasse A als „final“ markiert wurde und somit keine weiteren Klassen von A abgeleitet werden können. Da KESO eine statische JVM ist, kann zur Übersetzungszeit ermittelt werden, ob eine Klasse von A abgeleitet wird, die zusätzlichen Speicherplatz für Instanzfelder benötigt. Die Markierung mit „final“ ist also nicht nötig.

In der zweiten Darstellung ist A nicht mehr „final“. Da die Klasse B von A abgeleitete wird, benötigt A ein zweites Fragment und einen Zeiger auf dieses. Der Zeiger („nächstes“) auf das zweite Fragment zeigt aber nicht auf dessen Anfang, sondern hinter das letzte Referenzfeld des Fragmentes. An dieser Stelle würde sich der Objektkopf

befinden, wäre das zweite Fragment ein unabhängiges Objekt.

In der letzten Gegenüberstellung wird eine Instanz der Klasse B gezeigt. Ihre Referenzfelder wurden wieder von oben in das Fragment eingefügt, die primitiven Felder von unten. Der Verkettungszeiger „nächstes“ zeigt wieder hinter die letzte Referenz. Da keine Klasse von B erbt, wird kein Zeiger auf ein drittes Fragment benötigt.

Der Zugriff auf ein bestimmtes Instanzfeld ist noch immer von konstanter Komplexität. Jedoch werden für einige Felder zusätzliche Indirektionen nötig. Um auf das Feld $A.r0$ zuzugreifen, muss – wie beim bidirektionalen Objektformat – nur der Wert 4 von der Referenz auf das Objekt abgezogen werden (siehe Abb. 3.2 „bidirektional“). Feld $B.r2$ befindet sich auf dem zweiten Fragment. Um auf das Feld zuzugreifen, muss zuerst auf den Verkettungszeiger zugegriffen werden. Anschließend wird vom Wert des Zeigers 8 abgezogen, um die Adresse von $B.r2$ zu erhalten. Das liegt daran, dass sich das Feld acht Byte vor der Position befindet, auf die der Verkettungszeiger zeigt (Abb. 3.3). Da Java-Objekte vergleichsweise klein ausfallen, ist die Anzahl der zusätzlichen Indirektionen vertretbar. So besteht ein reguläres Objekt in der Testanwendung CDj aus maximal drei Fragmenten (siehe Abschnitt 2.2).

Durchsuchen der fragmentierten Objekte

KESO verwendet das bidirektionale Objektformat um Objekte effizient nach Referenzen durchsuchen zu können. Um von der Referenz auf ein Objekt auf dessen Anfang schließen zu können, wird in den Laufzeittypinformationen gespeichert, wie viele Referenzen eine bestimmte Klasse besitzt.

Beim fragmentierten Objektformat muss die Anzahl der Fragmente bestimmt werden können, sowie die Anzahl der Referenzen in jedem Fragment. Die Anzahl der Fragmente wird in den Laufzeittypinformationen gespeichert. Die Anzahl der Referenzen eines jeden Fragmentes in den Laufzeittypinformationen unterzubringen, würde deren Platzbedarf – ähnlich dem unidirektionalen Objektformat – erhöhen. Stattdessen wird die Anfangsadresse des Fragmentspeichers so ausgerichtet (engl. „alignment“), dass sie durch die Fragmentgröße teilbar ist. Die Anfangsadresse jedes Fragments ist nun durch die Fragmentgröße teilbar. Da es sich bei der Größe um eine Zweierpotenz handelt, kann die Anfangsadresse eines Fragmentes, in das der Zeiger „nächstes“ zeigt, berechnet werden, indem die niederwertigen Bits des Zeigers entsprechend der Fragmentgröße genullt werden.

Leider existiert ein Ausnahmefall: Besteht ein Fragment nur aus Referenzfeldern, zeigt „nächstes“ hinter das eigentliche Fragment. Wird der Zeiger nun ausgerichtet, zeigt er noch immer auf die gleiche Adresse. Da die Referenzen zwischen dem unausgerichteten und dem ausgerichteten Zeiger vermutet werden, werden keine Referenzen erkannt. Daher wird in den Laufzeittypinformationen für jedes Fragment ein Bit eingebracht, das diesen Spezialfall anzeigt. Der Wert dieses Bits wird vom Zeiger auf das Fragment subtrahiert, um zum eigentlichen Fragment zu gelangen. So werden alle Felder des Fragments vom Anfang bis zum Ende als Referenzen erkannt.

Fragmentgröße	Maximale Nutzdatengröße [Byte]
16	72
32	168
64	360

Abbildung 3.4: Maximale Gesamtgröße der in einer Klasse enthaltenen Instanzfelder

Die Größe dieser Laufzeittypinformationen hängt von der maximalen Anzahl der Fragmente eines Objekts ab. Bei einer Fragmentgröße von 16 Byte treten bei der Testanwendung *CDj* Objekte mit maximal drei Fragmenten auf. Bei einer Fragmentgröße von 32 Byte passt jedes reguläre Objekt in ein Fragment. Die gegenwärtige Implementierung beschränkt die Größe eines Objektes auf sechs Fragmente. Aufgrund der Indirektionen, die bei einem Feldzugriff aufgelöst werden, sollte ein Objekt ohnehin nicht aus mehr Fragmenten bestehen. In Abbildung 3.4 wird die maximale Größe der Nutzdaten beschrieben, die ein reguläres Objekt enthalten kann. Des Weiteren muss für das erste Fragment kein Bit gespeichert werden, da der Objektkopf verhindert, dass dieses nur aus Referenzfeldern besteht. Daher können diese Informationen meist in einem Byte gespeichert werden. Drei Bit geben die Anzahl der Fragmente an. Die restlichen fünf Bit markieren Fragmente, die nur Referenzfelder enthalten. Je nach Konfiguration kann dieses Byte beim Anlegen eines Objektes in das Bitfeld des Objektkopfes geschrieben werden, statt im „ClassStore“ gespeichert zu werden.

Zukünftig könnten größere Objekte unter Verwendung von Rückgraten implementiert werden. Wird ab einem bestimmten Fragment über ein Rückgrat auf die weiteren Fragmente zugegriffen, bleibt die Anzahl der Indirektionen beschränkt.

```
static void fgc_scan_regular_object(object_pointer obj) {
      travinfo_t cinfo = getTraversalInfo(obj);
      unsigned   nfrags = getFragmentCount(cinfo);
      pointer_t  next = obj;
      for(unsigned i = 0; i != nfrags; ++i) {
            HANDLE_FIRST_AND_LAST_FRAGMENT();
            fragment_t *begin = next & ~(FRAGMENT_SIZE - 1);
            begin -= getReferencesOnlyBit(cinfo, i);
            pushReferencesBetween(begin, next);
            next = getNextPointer(begin);
      }
}
```

Abbildung 3.5: Durchsuchen eines fragmentierten Objektes (vereinfacht)

Codebeispiel 3.5 fasst das Verfahren vereinfacht zusammen. Anfangs werden die Traversierungsinformationen – unter Verwendung der im Objektkopf gespeicherten Klassenkennung – geladen. Diese beinhalten die Anzahl der Fragmente. Anschließend wird über die Fragmente iteriert. Hierzu wird der vereinfachte Zeiger *next* (vgl. „nächstes“) zuerst

auf das erste Fragment gesetzt. In der Schleife wird der Anfang des Fragments *begin* ermittelt, indem *next* an der Fragmentgröße ausgerichtet wird. Die Fragmentgröße ist eine systemweite Konstante (siehe Abschnitt 3.1.5). Falls es sich um ein Fragment handelt, das nur aus Referenzen besteht, wird *begin* entsprechend korrigiert. Jetzt können die Referenzen, die von *begin* und *next* markiert werden, auf den Arbeitsstapel des Speicherbereinigers gelegt werden. Anschließend wird *next* auf das nächste Fragment weitergerückt. „HANDLE_FIRST_AND_LAST_FRAGMENT“ deutet an, dass für das erste Fragment kein „ReferencesOnlyBit“ berechnet wird. Genauso muss beim letzten Fragment, bei “pushReferences“ beachtet werden, dass dieses keinen Zeiger auf das nächste Fragment besitzt.

Aufbau der Fragmente

Im Folgenden wird genauer beschrieben, wie die Fragmente eines Objektes aufgebaut werden. Dabei muss darauf geachtet werden, dass die erzeugten C-Strukturen die Größe eines Fragmentes nicht übersteigen. Zusätzlich sollen alle Felder einer Klasse in möglichst wenigen Fragmenten untergebracht werden.

Das C-Typsystem Bisher wurde in KESOs Übersetzer JINO nicht auf die Größe der erzeugen Java-Klassen geachtet. Nun müssen Fragmente fester Größe mit den Instanzfeldern gefüllt werden. Dazu wurde eine Bibliothek zur Erstellung von C-Strukturen geschrieben, die für jeden möglichen Feldtyp Größe und Ausrichtung (engl. „alignment“) ermittelt. Diese Bibliothek geht davon aus, dass Felder auf Adressen ausgerichtet werden, die durch ihre Größe teilbar sind. Als maximales Alignment wird das der Referenz angenommen, da das bidirektionale Objektformat ohne diese Bedingung gegenwärtig nicht funktioniert [15]. Abbildung 3.6 zeigt, warum das bidirektionale Objektlayout kein

```
1  class A {
2    long a;
3  };
4  class B extends A {
5    Object b;
6  };
```

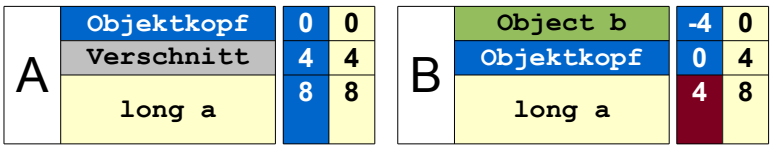

Abbildung 3.6: Fehlerhafte Platzierung eines Feldes mit zu hohem Alignment

Alignment zulässt, das das von Referenzen übersteigt. Rechts der Darstellung der Objekte wird jeweils einmal die Position relativ zum Objektkopf und einmal die Position in der generierten C-Struktur angezeigt. Im Objekt vom Typ A tritt vor Feld a Verschnitt auf, da es in der C-Struktur auf durch acht teilbare Adressen ausgerichtet wird. Die Position relativ zum Objektkopf ist 8. Im Objekt vom Typ B tritt in der C-Struktur kein Verschnitt auf. Die Position von Feld a ist nun 4. Würde über eine Referenz mit

statischem Typ A auf eine Instanz der Klasse B zugegriffen, würde auf Position 8 statt 4 zugegriffen werden.

Falls die angenommenen Eigenschaften des Typsystems zu falschen Ergebnissen führen, werden vom C-Übersetzer Fehlermeldungen ausgegeben, die von sogenannten „static asserts“ erzeugt werden (siehe unten).

Schrittweises Erstellen der Fragmente Ausgestattet mit diesen Informationen, wird der in Abbildung 3.7 beschriebene Algorithmus auf jede Klasse angewandt. Hierbei wurde vor allem darauf geachtet, den Verschnitt innerhalb der Fragmente möglichst gering zu halten. So werden die Fragmente, die noch nicht die maximale Größe erreicht haben, gesammelt (Z. 3, 33), um mit kleinen Feldern aufgefüllt zu werden. Des Weiteren werden primitive Felder nach ihrem Alignment sortiert (Z. 4), um Verschnitt zwischen den Feldern zu umgehen. Die sortierten, primitiven Felder werden dann mit den Referenzfeldern vermischt. Gegenwärtig werden die beiden Sorten abwechselnd eingefügt. Es wäre aber auch denkbar Felder zusätzlich nach der Häufigkeit der Zugriffe zu sortieren. „Heiße“ Felder würden so in ein vorderes, also schnelleres, Fragment kommen. Beim Einfügen der Felder (Z. 11) wird darauf geachtet, dass kein Fragment die Maximalgröße überschreitet. Würde diese überschritten werden, oder wird Speicherplatz für weitere Felder benötigt, wird ein neues Fragment erstellt und in das alte ein Verkettungszeiger eingefügt. Durch Prüfungen zur Übersetzungszeit (engl. „static asserts“) versichert der C-Übersetzer, dass kein Fragment die maximale Fragmentgröße überschreitet. Nach dem Einfügen der Felder muss der Versatz der Verkettungszeiger zum Anfang der Fragmente korrigiert werden. Abbildung 3.8 zeigt die C-Strukturen, die aus der Java-Klasse „LinkedList“ generiert werden. Die Fragmentgröße beträgt 16 Byte. In den Kommentaren hinter den Feldern wird angegeben, aus welcher Struktur das Feld ursprünglich stammt, welche Position es in der gegenwärtigen Struktur hat („offset“), die Größe („size“) und das Alignment. Für den Zeiger „next“ wird angedeutet, dass er auf eine Instanz von „c41_LinkedList_frag1_t“ zeigt, jedoch um die Größe von zwei Referenzen verschoben, da er hinter das Feld „c41f2_first“ zeigt. Die darunter definierten Makros beschreiben, wie eine Referenz, die bekanntlich auf den Objektkopf zeigt, in einen Zeiger auf die entsprechende C-Struktur umgewandelt wird. Um zu Fragment 0 zu gelangen, muss die Referenz den Zeiger „next“ überspringen, weswegen sie um 1 vermindert wird. Um zu Fragment 1 zu gelangen, wird der Zeiger „next“ um 2 vermindert. In der Datei „global.c“ wird sichergestellt, dass kein Fragment die Maximalgröße überschreitet.

Nichtallozierbare Objekte JINO webt in einige Klassen zusätzliche Felder ein („Rawfields“). Diese Felder können Typen besitzen, die von der Implementierung des Betriebssystems abhängen. Daher kann ihre Größe und Ausrichtung nicht bestimmt werden. Dies betrifft zum Beispiel die Klassen „Thread“, „Alarm“ oder „Resource“. Hierbei handelt es sich um Objekte, die ohnehin statisch konfiguriert werden. So ist es beispielsweise nicht möglich zur Laufzeit neue Programmfäden zu erzeugen. Die Fragmente der Klasse werden

```
 1  Erstelle alle Fragmente einer Klasse C:
 2      Kopiere alle Fragmente der Basisklasse
 3      Sammle alle „unfertigen Fragmente“
 4      Sortiere alle primitiven Felder, die von C hinzugefügt werden,
 5          nach ihrem Alignment
 6      Vermische die sortierten primitiven Felder mit den Referenzfeldern
 7      Füge diese Felder nacheinander in die Fragmente ein (Zeile 11)
 8      Korrigiere die Zeiger, die die Fragmente verbinden
 9      Fertig.
10
11  Füge Feld F zu den Fragmenten der Klasse C hinzu:
12      Falls F primitiv ist:
13          Versuche F an eines der „unfertigen Fragmente“ anzuhängen
14          Falls erfolgreich: Fertig.
15      (Referenzfelder werden Fragmenten von vorne hinzugefügt,
16          primitive Felder von hinten)
17      Überprüfe ob F in das zuletzt erstellte Fragment Z der Klasse C
18          eingefügt werden kann
19      Falls Z danach für einen Zeiger keinen Platz mehr bieten würde:
20          Falls Z die maximale Fragmentgröße überschreiten würde:
21              Hänge ein Fragment an Klasse C an (Zeile 28)
22          Falls Zs Größe der maximale Fragmentgröße entspräche:
23              Falls C oder eine Unterklasse von C weitere Felder besitzt:
24                  Hänge ein Fragment an Klasse C an (Zeile 28)
25      Füge das Feld F dem zuletzt erstellten Fragment hinzu
26      Fertig.
27
28  Hänge ein Fragment an Klasse C an:
29      Z sei das zuletzt erstellt Fragment der Klasse C
30      Füge einen Verkettungszeiger in Z ein
31      Falls Zs Größe nicht der Maximalgröße entspricht:
32          Zähle Z zu den „unfertigen Fragmenten“
33      Erstelle ein neues Fragment für Klasse C
34      Fertig.
```

Abbildung 3.7: Erstellen der Fragmente einer Klasse

zunächst so generiert, als ob die Rawfields nicht existieren würden. Anschließend werden die Rawfields an das erste Fragment als primitive Daten angehängt. Dadurch kann die maximale Fragmentgröße überschritten werden. Da diese Objekte nicht dynamisch angelegt werden können, müssen sie ohnehin nicht in die Fragmente der Halde passen. Die

```
/* In c41_LinkedList.h */
typedef struct {
      object_pointer  c41f3_last;  /*from "c41_LinkedList_frag1_t" (o:0,s:4,a:4)*/
      object_pointer  c41f2_first; /*from "c41_LinkedList_frag1_t" (o:4,s:4,a:4)*/
} c41_LinkedList_frag1_t KESO_ALIGN(16);
typedef struct {
      object_pointer* next /*c41_LinkedList_frag1_t+2*/;
      unsigned char gcinfo;   /*from "object_t" (o:4,s:1,a:1)*/
      unsigned char bitfld;   /*from "object_t" (o:5,s:1,a:1)*/
      class_id_t    class_id; /*from "object_t" (o:6,s:2,a:2)*/
      jint    c38f1_modCount; /*from "c38_AbstractList_t" (o:8,s:4,a:4)*/
      jint    c41f4_size;     /*from "c41_LinkedList_t" (o:12,s:4,a:4)*/
} c41_LinkedList_t KESO_ALIGN(16);

#define C41_LINKEDLIST_FRAG0(_obj_)\
      ((c41_LinkedList_t*)((object_pointer*)(_obj_)-1))
#define C41_LINKEDLIST_FRAG1(_obj_)\
      (((c41_LinkedList_frag1_t*)((C41_LINKEDLIST_FRAG0(_obj_)->next) - 2)))

/* In global.c */
KESO_STATIC_ASSERT(sizeof(c41_LinkedList_t) <= 16, c41_LinkedList_t_exceeds_fragment_size);
KESO_STATIC_ASSERT(sizeof(c41_LinkedList_frag1_t) <= 16, c41_LinkedList_frag1_t...);
```

Abbildung 3.8: Generierte C-Strukturen der Klasse LinkedList

Adressausrichtung der Fragmente muss erhalten bleiben, damit der Speicherbereiniger die Referenzfelder erkennen kann.

3.1.4 Das Vektorformat

Zusammenhängende Vektoren

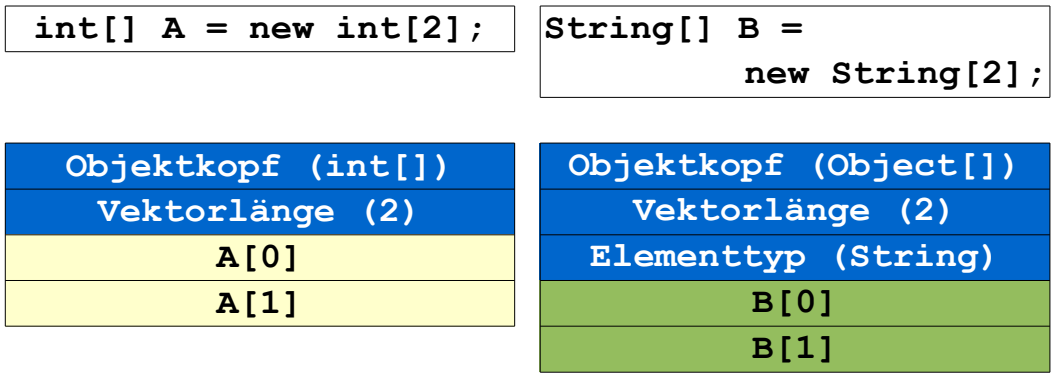

Abbildung 3.9: Zusammenhängende Vektoren

Bisher kennt KESO nur zusammenhängende Vektoren. In Abbildung 3.9 wird links ein Vektor dargestellt, der primitive Daten enthält, rechts einer, der Referenzen enthält. Vektoren sind Objekte, daher beginnen sie mit einem Objektkopf. Auf den Objektkopf folgt die Vektorlänge, die angibt, wie viele Elemente sich im Vektor befinden. Ein

Referenzvektor enthält zusätzlich die Klassenkennung des Typs, auf den die gespeicherten Referenzen zeigen. Der linke Vektor benötigt dies nicht, da jede Vektorklasse mit primitivem Elementtyp über eine eigene Klassenkennung verfügt. Gleich nach den Metainformationen werden die Elemente des Vektors abgelegt.

Fragmentierte Vektoren

Zur fragmentierten Allokation wird ein neues Vektorformat benötigt. Wie in der Analyse (2.4) beschrieben wurde, werden fragmentierte Vektoren von einem Rückgrat verwaltet. Anders als in Abbildung 2.12 zeigt eine Referenz jedoch nicht direkt auf das Rückgrat. Würde ein Rückgrat verschoben werden, müssten alle Referenzen, die auf das verschobene Rückgrat zeigen, angepasst werden. Statt dessen wird ein unbewegliches Wächterelement im Fragmentspeicher angelegt, das auf das Rückgrat zeigt. Wird das Rückgrat verschoben, muss nur der Zeiger im Wächterfragment angepasst werden.

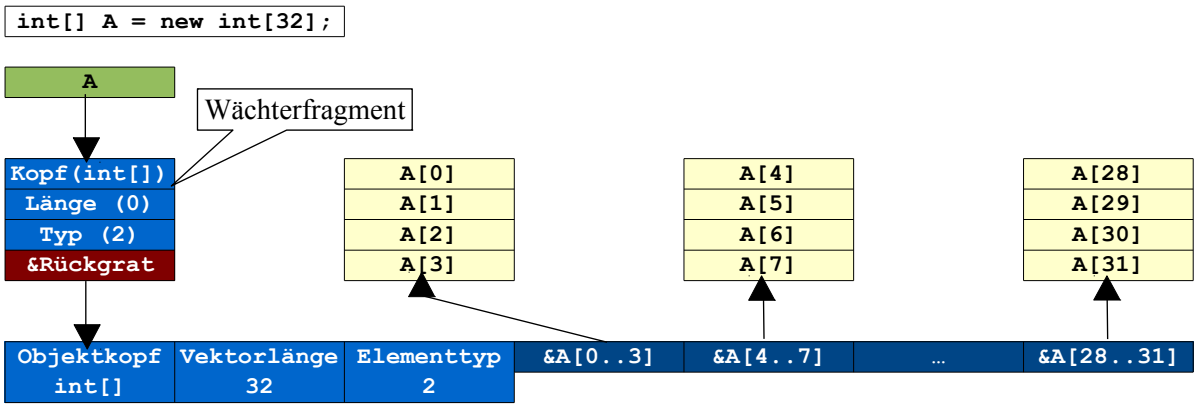

Abbildung 3.10: Fragmentierter Vektor mit Wächterfragment

Abbildung 3.10 zeigt einen fragmentierten Vektor. Es fällt auf, dass sowohl im Wächterfragment als auch im Rückgrat eine Längenangabe zu finden ist. Dies liegt daran, dass es noch immer möglich ist, zusammenhängende Vektoren anzulegen. Steht im Wächterelement die Länge 0, handelt es sich um einen fragmentierten Vektor. Sonst handelt es sich um einen zusammenhängenden Vektor.

Vektoren, die statisch und mit fester Größe angelegt werden, können so zusammenhängend abgelegt werden. Dadurch wird der Speicher für das Rückgrat gespart und Zugriffe laufen schneller ab. Durchschnittlich sinkt auch der Aufwand für Zugriffe auf Vektoren die sich auf der Halde befinden, da diese unter Umständen auch zusammenhängend angelegt werden können. Auch Rückgratspeicher wird im Durchschnitt gespart. Im ungünstigsten Fall können jedoch keine Vektoren zusammenhängend auf dem Haldenspeicher angelegt werden. Die Ausführungsgeschwindigkeit und der gesparte Speicher sinken mit dem Grad der Fragmentierung.

Eingebettete Rückgrate

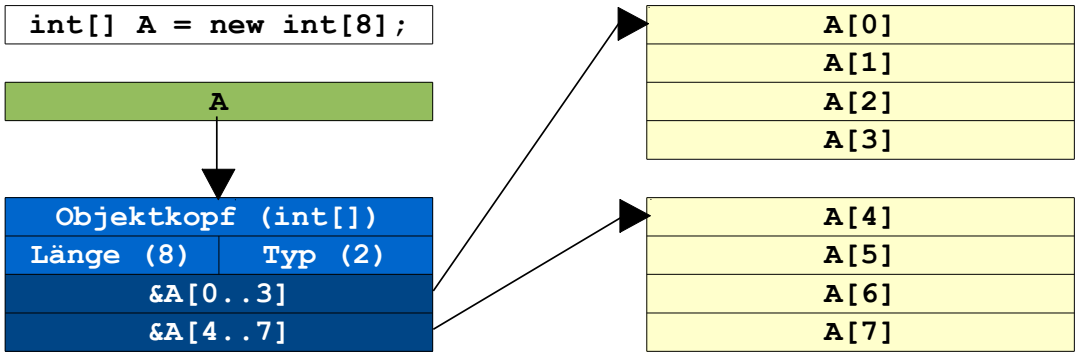

Abbildung 3.11: Vektor mit eingebettetem Rückgrat

Alternativ wird ein Vektorformat angeboten, das zusammenhängende Vektoren nicht unterstützt. Vektoren haben immer ein Rückgrat, jedoch kann das Rückgrat für kleine Vektoren in das Wächterfragment eingebettet werden. Ein solches eingebettetes Rückgrat ist in Abbildung 3.11 zu sehen. Ob das Rückgrat eingebettet wird oder nicht, ist nur von der Länge des Vektors abhängig. Daher hat Fragmentierung keine Auswirkung auf dieses Format. Indem der Speicher des Wächterfragments genutzt wird, wird Rückgratspeicherplatz gespart. Dieses Format eignet sich auch, um die Ausführungsgeschwindigkeit nach unten hin abzuschätzen.

Abbildung 3.12 zeigt die maximale Nutzlast, die von einem eingebetteten Rückgrat verwaltet werden kann. Sind die Nutzdaten eines Vektors zu groß, um von einem eingebetteten Rückgrat verwaltet zu werden, wird – wie gehabt – ein ausgelagertes Rückgrat im Rückgratspeicher angelegt.

Bereichsprüfungen

Vor einem Vektorzugriff wird eine Bereichsprüfung durchgeführt. Hierbei wird überprüft, ob der Index des Elementes, auf das zugegriffen werden soll, kleiner ist, als die Vektorlänge. Ist dies nicht der Fall, gibt es kein gültiges Element zu diesem Index. Beim bisherigen Vektorformat wird dann eine „IndexOutOfBounds“-Ausnahmebehandlung angestoßen. Diese Prüfung kann möglicherweise durch statische Analysen eingespart werden.

Fragmentgröße	Maximale Nutzdatengröße [B]
16	32
32	192
64	896

Abbildung 3.12: Maximale Nutzdatengröße eines eingebetteten Rückgrats

Wird ein Vektorformat verwendet, dass Rückgrate unterstützt, bedeutet ein Fehlschlag der einfachen Bereichsprüfung noch nicht, dass es sich um einen ungültigen Zugriff handelt. Ist als Vektorlänge 0 eingetragen, verfügt der Vektor über ein Rückgrat. In diesem Fall muss die Vektorlänge aus dem Rückgrat gelesen werden und die Bereichsprüfung erneut durchgeführt werden. Schlägt sie nun fehl, handelt es sich tatsächlich um einen ungültigen Zugriff.

Durch die von JINO durchgeführten statischen Analysen, kann die Bereichsprüfung gegebenenfalls eingespart werden. Dies ist dann der Fall, wenn nachgewiesen werden kann, dass der Index die Länge des Vektors nicht übersteigt.

Vektorzugriffe

Wie auf einen Vektor zugegriffen wird, unterscheidet sich, je nach dem ob er ein Rückgrat besitzt oder nicht. Ob er ein Rückgrat besitzt, wird wie bei der Bereichsprüfung ermittelt. Wurde die Bereichsprüfung durch Optimierungen entfernt, muss diese Unterscheidung dennoch eingefügt werden.

Handelt es sich um einen zusammenhängenden Vektor, wird durch die übliche Zeigerarithmetik auf dessen Elemente zugegriffen. Dabei wird der Index auf die Adresse des ersten Elementes addiert.

Handelt es sich um einen fragmentierten Vektor, nimmt der Zugriff die Verzweigung über das Rückgrat.

$$\begin{aligned} \mathit{Adresse}_0 &= \text{Index} \cdot \text{ElementGröße} \\ \text{FragmentIndex} &= \mathit{Adresse}_0 \gg \log_2(\text{FragmentGröße}) \\ \text{ElementIndex} &= \mathit{Adresse}_0 \ \& \ (\text{FragmentGröße} - 1) \\ \text{FragmentAdresse} &= \text{Rückgrat[FragmentIndex]} \\ \text{ElementAdresse} &= \text{FragmentAdresse} + \text{ElementIndex} \end{aligned}$$

Hierzu wird die Adresse des Elementes berechnet, die sich ergäbe, wenn sich das erste Element an Adresse 0 befände ($\mathit{Adresse}_0$). Diese Adresse lässt sich in einen Fragmentindex und einen Elementindex aufteilen. Um den Fragmentindex zu erhalten wird $\mathit{Adresse}_0$ durch die Fragmentgröße geteilt. Die Fragmentgröße ist eine Zweierpotenz und eine systemweite Konstante. Daher kann diese Division durch eine vorzeichenlose binäre Schiebeoperation ($\gg$) dargestellt werden. Der Elementindex ist der Rest der Division. Er kann durch ein bitweises *Und* erhalten werden. Die Fragmentadresse kann im Rückgrat unter Verwendung des Fragmentindex' nachgeschlagen werden. Die Adresse des Elementes ergibt sich, indem man den Elementindex auf die Fragmentadresse addiert.

Die Adressberechnung lässt sich mit einfachen Operationen bewältigen. Im Vergleich zum zusammenhängenden Vektor, wird durch das Nachschlagen im Rückgrat nur eine zusätzliche Indirektion nötig. Daher ergibt sich eine vertretbare WCET, die geringer als bei baumartigen Vektoren (Abschnitt 2.3.1) ausfällt.

Fragmentierte Vektoren könnten schlechte Lokalitätseigenschaften besitzen. Somit können Vektorzugriffe möglicherweise nicht vom „Cache“ beschleunigt werden.

3.1.5 Einbau in KESO

Bisher kannte KESOs Übersetzer JINO nur das bidirektionale Objektformat. Um ein alternatives Objektformat zu erlauben, müssen Abstraktionen in JINO und dem erzeugten C-Code eingefügt werden. Zudem muss das Objektformat in der Konfigurationsdatei ausgewählt werden können.

Konfiguration des Objektformats

Das Objektformat ist eine systemweite Eigenschaft und wird in KESOs Konfigurationssprache daher als Attribut eines „Systems“ angegeben. Da der Code von Methoden von mehreren Domänen geteilt wird, müssen alle Domänen das gleiche Objektformat benutzen. Schließlich unterscheiden sich die Formate darin, wie auf Instanzfelder oder Vektorelemente zugegriffen wird. Auch die Fragmentgröße darf sich zwischen den Domänen nicht unterscheiden. Wird für ein System kein Objektformat angegeben, wird das bidirektionale verwendet.

```
World (Keso) {
     System (Example) {
          ObjectLayout = FragGC {
               FragSize = 16;
               MaxChainCount = 6;
               ContiguousArrays = 1;
               ExpectContiguousArrays = 0;
               LargeArrays = 1;
               MaxTriesToAllocContiguous = 4;
          }
          # ...
          # Informationen über Module, Zielplattform,
          #    Betriebssystem, Ereignisse, Domänen etc.
     }
}
```

Abbildung 3.13: Konfiguration des Objektformats

Abbildung 3.13 zeigt wie die Konfiguration des fragmentierten bidirektionalen Objektformats aussehen kann.

„FragSize“ bestimmt die Größe der Fragmente in Byte. Bei der Wahl der Fragmentgröße ist die Gesamtgröße der Instanzfelder zu beachten, die ein reguläres Objekt im

Median besitzt. Einerseits sollten die Objekte aus möglichst wenigen Fragmenten bestehen, damit ein effizienter Zugriff auf die Instanzfelder möglich ist. Andererseits sollte durch die Fragmentgröße möglichst wenig Verschnitt entstehen. Der Zugriff auf fragmentierte Vektoren läuft immer gleich ab. Größere Fragmente können jedoch die Lokalitätseigenschaften von Vektorzugriffen verbessern.

„MaxChainCount“ beschränkt die Anzahl der Fragmente aus denen ein Objekt bestehen darf. So kann dem Entwickler mitgeteilt werden, wenn eine Klasse diese Größe überschreitet. Dadurch ist eine grobe WCET-Abschätzung für Feldzugriffe im Allgemeinen möglich.

Ist „ContiguousArrays“ gesetzt, ist es möglich zusammenhängende Vektoren anzulegen (siehe Abschnitt 3.1.4). Ist dieses Attribut nicht gesetzt, können Vektoren nur fragmentiert angelegt werden. Dies ist zur WCET-Abschätzung nützlich.

„ExpectContiguousArrays“ gibt an, ob der Sprungvorhersage des Prozessors mitgeteilt werden soll, dass zusammenhängende Vektoren häufiger auftreten als fragmentierte Vektoren.

Ist „LargeArrays“ aktiviert, wird die Anzahl der Elemente eines Vektors mit einer 32 Bit breiten Ganzzahl gespeichert. Sonst wird eine 16-Bit-Ganzzahl verwendet.

„MaxTriesToAllocContiguous“ teilt mit, wie oft bei der Allokation eines Vektors versucht werden darf, diesen zusammenhängend anzulegen. Wie diese Versuche aussehen, wird in Abschnitt 3.3.2 beschrieben. Die Anzahl der Versuche könnte zukünftig ein Parameter der Vektorallokation sein, wenn der Anwendung bekannt ist, dass fragmentierte Allokation verwendet wird.

Umsetzung in JINO

Aus der obigen Beschreibung wird für JINO ein von der abstrakten Klasse „ObjectLayout“ abgeleitetes Objekt erstellt. Dieses Objekt erzeugt nach dem Muster einer abstrakten Fabrik weitere formatspezifische Objekte. Beispielsweise werden Objekte zur Erzeugung von C-Initialisierungslisten erzeugt. Zudem greift das „ObjectLayout“ über virtuelle Methoden direkt in die C-Codegenerierung ein.

So wird der Zugriff auf Instanz- oder Vektorfelder hinter Makros versteckt, die je nach Objektformat definiert werden. Auch das Anlegen regulärer Objekte wird hinter Makros versteckt, da dies je nach Format verschiedene Parameter benötigt. Die Verschiedenen Objektformate bringen noch mehr dieser Unterscheidungen mit sich. Einige werden im Folgenden aufgezählt:

- Erstellen von statischen und methodenlokalen Objekten
- Attributierung von Objekten (Adressausrichtung)
- Erstellen von mehrdimensionalen Arrays
- Erstellen konstanter Zeichenfolgen („Strings“)

- Anlegen von Laufzeittypinformationen („ClassStore“)
- Definition der C-Strukturen benutzerdefinierter Klassen
- Definition der C-Strukturen von Vektorklassen
- Definition von Bereichsprüfungen
- Errechnen der Position des Objektkopfes in C-Strukturen
- Funktionen zur Ausgabe von Zeichenfolgen („write“)
- Kopieren von Objekten für Portalaufrufe

Diese Anpassungen an das fragmentierte Objektformat laufen immer nach dem gleichen Schema ab. Beispielsweise kann die Funktion „write“ nur zusammenhängende Daten in eine Datei schreiben. Soll sie auf einen fragmentierten Vektor angewendet werden, muss jedes Fragment einzeln übergeben werden.

3.2 Die Datenstrukturen

3.2.1 Übernommene Datenstrukturen

Zur Verwaltung des Fragmentspeichers konnten die meisten Datenstrukturen des IdleRoundRobin übernommen werden. In Abschnitt 2.1 wurden bereits einige dieser Datenstrukturen beschrieben. So werden die Referenzen der Objekte, die noch durchsucht werden müssen, auf dem Arbeitsstapel abgelegt. Um die Stapelspeicher der Anwendungsfäden nach Referenzen durchsuchen zu können, werden Henderson-Listen [4] verwendet. Weitere Datenstrukturen, die zur Implementierung des Speicherbereinigers notwendig sind, sind die Bitkarte (engl. „Bitmap“) und die Domänendeskriptortabelle.

Die Bitkarte

Der Fragmentspeicher wird in Fragmente fester Größe unterteilt. Diese Fragmente sind die kleinste Einheit, die von der Halde angefordert werden können. Ihre Größe beträgt üblicherweise 16 bis 64 Byte.

Während der Markierungsphase muss sich der Speicherbereiniger merken, welche Fragmente von lebendigen Objekten belegt werden und welche Objekte freigegeben werden können. Daher wird jedes Fragment auf ein Bit auf einer Bitkarte (engl. „Bitmap“) abgebildet. Ist das zum Fragment gehörige Bit gelöscht, kann das Fragment freigegeben werden. Ist es gesetzt, wird das Fragment von einem lebendigen Objekt verwendet.

Für alle Domänen wird nur eine Bitkarte verwendet. Diese muss genügend Bits enthalten, um jedem Fragment der größten Halde im System ein Bit zuweisen zu können.

Nach jeder Bereinigungsphase müssen alle Bits der Bitkarte wieder gelöscht werden, um für die nächste Bereinigungsphase wiederverwendet werden zu können.

Die Bitkarte wird durch einen Ganzzahlvektor implementiert.

Die Domänendeskriptortabelle

Die Domänendeskriptortabelle enthält pro Domäne einen Domänendeskriptor. Dieser Deskriptor speichert Laufzeitinformationen, die vor allem für den Speicherbereiniger nützlich sind. Welche Informationen bereitgestellt werden, hängt teilweise von dem Speicherbereiniger ab, der die jeweilige Domäne verwaltet. In einem System, das das fragmentierte Objektformat verwendet, kann bisher nur der hier implementierte Bereiniger „FragGC“ auftauchen. Zukünftig könnten fragmentierte Objekte auch auf einer RestrictedDomainScope-Halde angelegt werden.

Folgende Informationen sind in jedem FragGC-Domänendeskriptor vorhanden:

- Haldengröße: Die Größe des Fragmentspeichers
- FreieSlots: Die Anzahl der unbelegten Fragmente im Fragmentspeicher
- FarbBit: Der Wert, der in dieser Domäne momentan die Farben grau bzw. schwarz (also „erreichbar“) repräsentiert
- Wurzelmenge:
  - Ein Vektor, der die statischen Referenzfelder der Domäne enthält
  - Ein Vektor, der Referenzen auf globale Objekte dieser Domäne enthält

Für den Fragmentspeicher werden folgende Informationen gespeichert:

- NeuAngeforderteSlots: Die Anzahl der Fragmente, die seit der letzten Bereinigungsphase angefordert wurden
- Anfangs und Endadresse des Fragmentspeichers
- Das Wächterelement der Freispeicherliste (siehe Abschnitt 3.2.2)

Welche Informationen für den Rückgratspeicher angelegt werden, hängt von dessen Implementierung ab (siehe Abschnitt 3.2.3).

3.2.2 Unterbrechungstolerante Freispeicherliste

Die verfügbaren Fragmente werden in einer Liste verwaltet. Es handelt sich um eine einfach verkettete Liste mit Wächterelement. Die Synchronisation der Liste orientiert sich an der Freispeicherliste des IdleRoundRobin (IRR) [14].

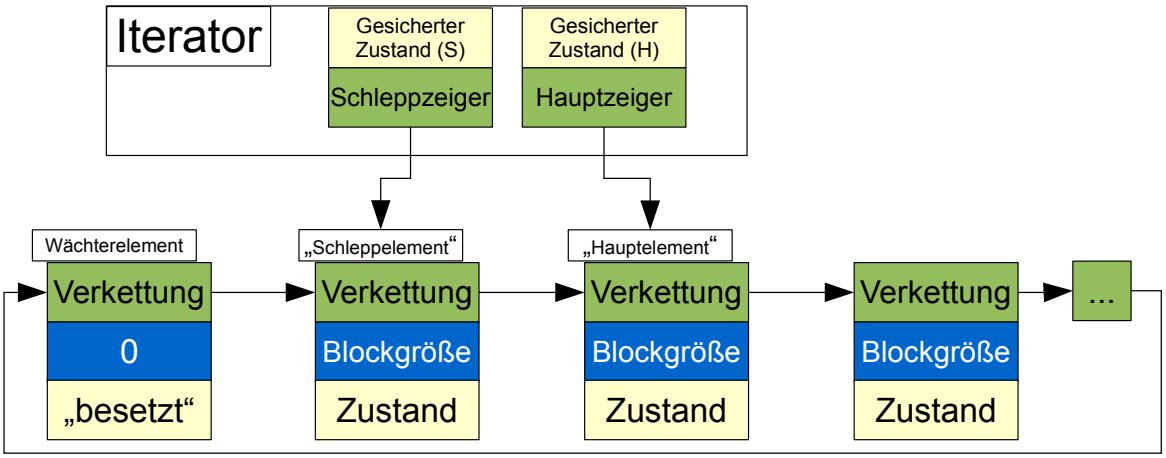

Abbildung 3.14: Iterator und Freispeicherliste

Aufbau der Listenelemente

Ein Freispeicherblock besteht aus einem oder mehreren ungenutzten Fragmenten („Slots“). Jeder Freispeicherblock wird von einem Listenelement der Freispeicherliste repräsentiert. Diese Listenelemente werden im ersten Fragment des Blocks gespeichert. So fällt für die Freispeicherliste kein zusätzlicher Speicherplatzbedarf an. Jedes Listenelement verfügt über folgende Eigenschaften:

- Die Adresse des nächsten Listenelements (Verkettungszeiger)
- Die Größe des Freispeicherblocks, in dem es sich befindet
- Den Zustand des Elementes („entnehmbar“ oder „besetzt“)

Für Verkettungszeiger und Größe wird jeweils ein Speicherplatz für einen Zeiger reserviert. Zur Speicherung des Zustands ist mindestens ein Byte notwendig. Auf Zweierpotenzen gerundet, nimmt ein Listenelement damit den Platz von vier Zeigern ein. Dies entspricht der minimalen Fragmentgröße, die nötig ist, um Instanzfelder (z.B. vom Typ „long“) ohne Einschränkungen speichern zu können.

Der Verkettungszeiger zeigt auf das nächste Listenelement. Der des letzten Elementes zeigt auf das Wächterelement. Dieses Wächterelement befindet sich im Domänendeskriptor. Durch den Verkettungszeiger dieses Wächters ist der erste Freispeicherblock erreichbar. Im Gegensatz zum IRR, werden die Listenelemente nicht nach ihrer Anfangsadresse sortiert. Es wird keine Verschmelzung aneinander angrenzender Speicherblöcke vorgenommen. Die Verschmelzung gemeinsam freigegebener Blöcke geschieht jedoch in der Löschphase automatisch durch das Durchsuchen der Bitkarte. Im Bezug auf die Allokation zusammenhängender Vektoren könnte die Verschmelzung von Blöcken den mittleren Durchsatz der Anwendung jedoch erhöhen.

Um die Größe des Blocks zu beschreiben, wird die Anzahl der Fragmente gespeichert, die er umfasst. Dazu wird eine vorzeichenlose Ganzzahl von der Größe eines Zeigers verwendet. Der Anfang des Blocks wird von der Adresse des Listenelements bestimmt.

Der Zustand des Listenelements wird verwendet, um nebenläufige Zugriffe auf die Liste zu koordinieren. Der Zustand „entnehmbar“ bedeutet, dass das Element aus der Liste entfernt werden darf. Befindet es sich im Zustand „besetzt“, zeigt mindestens ein Iterator auf das Element. Würde das Element dann entfernt werden, würde der Iterator von der Liste „entgleisen“.

Der Iterator

Der Speicherbereiniger iteriert über die Liste, um diese in der Bitkarte zu markieren. Anwendungsfäden durchsuchen die Liste und entnehmen gegebenenfalls Elemente.

Solange der Iterator verwendet wird, darf der zugehörige Ausführungsstrang nicht blockieren. Durch die Prioritäten wird dann sichergestellt, dass sich Iterationen nur gestapelt überlappen können (LIFO, engl. „last in – first out“). D.h. falls eine Iteration die andere unterbricht, wird die unterbrechende Iteration abgeschlossen, bevor die unterbrochene fortgesetzt wird.

Der Speicherbereiniger besitzt die niedrigste Priorität im System. Er kann von jedem Anwendungsfaden unterbrochen werden. Die Anwendungsfäden können während der Allokation von höherprioren Fäden unterbrochen werden.

Attribute: Der Iterator verfügt über folgende Attribute:

- Hauptzeiger
- Schleppzeiger
- Den gesicherten Zustand des aktuellen Elements
- Den gesicherten Zustand des vorherigen Elements

Über den Hauptzeiger kann auf das aktuelle Element zugegriffen werden. Erreicht dieser das Wächterelement, wurde die Liste einmal durchlaufen.

Der Schleppzeiger zeigt auf das vorherige Element. Er wird benötigt, um das Entfernen des aktuellen Elements zu ermöglichen.

Zusätzlich werden die beiden Zustände der Elemente, bevor sie vom Iterator erreicht wurden, gespeichert. Zeigt der Iterator auf ein Listenelement, muss dieses als „besetzt“ markiert werden. Ist der Iterator der letzte, der auf das Element zeigt, muss es als „entnehmbar“ markiert werden, wenn er das Element wieder verlässt. War der Iterator der erste, der auf das Element gezeigt hat, muss er auch der letzte sein, der auf das Element zeigt. Dies ergibt sich daraus, dass sich die Iterationen nur gestapelt überlappen können (hier: „first in – last out“).

Funktionen Mit dem unterbrechungstoleranten Iterator können Freispeicherblöcke besucht und aus der Liste entfernt werden. Er verfügt über folgende Funktionen:

- „begin“ initialisiert den Iterator
- „release“ gibt den Iterator wieder frei
- „unlink“ entfernt das aktuelle Element aus der Liste
- „inc“ rückt den Iterator um ein Element weiter

Initialisierung des Iterators

Die Initialisierung umfasst folgende Schritte:

1. Schleppzeiger auf das Wächterelement der Liste setzen
2. Hauptzeiger auf das erste Element setzen
3. Zustand des ersten Elements speichern
4. Erstes Element gegebenenfalls als „besetzt“ markieren

Der Schleppzeiger zeigt auf das Element vor dem Hauptelement. In diesem Fall ist das das Wächterelement. Da dieses nicht aus der Liste entnommen werden kann, wird als Zustand des Wächterelements „besetzt“ gespeichert.

Die Schritte 2 bis 4 müssen unteilbar ausgeführt werden. Hierbei wird der Hauptzeiger auf das erste Element nach dem Wächterelement gesetzt. Anschließend wird dessen Zustand gelesen. Das Element muss als „besetzt“ markiert werden, falls das noch nicht der Fall ist.

Eine Unterbrechung zwischen Schritt 2 und 4 könnte das erste Element entfernen. Der Iterator würde damit aus der Liste „entgleisen“ und könnte die Fragmente eines bereits konstruierten Objektes beschreiben.

Nach Schritt 4 ist sichergestellt, dass das erste Element nicht aus der Liste entfernt werden kann. War das Element bereits besetzt, wird es erst nach dieser Iteration freigegeben (LIFO). Wird es von diesem Iterator besetzt, wird er es erst dann freigeben, wenn er es nicht mehr benötigt.

Freigabe des Iterators

Bei der Freigabe des Iterators, werden alle Elemente, die dieser gegenwärtig besetzt auf „entnehmbar“ gesetzt. Jeder Iterator, der initialisiert wurde, muss nach seiner Verwendung wieder freigegeben werden. Andernfalls können die entsprechenden Listenelemente nie wieder aus der Liste entnommen werden.

Elemente entfernen

Ein Iterator darf das Element, auf das der Hauptzeiger zeigt, dann entnehmen, wenn er es „besetzt“ hält. Hält dieser Iterator das Element nicht besetzt, gibt es einen anderen Iterator, der bereits auf dieses Element zeigt. Würde dieser Iterator das Element entnehmen, würde der andere Iterator „entgleisen“, da er nicht mehr auf ein gültiges Listenelement zeigt. Ungeachtet dessen, kann ein Iterator, der das Element nicht besetzt, den zugehörigen Freispeicherblock verkleinern. Das erste Fragment des Blocks, das das Listenelement enthält, muss dabei erhalten bleiben.

Beim Entfernen eines Elementes, wird dem Verkettungszeiger des Elementes, auf das der Schleppzeiger zeigt, der Wert des Verkettungszeigers des Hauptelements zugewiesen. Die Liste überspringt nun das Hauptelement.

Hierbei ist zu beachten, dass sich zwischen dem Schleppzeiger und dem Hauptzeiger keine weiteren Elemente befinden. Dies ist dadurch sichergestellt, dass nur der Speicherbereiniger Listenelemente einfügt. Dieser kann die Iteration nicht unterbrechen, da er die niedrigste Priorität im System besitzt.

Das Lesen des Verkettungszeigers des Hauptelements und die Zuweisung an das Schleppelement muss atomar geschehen. Sonst könnte eine höherpriore Iteration das Element, auf das der Verkettungszeiger zeigt, entfernen. Nach der Zuweisung, würde das Schleppelement auf kein gültiges Listenelement mehr zeigen.

Vorrücken des Iterators zum nächsten Element

Das Weiterrücken des Iterators unterscheidet sich je nach dem, ob er das Hauptelement aus der Liste entfernt hat, oder nicht.

Wurde das Hauptelement nicht entnommen, wird der Schleppzeiger auf den Hauptzeiger gesetzt und der Hauptzeiger auf das nächste Element. Dazu sind folgende Schritte notwendig:

1. Lese den Verkettungszeiger des Hauptelements als nächsten Hauptzeiger
2. Lese den Zustand des nächsten Hauptelements
3. Setzte den Zustand des nächsten Hauptelements auf „besetzt“
4. Setze den Zustand des Schleppelements zurück
5. Der Schleppzeiger erhält den Wert des vorherigen Hauptzeigers
6. Der gesicherte Zustand wird weitergerückt

Die Schritte 1 bis 3 müssen atomar ablaufen. Während dieser Schritte stellt der Iterator das nächste Element fest und besetzt dieses, falls es nicht schon besetzt ist. Würde der Vorgang zwischen Schritt 1 und 3 unterbrochen werden, könnte eine anderer Iterator das nächste Element entfernen. Der Iterator „entgleist“.

Anschließend wird das Schleppelement „entnehmbar“, falls es von diesem Iterator besetzt gehalten wurde. Der Schleppzeiger wird auf den Wert des ehemaligen Hauptzeiger gesetzt. Der im Iterator gespeicherte Zustand des ehemaligen Hauptelement wird nun an den gespeicherten Zustand des Schleppelements überwiesen.

Wurde das Hauptelement entnommen, verbleibt der Schleppzeiger auf seinem gegenwärtigen Element. Dem Hauptzeiger wird der Wert des Verkettungszeigers des Schleppelements zugewiesen. Der Zustand des neuen Hauptelementes wird wieder gesichert und gegebenenfalls auf “besetzt“ geändert. Diese Schritte müssen aus den gleichen Gründen wie vorher atomar ablaufen.

Einfügen von Elementen

Neue Elemente der Freispeicherliste werden immer zwischen dem ersten und dem Wächterelement eingefügt. Der Verkettungszeiger des neuen Elements wird auf das erste Listenelement gesetzt. Anschließend wird der Verkettungszeiger des Wächterelements auf das neue Listenelement gesetzt. Diese beiden Schritte laufen ununterbrechbar ab. Andernfalls könnte eine Allokation das erste Element entfernen, nachdem der Verkettungszeiger des neuen Elements gesetzt wurde. Das neue Element würde dann nicht mehr auf ein gültiges Element zeigen. Der Rest der Liste wäre verloren.

3.2.3 Der Rückgratspeicher

Der Rückgratspeicher kann mit einem replizierenden oder mit einem generationellen Speicherbereiniger verwaltet werden. Für jede Domäne wird ein Rückgratspeicher der konfigurierten Größe angelegt. Im Domänendeskriptor wird die Anfangs- und Endadresse des Speichers abgelegt. Welche weiteren Informationen nötig sind, hängt von der Art der Speicherbereinigung ab.

Der replizierende Rückgratspeicher

Wie in Abschnitt 2.3.2 beschrieben wird, teilen replizierende Speicherbereiniger ihre Halde in zwei Hälften. Die jeweils aktive Hälfte wird sukzessive mit Rückgraten gefüllt. Zur Bereinigung tauschen die aktive und inaktive Hälfte ihre Rollen. Dazu müssen folgende Zustände im Domänendeskriptor gespeichert werden:

- Die Position, bis zu der die aktive Hälfte belegt ist
- Ein Bit, das anzeigt, welche Hälfte gerade aktiv ist

Der generationelle Rückgratspeicher

Der in Abschnitt 2.4 beschriebene generationelle Speicherbereiniger teilt seine Halde in insgesamt drei Teilbereiche auf. Es gibt die junge Generation, die die neu angelegten

Rückgrate aufnimmt. Es gibt die alte Generation, in die Rückgrate verschoben werden, die eine Speicherbereinigungsphase überlebt haben. Die alte Generation wird in zwei Hälften unterteilt, die mit einem replizierendem Speicherbereiniger verwaltet werden. In der inaktiven Hälfte werden Rückgrate während der Bereinigungsphase angelegt.

Damit gibt es drei Teilbereiche, in denen Rückgrate angelegt werden können. Daher muss für alle drei Bereiche die Position gespeichert werden können, bis zu der sie belegt sind.

Zusätzlich wird für die alte Generation ein zweites Farbbit benötigt. Schließlich wird diese nicht in jeder Bereinigungsphase mitbereinigt. Würde das Farbbit des Fragmentspeichers mitbenutzt werden, könnte es sein, dass die Bereinigung der alten Generation immer mit der gleichen Farbe stattfindet. So könnten keine Rückgrate als „tot“ erkannt werden.

In der gegenwärtigen Implementierung wird die alte Generation in jeder vierten Bereinigungsphase aufgeräumt. Daher ist ein Zähler nötig, der die Anzahl der Bereinigungsphasen mitzählt.

Insgesamt sind damit folgende Informationen im Domänendeskriptor zu speichern:

- Ein Bit, das anzeigt, welche Hälfte der alten Generation aktiv ist
- Ein Bit, das anzeigt, ob der Speicher gerade bereinigt wird
- Das Farbbit der alten Generation
- Die Anzahl der bisherigen Bereinigungsphasen
- Die Füllstände
  - der jungen Generation
  - der aktiven Hälfte der alten Generation
  - der inaktiven Hälfte der alten Generation

3.2.4 Konfiguration der Halde

Abbildung 3.15 zeigt eine mögliche Konfiguration einer mit „FragGC“ verwalteten Halde.

Das Attribut „HeapSize“ gibt die Größe des Fragmentspeichers an. Sie beträgt hier 256 Kibibyte. Die Größe des Rückgratspeichers ist nicht mit inbegriffen. Das Rückgrat wird als Verwaltungsstruktur gesehen, da es keine Nutzdaten trägt. Dies hängt damit zusammen, dass FragGC die Verwaltung des Fragmentspeichers auf den „slotbasierten“ Speicherbereinigern (z.B. IdleRoundRobin) aufbaut. Auf Grund der Fragmentierungstoleranz kann beim Anlegen der Halde damit gerechnet werden, dass (fast) alle freien Objektfragmente genutzt werden können. Die Ausnahme bilden Fragmente, die von einem Iterator besetzt werden (siehe Abschnitt 3.3.1). So müssen pro Programmfaden im System (einschließlich Speicherbereiniger) zwei Fragmente eingeplant werden, die nicht verwendet werden können, obwohl sie frei sind.

```
System (Example) {
     Domain (MyDomain) {
          Heap = FragGC {
               HeapSize = 256Ki;
               SlotSize = 16;
               SpineSize = 48Ki;
               SpineNurserySize = 42Ki;
               GCTMode = "Lazy";
          }
          # ...
          # Informationen über Tasks, Alarme etc.
     }
}
```

Abbildung 3.15: Konfiguration der Halde

Aus diesem Grund kann auch eine Slotgröße angegeben werden. Diese muss der Fragmentgröße entsprechen. Zukünftig könnte ein vielfaches der Fragmentgröße angegeben werden können. Dadurch könnte die Lokalität zusammengehöriger Daten in der entsprechenden Domäne verbessert werden. Wird keine Slotgröße angegeben, wird sie automatisch auf die Fragmentgröße gesetzt.

Als nächstes wurde die Gesamtgröße des Rückgratspeichers angegeben („SpineSize“). Diese umfasst beide Hälften bzw. beide Generationen des Rückgratspeichers. Wird keine „SpineSize“ angegeben, wird sie anhand der Fragmentspeichergröße bestimmt. Hierzu wird die Anzahl der Slots bzw. Fragmente bestimmt, die auf der Halde Platz finden. Anschließend wird im Rückgratspeicher Platz für zwei Zeiger pro Fragment eingeräumt.

Des Weiteren kann die Größe der jungen Generation des Rückgratspeichers angegeben werden („SpineNurserySize“). Hierdurch wird kein zusätzlicher Speicherplatz belegt. Die junge Generation liegt im Rückgratspeicher. Wird keine „SpineNurserySize“ angegeben, wird ein replizierender statt einem generationellen Speicherbereiniger verwendet.

„GCTMode“ gibt an, wie sich der OSEK-Task verhält, der die Speicherbereinigung ausführt. Im Modus „Lazy“ terminiert dieser Task, wenn keine Domäne einer Speicherbereinigung bedarf. Durch Allokationen wird der Task wieder aktiviert und läuft an, sobald kein anderer Task mehr bereit zur Ausführung ist. Im Modus „Workaholic“ terminiert der Task nie. Verlangt keine Domäne nach einer Bereinigung, wird diejenige bereinigt die den numerisch niedrigsten „need“-Wert besitzt (siehe Abschnitt 2.1.3).

3.3 Die Allokation

Dieser Abschnitt beschäftigt sich mit dem Anlegen von Objekten. Zuerst wird erläutert, wie die Grundbestandteile der Objekte angefordert werden: Fragmente und Rückgrate. Anschließend wird auf reguläre Objekte und Vektoren eingegangen.

3.3.1 Allokation von Fragmenten

Verfügbare Fragmente werden in einer Freispeicherliste gesammelt. Um diese Liste unterbrechungstolerant durchsuchen und manipulieren zu können, wird der in Abschnitt 3.2.2 beschriebene Iterator verwendet. Da die Fragmente nicht zusammenhängen müssen, werden von jedem Freispeicherblock so viele Elemente entnommen, wie nötig sind, um die Anfrage zu erfüllen. Wird das Ende der Liste erreicht, ohne dass genug Fragmente gesammelt werden konnten, wird eine „OutOfMemory“-Ausnahme geworfen. Die aus der Freispeicherliste entfernten Elemente werden in eine lokale Liste eingereiht. Da diese nur einem Programmfaden zugänglich ist, benötigt sie keine Synchronisierung. Im Folgenden wird beschrieben, wie die einzelnen Blöcke bearbeitet werden.

„Besetzt“ der Iterator das aktuelle Listenelement, kann er es entfernen. Dies wird dann getan, wenn der zugehörige Block nicht mehr Fragmente enthält, als zur Erfüllung der Anfrage noch nötig sind.

Treffen diese beiden Bedingungen nicht zu, wird der Block gespalten. Dabei entsteht hinter dem Freispeicherblock, der in der Liste hängt, ein weiterer Block. Dazu werden folgende Schritte durchgeführt:

1. Lese die Größe des Freispeicherblocks
2. Bestimme die Anzahl der zu entnehmenden Fragmente
3. Bestimme die verbleibende Größe des Blocks
4. Trage die verbleibende Größe des Blocks ein

Dabei muss mindestens ein Fragment im Freispeicherblock verbleiben, um das Listenelement zu beherbergen (zweiter Schritt). Es werden natürlich nicht mehr Blöcke entfernt als benötigt. Wird die Größe, die im dritten Schritt bestimmt wurde, auf die Anfangsadresse des Freispeicherblocks addiert, ergibt sich die Adresse des neuen Blocks. Dieser kann nun in die lokale Liste eingefügt werden.

Diese Schritte müssen unteilbar durchgeführt werden. Verkleinert eine unterbrechende Allokation den Block zwischen dem ersten und dem vierten Schritt, besteht ein „lost update“ Problem. Die Änderungen, die die unterbrechende Allokation vorgenommen hat, gehen verloren.

Im ungünstigsten Fall muss jedes Fragment von einem anderen Freispeicherblock genommen werden. Des Weiteren könnten „besetzte“ Blöcke der Größe 1 besucht werden. Aus diesen kann kein Block entnommen werden. Dadurch, dass sie von einem anderen Iterator besetzt werden, können sie auch nicht aus der Liste entnommen werden. Die Iteration, die den Block besetzt, muss eine niedrigere Priorität besitzen, als die gegenwärtige Allokation. Iteratoren werden von Allokationen und dem Bereiniger verwendet. Die Anzahl der Anwendungsfäden mit einer niedrigeren Priorität, die eine Allokation durchführen könnten, ist für jeden Faden bekannt. Es werden maximal zwei Fragmente pro Programmfaden besetzt: Schlepp- und Hauptelement. Damit lässt sich

eine Obergrenze für die Anzahl der nicht nutzbaren Fragmente bestimmen. Die WCET richtet sich also nach der Anzahl der anzulegenden Fragmente und nach der maximalen Anzahl der nicht nutzbaren Fragmente.

3.3.2 Allokation zusammenhängender Fragmente

Um zusammenhängende Vektoren anzulegen, kann versucht werden zusammenhängende Fragmente zu allozieren. Um zu einer verlässlichen WCET zu kommen, ist die Anzahl der Versuche begrenzt. Wurde eine bestimmte Anzahl von Freispeicherblöcken besucht, oder das Ende der Liste erreicht, wird ein Fehlschlag signalisiert. Im Folgenden wird beschrieben, wie die einzelnen Blöcke bearbeitet werden.

„Besetzt“ der Iterator das aktuelle Listenelement, kann er es entfernen. Dies wird getan, wenn die Anzahl der Fragmente im Block der Anzahl der angeforderten Elemente entspricht. Der entfernte Block wird anschließend als Ergebnis zurückgegeben.

Treffen diese beiden Bedingungen nicht zu, wird geprüft, ob der Block gespalten werden kann. Dies ist dann der Fall, wenn er mehr Fragmente beherbergt, als angefordert wurden. Dazu werden folgende Schritte durchgeführt:

1. Überprüfe, ob der Block mehr Fragmente beherbergt, als angefordert wurden
2. Bestimme die verbleibende Größe des Blocks
3. Trage die verbleibende Größe des Blocks ein

Diese Schritte müssen unteilbar ausgeführt werden, dass keine andere Allokation den Block verkleinert. Der dadurch entstandene Block wird als Ergebnis zurückgegeben.

Die WCET ist durch die Anzahl der Versuche begrenzt, die der Allokation eingeräumt werden.

3.3.3 Allokation von Rückgraten

Rückgrate werden hintereinander angelegt, indem ein Index einfach um den angeforderten Betrag erhöht wird. Die Komplexität dieser Operation ist konstant - unabhängig von der Größe des Rückgrats.

Abbildung 3.16 zeigt das Erhöhen des Index'. Der alte Wert des Index' wird gelesen und um die Größe des anzulegenden Rückgrats erhöht (Z. 14f). Passt das Rückgrat nicht in den Speicherbereich, wird der Wert *EXHAUSTED* zurückgegeben, welcher anschließend zu einer Ausnahmebehandlung führt. Passt das Rückgrat in den Speicherbereich, der die Anfrage bedient, wird der Index mit dem erhöhten Wert beschrieben. Die Adresse des Rückgrats ergibt sich, indem man *pos* auf den Anfang des Speicherbereiches addiert, der die Anfrage bedient hat. Zwischen Zeile 14 und Zeile 19 liegt ein Lesen-Ändern-Schreiben-Problem vor. Daher wird die Operation von einer Unterbrechungssperre geschützt („FGC_FL_LOCK“ und „UNLOCK“). Dies könnte auch durch

```
1   uintptr_t
2   /*Größe des Bereichs, in dem das Rückgrat angelegt werden soll*/
3     spine_size = ...,
4   /*Größe des Rückgrats, das angelegt werden soll */
5     alloc_size = ...,
6   /*Ergebnis: Position des angelegten Rückgrats*/
7     pos,
8   /*Ergebnis: Neue Position des Index*/
9     npos;
10  /*Zeiger auf den Index, der verschoben wird*/
11  uintptr_t *pbump = ...;
12
13  FGC_FL_LOCK {
14    pos = *pbump;
15    npos = pos + alloc_size;
16    if (npos > spine_size) {
17      pos = EXHAUSTED;
18    } else {
19      *pbump = npos;
20    }
21  } FGC_FL_UNLOCK;
```

Abbildung 3.16: Anlegen eines Rückgrats

eine atomare Addition erreicht werden, sofern diese vom Zielprozessor unterstützt wird. Der TC1796 [5] bietet das nicht.

Je nachdem, ob die Rückgrate von einem replizierenden oder generationellen Speicherbereiniger verwaltet werden, werden die Variablen *spine_size*, *alloc_size* und *pbump* verschieden zugewiesen. Bei keinem der beiden Verfahren treten zusätzliche Unterbrechungssperren auf.

Replizierende Speicherbereinigung

Werden die Rückgrate von einem replizierenden Speicherbereiniger verwaltet, beträgt *spine_size* immer die Hälfte des gesamten Rückgratspeichers. Es wird nur ein Index verwendet, der immer für die jeweils aktive Speicherhälfte gilt. Daher kann *pbump* nur auf diesen Index zeigen. *alloc_size* entspricht der Größe des angeforderten Rückgrats.

Um die Adresse des angelegten Rückgrats zu erlangen, wird abgefragt, welche Hälfte gegenwärtig aktiv ist. Diese Abfrage muss nicht von Sperren geschützt werden, da nur der Speicherbereiniger diesen Zustand ändert. Dieser kann die Allokation nicht unterbrechen, da er die niedrigste Priorität besitzt. Die Adresse ergibt sich indem man *pos* zur Anfangsadresse der aktiven Hälfte addiert.

Generationelle Speicherbereinigung

Werden die Rückgrate von einem generationellen Speicherbereiniger verwaltet, hängen alle Variablen vom gegenwärtigen Zustand ab. Der Speicherbereiniger kann sich in oder außerhalb der Markierungsphase befinden. Wie in Abschnitt 3.4.2 beschrieben wird, werden die Rückgrate im gegenwärtigen „Allokationsbereich“ angelegt.

Es existieren drei Indices. Ein Index verwaltet die junge Generation. Die anderen beiden Indices verwalten die aktive und inaktive Hälfte der alten Generation.

Wurde die Speicherbereinigungsphase nicht unterbrochen, werden die Rückgrate in der jungen Generation angelegt. Dieser Vorgang ist mit dem des replizierenden Bereinigers vergleichbar.

Wurde eine Markierungsphase unterbrochen, werden die Rückgrate in der inaktiven Hälfte der alten Generation angelegt. Nach der Bereinigung werden diese in die junge Generation verschoben. Um dieses Verschieben zu ermöglichen, muss dem Rückgrat ein Zeiger auf das Wächterelement angefügt werden. Dieser Zeiger erhöht den Wert von *alloc_size*. Die Variablen *pbump* und *spine_size* müssen dem Allokationsbereich entsprechend angepasst werden.

3.3.4 Allokation regulärer Objekte

Ablauf Die Allokation regulärer Objekte ist nicht mit der bisherigen Schnittstelle kompatibel. Um zusammenhängende Objekte anzulegen, musste die Größe des Objekts und der Abstand des Objektkopfes zum Anfang des Objektes bekannt sein. Zur Allokation fragmentierter Objekte, muss die Anzahl der Fragmente, die das Objekt umfasst und die Anzahl der Referenzen pro Fragment bekannt sein. Diese Informationen werden der Allokationsfunktion übergeben. Eine Allokation unter alleiniger Angabe der Klassenkennung wird nicht unterstützt, da dazu eine Tabelle angelegt werden müsste, in der obige Informationen nachgeschlagen werden können. Da das fragmentierte Objektformat nicht mit dem bidirektionalen Objektformat kombinierbar ist, entstehen daraus keine Probleme.

Reguläre Objekte verfügen über kein Rückgrat. Als erstes wird die benötigte Anzahl an Fragmenten angefordert (siehe Abschnitt 3.3.1). Die Fragmente werden in einer Liste von Freispeicherblöcken geliefert. Diese Blöcke werden nun in Fragmente zerlegt und mit Verkettungszeigern verbunden. Anschließend werden die Verkettungszeiger – je nach dem wie viele Referenzen sich in einem Fragment befinden – verschoben. Die Referenz auf das Objekt ergibt sich, indem der Zeiger auf das erste Fragment um die im ersten Fragment enthaltenen Referenzen verschoben wird. Abschließend wird die Referenz auf das Objekt dem Konstruktor übergeben.

Maximale Ausführungszeit Die WCET zum Anlegen der Fragmente richtet sich nach deren Anzahl (Abschnitt 3.3.1). Diese ist für jede Klasse konstant. Die WCET zur Initialisierung der Fragmente ist ebenfalls von deren Anzahl abhängig. Damit kann eine

obere Schranke für die Ausführungszeit der Allokation der jeweiligen Klasse angegeben werden.

Maximaler Speicherverbrauch Wie viele Fragmente benötigt werden, ist für jede Klasse bekannt. Die Anzahl ergibt sich aus der Beschreibung der Klasse und der Fragmentgröße nach dem, in Abschnitt 3.1.3 beschriebenen, Algorithmus.

3.3.5 Allokation von Vektoren

Ablauf Vektoren bestehen aus Rückgraten und Fragmenten. Es existieren zwei verschiedene Vektorformate. Diese unterscheiden sich darin, ob zusammenhängende Vektoren darstellbar sind, oder nicht.

Zur Allokation eines Vektors, wird die Anzahl der Elemente angegeben, die der Vektor aufnehmen können soll. Zusätzlich wird der Zweierlogarithmus der Größe der Elemente angegeben.

Zuerst werden die benötigten Fragmente angefordert. Die Anzahl der Fragmente ergibt sich aus der Anzahl der Nutzdatenfragmente und dem Wächterfragment. Die Anzahl der Nutzdatenfragmente ergibt sich nach $\lceil \frac{N \cdot E}{F} \rceil$. N ist die Anzahl der Elemente. E ist die Größe eines Elements. F ist die Größe eines Fragments. Falls konfiguriert, kann vor der Allokation der Fragmente versucht werden, die Fragmente zusammenhängend anzulegen (siehe Abschnitt 3.3.2).

Hat die Liste, in der die angeforderten Fragmente gesammelt wurden, nur ein Element (neben dem Wächterelement), hängen alle Fragmente zusammen. Werden zusammenhängende Vektoren unterstützt, kann das Element als Vektor interpretiert werden. Nach Initialisierung der Metainformationen, wird die Adresse des Elements als Ergebnis zurückgegeben.

Sind die Elemente nicht zusammenhängend wird ein Rückgrat angelegt. Werden eingebettete Rückgrate unterstützt und ist der Vektor klein genug, kann das Wächterfragment als Rückgrat interpretiert werden. Sonst wird ein Rückgrat vom Rückgratspeicher angefordert. Ins Wächterfragment wird dann die Länge 0 und ein Zeiger auf das Rückgrat eingetragen.

Die Adresse des Wächterfragments wird anschließend, nach der Initialisierung der Metainformationen, als Ergebnis zurückgegeben.

Ein Sonderfall tritt auf, wenn ein leerer Vektor angelegt werden soll. Da die Länge 0 bedeutet, dass der Vektor ein Rückgrat besitzt, muss dieser ein Rückgrat besitzen. Dieses Rückgrat ist ebenfalls leer.

Maximale Ausführungszeit Die WCET der Allokation lässt sich anhand der beteiligten Funktionen bestimmen. Die WCET des Anlegens der Fragmente hängt von der Anzahl der angeforderten Fragmente ab (Abschnitt 3.3.1). Wird explizit versucht zusammenhängende Vektoren anzulegen, kommt die WCET dessen zusätzlich hinzu. Sie

ist durch die Anzahl der Versuche begrenzt, die der Allokation eingeräumt werden (Abschnitt 3.3.2).

Maximaler Speicherverbrauch Für den Speicherplatz den ein Vektor einnimmt kann ebenfalls eine obere Grenze angegeben werden. Die Größe der Nutzdaten wird dazu auf eine, durch die Fragmentgröße teilbare, Größe aufgerundet. Hinzu kommt die Größe des Wächterfragments. Wird ein Rückgrat angefordert, enthält dieses für jedes Nutzdatenfragment einen Zeiger. Zusätzlich wird in dieser Implementierung ein Objektkopf, die Größe und der Typ gespeichert. Der generationelle Rückgratspeicher kann zusätzlich einen Zeiger auf das Wächterfragment hinzufügen.

3.4 Die Speicherbereinigung

Dieser Abschnitt beschreibt, wie eine Speicherbereinigungsphase abläuft. Ein Durchlauf der Speicherbereinigung kann in folgende Phasen unterteilt werden:

1. Auswahl der Domäne
2. Vorbereitung des Rückgratspeichers
3. Markierungsphase
4. Nachbereitung des Rückgratspeichers
5. Löschphase
6. Nullen der Bitkarte

Nachdem der „GC-Task“ eine Domäne ausgewählt hat, die bereinigt werden soll, werden die lebendigen Objekte dieser Domäne markiert. Die toten Objekte werden anschließend gelöscht. Um die Bitkarte für die Bereinigung der nächste Domäne wiederverwenden zu können, wird sie zurückgesetzt. Im Folgenden werden diese Schritte genauer beschrieben.

3.4.1 ?? Auswahl der Domäne

Wie in Abschnitt 2.1.3 beschrieben, wird für jede Domäne ein „need“-Wert bestimmt, der sich wie folgt berechnet:

$$need = \text{FreieSlots} - \text{NeuAngeforderteSlots} - \frac{\text{Haldengröße}}{4}$$

Je kleiner „need“ numerisch ist, desto höher ist der Bedarf einer Speicherbereinigung. Der „GC-Task“ wechselt reihum durch die Domänen. Wird ein negativer „need“ berechnet, wird der Speicher bereinigt.

„FragGC“ verwaltet neben dem Fragmentspeicher auch noch einen Rückgratspeicher. Ist letzterer nicht groß genug, um alle Fragmente in einem Rückgrat zu verwalten, muss für das Rückgrat ebenfalls ein Need-Wert berechnet werden. Dieser berechnet sich analog zum „need“ des Fragmentspeichers. Das numerische Minimum aus dem „need“ des Rückgrat- und des Fragmentspeichers, ist der Need-Wert der Domäne.

Da der Speicherbereiniger auf die Anwendung abgestimmt werden muss, wird es dieser zusätzlich erlaubt, einen negativen Need-Wert zu erzwingen Dazu wird die bisher ignorierte Java-Methode „java/lang/Runtime.gc()“ verwendet. Diese setzt nun ein „ForceNeedBit“ im Domänendeskriptor und aktiviert den „GC-Task“. Ist dieses Bit gesetzt, wird als „need“ der Domäne maximal (numerisch) -1 angegeben. Dies ist der niedrigste Bedarf, der eine Bereinigung auslöst. Nach der Auslösung wird das „ForceNeedBit“ wieder gelöscht.

Damit weicht „gc()“ von seinem Standardverhalten ab. Denn KESOs Speicherbereiniger kann erst dann anlaufen, wenn keine anderen Programmfäden lauffähig sind. Daher wurde nach der Rückkehr aus „gc()“ noch keine Bereinigung durchgeführt. Der Programmfaden muss erst terminieren oder auf ein Ereignis warten.

3.4.2 Bereinigung des Rückgratspeichers

Zur Bereinigung des Rückgratspeichers, werden alle erreichbaren Rückgrate vom „Räumungsbereich“ in den „Zufluchtsbereich“ kopiert. Unerreichbare Rückgrate werden im Räumungsbereich zurückgelassen. Während der Bereinigung angelegte Rückgrate dürfen nicht im Räumungsbereich platziert werden, da sie sonst übersehen werden könnten. Sie werden im „Allokationsbereich“ angelegt. Wo sich Räumungs-, Zufluchts- und Allokationsbereich befinden, hängt von der Implementierung des Rückgratspeichers ab.

Vor der Markierungsphase muss der Rückgratspeicher vorbereitet werden. Hierbei werden die verschiedenen Bereiche festgelegt.

Wird während der Markierungsphase ein Vektor erreicht, der über ein Rückgrat verfügt, wird überprüft, ob es sich im Räumungsbereich befindet. Ist dies der Fall, wird ein neues Rückgrat im Zufluchtsbereich angelegt. Dann wird der Inhalt des alten Rückgrats in das neue kopiert. Abschließend wird der Rückgratzeiger des Vektors auf das neue Rückgrat gesetzt.

Bereinigung des replizierenden Rückgratspeichers

Zur Vorbereitung der Markierungsphase tauschen die inaktive und die aktive Hälfte ihre Rollen. Der Index der angibt, wie weit die aktive Hälfte belegt ist, wird auf 0 zurückgesetzt.

Der Räumungsbereich ist die nun inaktive Hälfte. Die nun aktive Hälfte ist Zufluchts- und Allokationsbereich zugleich.

Rückgrate werden genauso in der aktiven Hälfte angelegt, wie außerhalb der Markierungsphase. Beim Umkopieren vom Räumungs- in den Zufluchtsbereich, findet die nötige Allokation auf die gleiche Weise statt.

Eine Nachbereitung der Markierungsphase ist nicht nötig. Die lebendigen Rückgrate wurden bereits repliziert. Allokationen finden auch weiterhin in der aktiven Hälfte statt.

Bereinigung des generationellen Rückgratspeichers

Bei der Bereinigung des generationellen Rückgratspeichers, werden Überlebende der jungen Generation in die alte Generation verschoben. Der Räumungsbereich ist also die junge Generation. Der Zufluchtsbereich ist die aktive Hälfte der alten Generation.

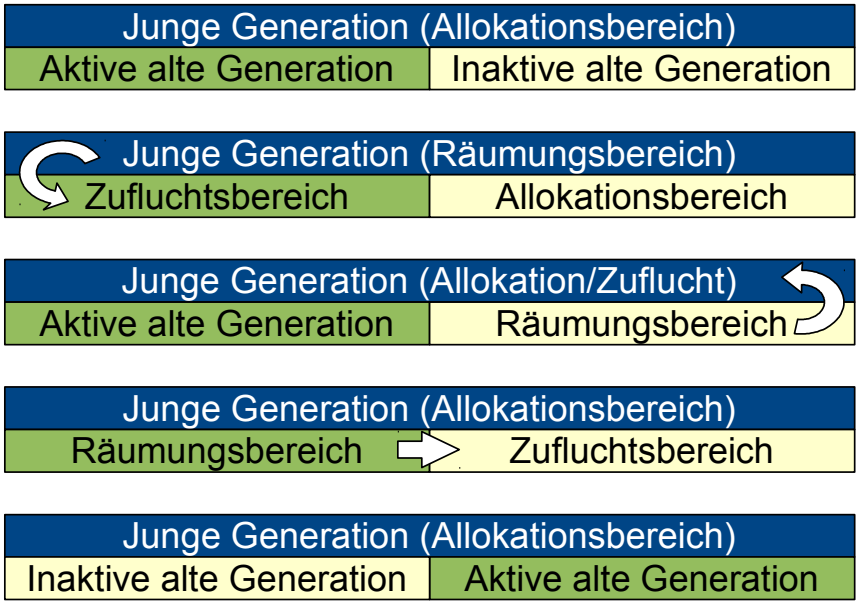

Abbildung 3.17: Bereinigung des generationellen Rückgratspeichers

Vorbereitung der Markierungsphase Anders als der replizierende Rückgratspeicher befindet sich der generationelle während der Markierungsphase in einem besonderen Zustand. Außerhalb der Markierungsphase, werden Rückgrate in der jungen Generation angelegt. Würden sie während der Markierungsphase weiterhin dort angelegt werden könnten sie entweder übersehen werden, oder könnten sofort „altern“. Würden Rückgrate zu früh in die alte Generation kopiert werden, könnte dies deren Kapazität erschöpfen. Schließlich könnte es sich um Rückgrate handeln, die sonst keine Bereinigungsphase überleben würden. Daher werden Rückgrate in der inaktiven Hälfte der alten Generation angelegt. Diese ist damit der „Allokationsbereich“. Der Zustand des Rückgratspeichers während der Bereinigungsphase wird in Abbildung 3.17 (zweite Zeile) dargestellt.

Dadurch dass die inaktive alte Generation als Allokationsbereich genutzt wird, ergibt sich eine weitere Einschränkung der Anwendung. Der Allokationsbereich muss alle Rückgrate, die während der Markierungsphase angelegt werden, aufnehmen können. Steht der Allokationsrate (siehe Abschnitt 2.2.4) nicht genug Schlupfzeit gegenüber, um die Markierungsphase zu beenden, muss der Speicherplatz der alten Generation vergrößert werden.

Zur Vorbereitung der Markierungsphase wird die Anzahl der Bereinigungsphasen mitgezählt. Gegenwärtig wird in jeder vierten Phase eine Bereinigung der alten Generation durchgeführt. Dazu wird das Farbbit der alten Generation invertiert. Wird während der Markierungsphase ein Rückgrat erreicht, das sich in der alten Generation befindet, wird das Farbbit entsprechend gesetzt.

Nachbereitung der Markierungsphase Nach der Markierungsphase, wird der Allokationsbereich wieder auf die junge Generation gelegt. Der Index, der angibt, wie weit die junge Generation belegt ist, wird auf 0 zurückgesetzt.

Anschließend werden die Rückgrate, die während der Markierungsphase in der inaktiven alten Generation angelegt wurden, in die junge Hälfte verschoben. Der Rückgratzeiger der zugehörigen Vektoren wird entsprechend angepasst. Dies wird in der dritten Zeile der Abbildung 3.17 dargestellt.

Gegebenenfalls wird abschließend eine Bereinigung der alten Generation durchgeführt. Dazu wird über die Rückgrate der aktiven alten Generation iteriert. Wurde ein Rückgrat als erreichbar markiert, wird es in die andere Hälfte verschoben. Der Rückgratzeiger der zugehörigen Vektoren wird entsprechend angepasst. Die beiden Hälften der alten Generation tauschen ihre Rollen. Dieser Vorgang ist in den letzten beiden Zeilen der Abbildung zu sehen.

3.4.3 Markierungsphase

In der Markierungsphase werden alle erreichbaren Fragmente in der Bitkarte markiert. Des weiteren werden alle erreichbaren Rückgrate in ihren „Zufluchtsbereich“ kopiert. Eine Markierungsphase läuft wie folgt ab:

1. Markierung der Freispeicherliste
2. Invertieren der Farbmarkierung
3. Aktivierung der Schreibbarrieren
4. Ablegen der Wurzelmenge auf dem Arbeitsstapel
5. Traversierung des Erreichbarkeitsgraphen
6. Deaktivierung der Schreibbarrieren

Im Folgenden wird erläutert was in den einzelnen Schritten passiert. Außerdem wird geklärt, weshalb sie in dieser Reihenfolge ablaufen müssen.

Markierung der Freispeicherliste

Anwendungen können die Markierungsphase unterbrechen und neue Objekte anlegen. Geschieht dies nachdem die Stapelspeicher der Anwendungsfäden durchsucht wurden, kann sich das Objekt in einer lokalen Referenz vor dem Speicherbereiniger „verstecken“. Da das Objekt nicht erreicht wird, werden die Fragmente, die es einnimmt, nicht in der Bitkarte markiert. Dadurch würden diese in der Löschphase freigegeben werden, obwohl das Objekt noch benutzt wird.

Um dies zu vermeiden, wird vor dem Feststellen der Erreichbarkeit jeder freie Block des Fragmentspeichers in der Bitkarte markiert. Ein Objekt, das nun angelegt wird, erhält markierte Fragmente, die nicht gelöscht werden.

Der Aufwand der Markierung hängt von der Anzahl der Freispeicherblöcke ab. Er steigt mit der Fragmentierung des Speichers. Im ungünstigsten Fall entspricht die Anzahl der Freispeicherblöcke der Anzahl an Objektfragmenten, die auf der Halde insgesamt angelegt werden können. Dieser tritt dann auf, wenn alle Objektfragmente, jedoch nie zwei nebeneinander liegende gleichzeitig, freigegeben wurden. Er kann deshalb eintreten, weil FragGC Blöcke nicht verschmilzt, solange sie nicht gleichzeitig freigegeben werden. Würde FragGC sie verschmelzen, wäre die maximale Anzahl an Blöcken höchstens halb so groß. Dies wäre dann der Fall, wenn nur jedes zweite Fragment der Halde belegt ist. Die Größe er Halde ist für jede Domäne konstant. Somit lässt sich die WCET für das Markieren der Freispeicherliste bestimmen.

Die WCET lässt sich noch weiter einschränken, wenn die in Abschnitt **??** beschriebene Berechnung des need-Werts verwendet wird. Der Speicherbereiniger aktiviert sich demnach nur dann, wenn mehr als 25% der Halde belegt sind. Damit entspräche die maximale Anzahl der Freispeicherblöcke drei Viertel der Haldenslots.

Invertierung der Farbmarkierung

Für jede Domäne wird gespeichert, welche Farbmarkierung aktuell bedeutet, dass ein Objekt erreichbar ist. Es gibt zwei verschiedene Farbwerte. Wird die Farbmarkierung invertiert, gelten alle Objekte, die vorher als erreichbar galten, als unerreichbar. Objekte, die vorher als unerreichbar galten, existieren nicht auf der Halde, da sie vom vorherigen Bereinigungszyklus gelöscht wurden. Objekte, die neu angelegt werden, werden in ihrem Objektkopf als erreichbar markiert.

Das Invertieren der Farbmarkierung darf nicht vor dem Markieren der Freispeicherliste geschehen. Ein Objekt, das nach dem Invertieren der Farbmarkierung, aber vor dem Markieren der Freispeicherliste angelegt werden würde, würde nicht in der Bitkarte markiert werden. Einerseits, da es nicht mehr in der Freispeicherliste vorhanden ist, wenn

diese markiert wird. Andererseits wird es beim Traversieren des Erreichbarkeitsgraphen nicht in der Bitkarte markiert, da es durch seine Farbmarkierung bereits als erreicht gilt.

Aktivierung und Deaktivierung der Schreibbarrieren

Die Schreibbarrieren (siehe Abschnitt 2.1.3) sind möglichst spät zu aktivieren, da sie die Ausführungsgeschwindigkeit der Anwendung beeinträchtigen. Sind sie aktiv, muss der Wert jedes Referenzfeldes, das überschrieben werden soll, auf den Arbeitsstapel des Speicherbereinigers abgelegt werden, falls es noch nicht erreicht wurde.

Die Aktivierung der Barrieren muss vor der Durchsuchung der Stapel der Programmfäden geschehen. Andernfalls könnte eine Referenz von einem noch nicht durchsuchten Programmfaden zu einem bereits durchsuchten Faden wandern. Diese Wanderung könnte über ein Klassen- oder Instanzfeld geschehen. Da das Feld dazu beschrieben und wieder überschrieben wird, wird dies von Schreibbarrieren verhindert.

Mit dem Abschließen der Markierungsphase werden die Schreibbarrieren wieder deaktiviert.

Die Aktivierung der Schreibbarriere geschieht, indem in einer globalen Variablen die Kennung der Domäne gespeichert wird, in der sie gilt. Der Anwendungscode prüft diese, jedes mal bevor er ein Referenzfeld beschreibt. Entspricht die gespeicherte Kennung der aktuellen Domäne, wird die überschriebene Referenz dem Arbeitsstapel angeboten. Die Deaktivierung geschieht, indem die globale Variable auf eine ungültige Kennung gesetzt wird.

Ablegen der Wurzelmenge auf dem Arbeitsstapel

Die Wurzelmenge besteht aus den lokalen Referenzen auf den Stapeln der Anwendungsfäden, den statischen Referenzfeldern und den Feldern der statisch angelegten Objekte.

Die Referenzen auf den Stapeln der Fäden müssen als erstes abgelegt werden, da diese über keine Schreibbarrieren verfügen. Würden die statischen Referenzfelder vor den Stapeln durchsucht werden, könnte eine Referenz von einer lokalen Variablen in ein Klassenfeld geschrieben werden. Wird der Wert der lokalen Referenz jetzt überschrieben, kann er beim Durchsuchen des Stapelspeichers nicht mehr gefunden werden. Auch die Schreibbarrieren können das nicht verhindern, da diese den Wert des Klassenfeldes erst dann ablegen würden, wenn dieses wieder überschrieben wird.

Wie bereits in Abschnitt 2.1.3 beschrieben wurde, werden die Programmfäden blockiert, solange ihr Stapel durchsucht wird. Dazu fordert der Speicherbereiniger eine OSEK-Ressource an, die die Priorität des OSEK-Task besitzt, der durchsucht werden soll. Dadurch wird eine Prioritätsobergrenze (engl. „priority ceiling“) aktiviert. Diese sorgt dafür, dass dieser (und alle Tasks mit einer niedrigeren Priorität) den Speicherbereiniger nicht verdrängen können. Zur Durchsuchung werden Henderson-Listen [4] verwendet.

Nach den lokalen Referenzen werden die Werte der statischen Referenzfelder abgelegt. Schließlich folgen die Referenzfelder statisch angelegter Objekte. Dabei handelt es sich um die Instanzen von Java-Klassen, die die Einstiegspunkte der Tasks definieren, indem sie die Schnittstelle „Runnable“ implementieren.

Die maximale Ausführungszeit, die zum Durchsuchen der Stapelspeicher benötigt wird, kann, wie in Abschnitt 2.1.3 beschrieben wurde, abgeschätzt werden. Hierzu wird die maximale Tiefe des Stapels beim Aufruf von $WaitEvent()$ bestimmt.

Der Aufwand zur Ablegung der restlichen statischen Referenzen verhält sich linear zu deren Anzahl. Diese ist zur Übersetzungszeit bekannt.

Traversierung des Erreichbarkeitsgraphen

Nachdem die Wurzelmenge bekannt ist, werden nun alle Objekte besucht, die von dieser aus erreichbar sind. Dazu werden die Referenzen nacheinander vom Arbeitsstapel genommen und auf Referenzen durchsucht. Die Referenzen, die dabei gefunden werden, werden wieder auf dem Arbeitsstapel abgelegt.

Durchsuchen der Objekte Beim Durchsuchen wird anhand der Klassenkennung des Objekts unterschieden, ob es sich um ein reguläres Objekt oder um einen Vektor handelt.

Handelt es sich um ein reguläres Objekt, wird es, wie in Abschnitt 3.1.3 beschrieben wurde, durchsucht. Dabei wird jedes besuchte Fragmente in der Bitkarte markiert.

Handelt es sich um einen Vektor, wird überprüft, ob er ein Rückgrat besitzt, das sich im „Räumungsbereich“ befindet (siehe Abschnitt 3.4.2). Ist dies der Fall, wird es in den „Zufluchtsbereich“ kopiert. Des Weiteren werden alle Nutzdatenfragmente des Vektors in der Bitkarte markiert. Handelt es sich um einen Vektor, der Referenzen enthält, werden diese auf dem Arbeitsstapel abgelegt.

Maximale Ausführungszeit Der Aufwand, der zum Markieren aller lebendigen Objekte notwendig ist, hängt von der Anzahl der überlebenden Fragmente ab. Diese ergibt sich daraus, dass kein Objekt mehrfach auf dem Arbeitsstapel abgelegt wird. Im ungünstigsten Fall besitzen alle Objekte nur Referenzfelder. Der Speicherbereiniger muss dann eine maximale Anzahl an Objekten auf ihre Farbe überprüfen.

Der Aufwand zum Replizieren der Rückgrate, richtet sich nach der Anzahl der überlebenden Vektoren. Natürlich spielt auch die Größe ihrer Rückgrate eine Rolle.

3.4.4 Löschphase

Nachdem alle erreichbaren Objekte in der Bitkarte markiert wurden, wird diese nun auf freie Stellen untersucht. Die zu den freien Stellen gehörigen Fragmente werden anschließend in die Freispeicherliste eingefügt. Damit werden Slots die gemeinsam freigegeben werden zu einem Freispeicherblock verschmolzen.

Im Gegensatz zu IdleRoundRobin werden die neuen Freispeicherblöcke nicht mit den bereits vorhandenen Blöcken verschmolzen. Da FragGC Objekte fragmentiert anlegen kann, benötigt er nicht zwingend große zusammenhängende Blöcke. Deshalb muss FragGC Blöcke auch nicht nach Anfangsadressen sortiert in die Freispeicherliste einfügen. Das Verschmelzen könnte trotzdem die durchschnittliche Ausführungszeit des Bereinigers und der Allokationen verbessert.

Vor dem Einfügen in die Freispeicherliste, werden die Blöcke noch genullt, da Java-Objekte erwarten, dass alle Felder mit 0 bzw. *null* initialisiert werden. Der Aufwand des Nullens hängt von der Summe der freigegebenen Slots ab.

Das Durchsuchen der Bitkarte nach freien Fragmenten hat eine Laufzeit, die sich linear zur Größe der Halde verhält. Die Größe der Halde ist für jede Domäne konstant. Damit lässt sich die WCET des Durchsuchens bestimmen. Trotz der Größe der Halde fällt es, durch die Verwendung von Bitkarten, leichtgewichtig aus.

Je mehr Speicherblöcke freigegeben werden, desto höher ist der Aufwand der Löschphase. Die Anzahl der Blöcke, die im ungünstigsten Fall freigegeben werden, ist wie folgt beschränkt: Es können nicht mehr Blöcke als Objektfragmente freigegeben werden. Die maximale Anzahl an Freispeicherblöcken pro Löschphase entspricht der halben Anzahl an Slots der gesamten Halde. Sie tritt dann auf, wenn immer jedes zweite Fragment des gesamten Fragmentspeichers freigegeben wird.

3.4.5 Maximale Ausführungszeit

Wie in den einzelnen Abschnitten beschrieben wurde, ist die maximale Ausführungszeit von drei Gegebenheiten abhängig.

- Die überlebenden Objekte
- Die freigegebenen Objektfragmente
- Die Größe der Halde und andere Konstanten

Die Anzahl der freigegebenen Fragmente bestimmt im ungünstigsten Fall die Anzahl der Speicherblöcke, die in die Freispeicherliste eingehangen werden müssen. Alternativ lässt sich diese Anzahl auch durch die halbe Haldengröße begrenzen.

Der Aufwand der Traversierung (Markierungsphase) verhält sich linear zu der Anzahl der überlebenden Objekte. Ihre Fragmente müssen markiert und ihre Rückgrate kopiert werden. Zur Abschätzung des ungünstigsten Falls kann davon ausgegangen werden, dass auf einer gefüllten Halde alle Objekte überleben. Die WCET richtet sich dann nach der Größe der Halde. Diese Abschätzung ist sehr pessimistisch. In der Regel überleben sehr wenige Objekte. Mit den Kategorien aus Abschnitt 2.2.2 kann das Überleben bestimmter Objekte ausgeschlossen werden.

Die Behandlung überlebender Objektfragmente ist aufwändiger als deren Freigabe. Überlebende Haldenobjekte werden auf dem Arbeitsstapel abgelegt. Schon allein das

ist mit dem Aufwand zur Einreihung eines Freispeicherblocks in die Freispeicherliste vergleichbar. Zusätzlich müssen überlebende Objekte durchsucht werden. Auch das Kopieren eines überlebenden Rückgrats ist aufwändiger, als es im toten Speicherbereich zurückzulassen.

Die Größe der Halde bestimmt die WCET der Markierung der Freispeicherliste und der Durchsuchung der Bitkarte. Der Aufwand zur Freigabe von Freispeicherblöcken lässt sich ebenfalls mit der Haldengröße abschätzen. Die Anzahl der statischen Referenzen und der Referenzen, die beim Aufruf von $WaitEvent()$ auf dem Stapel liegen könnten, fallen beim Ablegen der Wurzelmenge ins Gewicht.

Des Weiteren ist zu beachten, dass ein Objektfragment nicht gleichzeitig lebendig, tot oder verfügbar sein kann. Es wird also entweder bei der Traversierung der lebendigen Objekte bearbeitet, oder in der Löschphase, oder während der Markierung der Freispeicherliste. Die WCET zur Bereinigung des Fragmentspeichers ergibt sich also nicht aus der Summe der Teilabschnitte. Statt dessen wird das Maximum der WCETs berechnet, die zur Freigabe oder dem Löschen der nicht-überlebenden Fragmente benötigt wird. Dieses Maximum wird auf die WCET der Traversierungsphase addiert.

4 Auswertung

4.1 Versuchsaufbau

4.1.1 Die Testanwendung CDj

Um die Änderungen in Codegröße und Ausführungsgeschwindigkeit zu messen, wird die Testanwendung CDj ausgeführt. Hierbei handelt es sich um ein Mitglied der CDx-Familie [8]. Diese Java-Benchmarks stellen harte Echtzeitsysteme dar. Sie dienen zur Erkennung von Flugzeugkollisionen auf einem simulierten Radar. Es wird die Variante „on the go“ getestet. Diese besteht hauptsächlich aus dem Anwendungsfaden „collision detector”, der periodisch angestoßen wird. Dieser misst für jede Periode, wann er seine Arbeit aufgenommen und wann er sie beendet hat. Nachdem eine bestimmte Anzahl von Radarbildern abgearbeitet wurde, beendet sich die Anwendung und die Messergebnisse werden ausgegeben. Es werden jeweils 100 Radarbilder simuliert. Als Haldengröße werden 600 Kibibyte verwendet. In Fällen in denen das zum Überlauf des Hauptspeichers führt, werden 599 Kibibyte verwendet.

4.1.2 Zielarchitektur

Die Tests wurden auf einem Infineon TriCore TC1796 [5] durchgeführt. Als Betriebssystem wurde CiAO mit Revision r1804 verwendet. Als C-Übersetzer wurde der GCC 4.5.2 verwendet. KESOs Revision ist r3967.

Da FragGC ein unüblich hohes Alignment fordert, kam es bei der Tricore-Variante CiAOs zu fehlerhaftem Verhalten. Um diesen Fehler zu korrigieren, reicht es meistens das Alignment in CiAOs Linkerskripts an die Fragmentgröße anzupassen. Eine endgültige Lösung dieses Problems wurde jedoch nicht gefunden.

Bei der x86-Variante CiAOs wurden diese Probleme nicht beobachtet. Ebenso treten auf dem „Trampoline“ [2] keine Probleme auf.

4.1.3 Methode zur Messung der Ausführungszeiten

Um die Ausführungszeiten bestimmter Abschnitte im Speicherbereiniger messen zu können, werden von JINO „Stoppuhren“ angelegt. Hierbei werden in den generierten C-Code Datenstrukturen eingebunden. Diese Speichern folgende Informationen der Stoppuhr:

- Die Summe der Dauer aller gemessenen Zeitspannen
- Die Anzahl der gemessenen Zeitspannen
- Die maximale Dauer der gemessenen Zeitspannen

Um die Zeitspannen zu markieren, die gemessen werden sollen, werden „Start“- und „Stopp“-Makros definiert. An diesen Positionen wird eine Funktion zur Bestimmung des aktuellen Zeitstempels aufgerufen. Durch den zusätzlich eingefügten Code kann sich die Ausführungsgeschwindigkeit der Anwendung leicht verändern.

4.2 Laufzeiten der Testanwendung CDj

Abk.	Speicher-bereinigung	Slot-größe	zus. Vektor Versuche	durchschnittl. Laufzeit [ns]	relativer Durchsatz
fgc16c	FragGC	16	4	66.631.530	71,96%
fgc16i	FragGC	16	-	71.546.936	67,02%
fgc32c4	FragGC	32	4	63.051.262	76,05%
fgc32c0	FragGC	32	0	63.436.795	75,59%
fgc32i	FragGC	32	-	71.657.582	66,92%
irr32	IdleRoundRobin	32	(immer zus.)	48.058.793	99,77%
irr16	IdleRoundRobin	16	(immer zus.)	47.950.067	100,00%

Abbildung 4.1: Durchschnittliche Laufzeiten (CDj)

Abbildung 4.1 zeigt die durchschnittlichen Laufzeiten der Testanwendung CDj unter Verwendung verschieden konfigurierter Speicherbereiniger. Des Weiteren wird der Durchsatz der jeweiligen Konfiguration mit dem der schnellsten ($irr16$) verglichen.

Die Konfigurationen werden nach Speicherbereiniger, Slotgröße und der maximalen Anzahl an Versuchen, die zum Anlegen eines zusammenhängenden Vektors erlaubt sind, unterschieden. Als Speicherbereiniger werden der IdleRoundRobin und FragGC verwendet.

Es ist zu erkennen, dass die FragGC-Varianten auf dem TriCore nur ca. 67% bis 76% des Durchsatzes der IdleRoundRobin-Varianten errreichen. Dies ist dem zusätzlichen Aufwand des fragmentierten Objektformats geschuldet.

Die beiden Varianten, die keine zusammenhängenden Vektoren unterstützen ($fgc16i$ u. $fgc32i$), verfügen mit etwa 67% über den geringsten Durchsatz. Wird das zusammenhängende Vektorformat unterstützt, steigt der Durchsatz auf mindestens 71% an ($fgc16c$, $fgc32c4$ und $fgc32c0$). $fgc16c$ und $fgc32c4$ geben jeder Vektorallokation vier Versuche zusammenhängende Fragmente zu erhalten. $fgc32c0$ versucht nicht zusammenhängende Vektoren anzulegen. Dennoch erreicht $fgc32c0$ einen Durchsatz von 75%

und scheint daher zusammenhängende Fragmente zu erhalten. Dies liegt daran, dass bei der Testanwendung bisher kaum Fragmentierung auftritt. Auch Testläufe mit 3000 Iterationen der x86-Variante bestätigen dies. Die vier Versuche die $fgc32c4$ eingeräumt werden, bringen nur 1%.

Vergleicht man $fgc16c$, das über 16-Byte-Fragmente verfügt, mit der 32-Byte-Variante $fgc32c4$, wird der zusätzliche Aufwand regulärer fragmentierter Objekte deutlich. Objekte können in $fgc16c$ bis zu drei Fragmente umfassen. Der Durchsatz beträgt 72%. Bei $fgc32c4$ passen alle regulären Objekte in ein Fragment. Der Druchsatz beträgt 75%.

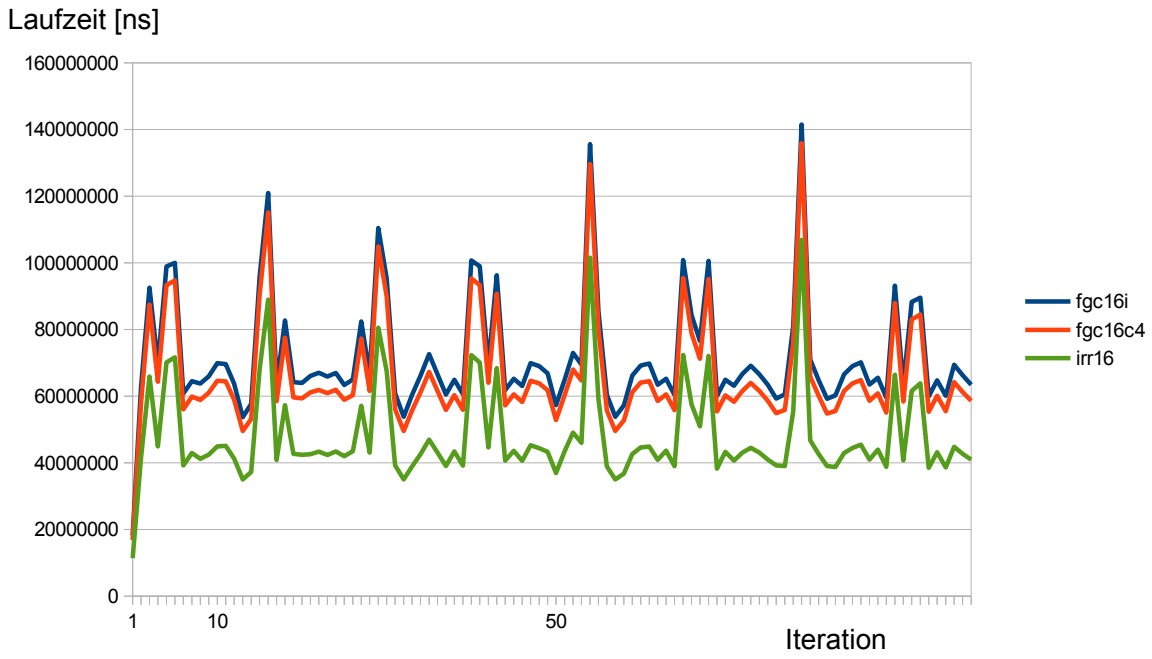

Abbildung 4.2: Laufzeit der jeweiligen Iteration

Abbildung 4.2 zeigt die Laufzeiten der 100 Iterationen. Die Fragmentgröße beträgt 16 Byte. Die Konfiguration $fgc16i$, die keine zusammenhängenden Vektoren unterstützt, ist in jeder Iteration am langsamsten. Damit schätzt sie die maximale Ausführungszeit der Konfiguration $fgc16c$ grob ab.

Aufgrund der in Abschnitt 4.1.2 genannten Probleme, konnten keine weiteren Messungen zur Laufzeit des Speicherbereinigers, der Dauer der Allokationen oder der Unterbrechungssperren durchgeführt werden.

4.3 Codegrößen der Testanwendung CDj

Abbildung 4.3 zeigt die Codegrößen, die für verschiedene Objektformat- und Speicherbereinigerkonfigurationen entstehen. Als Testanwendung wurde der CDj für den TriCore

Abk.	Speicher-bereinigung	Slot-größe	zusammenh. Vektorformat	Textsegment [B]			Daten-segment [B]
				Java-Code	GC	Gesamt	
fgc16c	FragGC	16	X	31.322	6.759	60.363	2.437
fgc16i	FragGC	16		30.518	5.579	58.351	2.709
fgc16ig	FragGC (Gen)	16		30.518	6.399	59.171	2.725
fgc32c	FragGC	32	X	30.862	6.757	59.891	3.333
fgc32i	FragGC	32		30.054	5.575	57.871	4.037
irr32	IdleRoundRobin	16/32	(immer)	24.598	4.406	52.283	1.982

Abbildung 4.3: Codegrößen der verschiedenen Konfigurationen des Speicherbereinigers

Abk.	Speicher-bereinigung	Slot-größe	zusammenh. Vektorformat	Textsegment			Daten-segment
				Java-Code	GC	Gesamt	
fgc16c	FragGC	16	X	27,34%	53,40%	15,45%	22,96%
fgc16i	FragGC	16		24,07%	26,62%	11,61%	36,68%
fgc16ig	FragGC (Gen)	16		24,07%	45,23%	13,17%	37,49%
fgc32c	FragGC	32	X	25,47%	53,36%	14,55%	68,16%
fgc32i	FragGC	32		22,18%	26,53%	10,69%	103,68%
irr32	IdleRoundRobin	16/32	(immer)	0,00%	0,00%	0,00%	0,00%

Abbildung 4.4: Zuwachs im Verhältnis zu IdleRoundRobin

übersetzt. Die Optimierungsstufe des C-Übersetzers betrug „-O2“.

Die Konfigurationen werden nach Speicherbereiniger, Slotgröße und der Unterstützung des zusammenhängenden Vektorformats unterschieden. Als Speicherbereiniger werden der IdleRoundRobin, FragGC und FragGC mit Generationenkonzept verwendet. Als Slotgröße wurden 16 und 32 Byte verwendet. Bei FragGC bestimmt die Slotgröße ebenfalls die Größe der Objektfragmente und hat daher Auswirkungen auf die Codegröße. Auf die Codegröße der IdleRoundRobin-Konfigurationen hat die Slotgröße keine Auswirkung. Weiter unterstützt der IRR zusammenhängende Vektoren immer. Für FragGC kann gewählt werden, ob zusammenhängende Vektoren (Abk.: c) oder eingebettete Rückgrate (Abk.: i) unterstützt werden.

Untersucht wurden die Größe des Text- und des Datensegments. Diese Größen werden im Folgenden genauer beschrieben und verglichen. Der IRR dient dabei dazu, die Kosten, mit denen die Fragmentierungstoleranz erkauft wurde, abzuschätzen. Abbildung 4.4 fasst den Zuwachs an Codegröße im Verhältnis zur IdleRoundRobin-Variante zusammen.

4.3.1 Textsegment

Das Textsegment enthält hauptsächlich den ausführbaren Maschinencode eines Programms. Wie in Abbildung 4.3 zu sehen ist, sind hier drei Größen von Interesse. „Java-Code“ beschreibt die Größe der Funktionen, die aus Java-Methoden generiert werden.

„GC“ beschreibt die Codegröße des Speicherbereinigers. Anschließend wird die Größe des Textsegments der gesamten ELF-Datei genannt. Dieses enthält unter anderem das Textsegment der Java-Methoden und des Speicherbereinigers.

Codegrößen der Java-Methoden

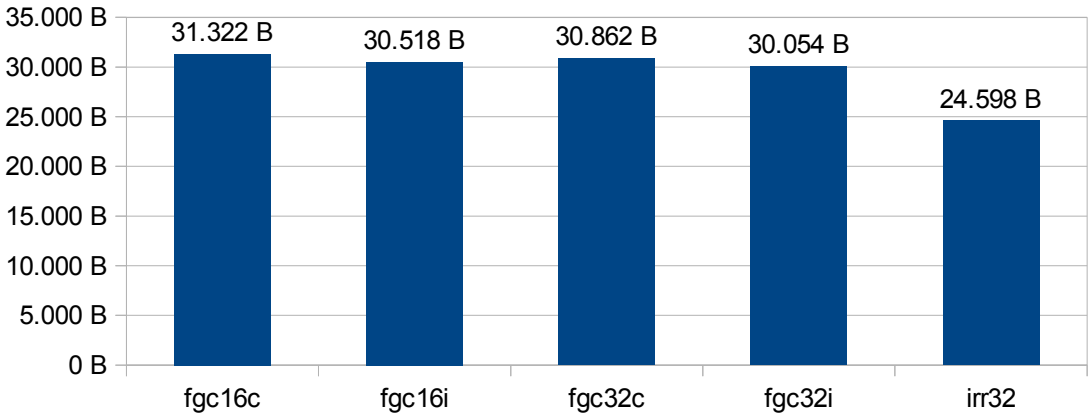

Abbildung 4.5: Codegrößen des Java-Anwendungscodes

Abbildung 4.5 zeigt die Codegröße der aus Java-Methoden entstandenen Funktionen. Die Änderungen in der Größe des Anwendungscodes erklärt sich durch die Änderungen im Objektformat.

Die Konfiguration $fgc16ig$ benutzt das gleiche Objektformat, wie $fgc16i$. Es unterscheidet sich nur dadurch, dass ein generationeller Speicherbereiniger für die Verwaltung der Rückgrate verwendet werden. Dadurch ändert sich die Größe des Java-Anwendungscodes nicht.

Im Vergleich zur Konfiguration $irr32$ die das zusammenhängende bidirektionale Objketformat verwendet, ist ein Codezuwachs von 5456 bis 6724 Byte festzustellen. Die Kosten der Fragmentierungstoleranz betragen für den Anwendungscode des CDj also $22,2\%$ bis $27,3\%$ der ursprünglichen Codegröße.

Zugriff auf Instanzfelder Wird auf Instanzfelder eines fragmentierten Objektes zugegriffen, müssen Indirektionen aufgelöst werden, um zum enthaltenden Fragment zu gelangen. Dies lässt sich anhand der Konfigurationen $fgc16i$ und $fgc32i$ verdeutlichen. $fgc16i$ benutzt eine Fragmentgröße von 16 Byte. $fgc32i$ verwendet 32 Byte. Dadurch passen alle regulären Objekte bei $fgc32i$ in ein Fragment, während bei $fgc16i$ bis zu drei Fragmente benötigt werden. Die höhere Codegröße von $fgc16i$ erklärt sich also durch das Auflösen der genannten Indirektionen. Sie beträgt 464 Byte, was $1,5\%$ der

Codegröße von $fgc32i$ entspricht. Gleiches gilt für den Vergleich von $fgc32c$ mit $fgc16c$. Hier beträgt der Zuwachs 460 Byte, also $1,5\%$. Die Konfiguration $irr32$ verwendet das zusammenhängende bidirektionale Objektformat. Hier entfällt dieser Aufwand immer – unabhängig von der Slotgröße.

Zugriff auf Vektoren Wird fragmentierte Allokation verwendet (FragGC), muss für jeden Vektor unterschieden werden, ob er ein (ausgelagertes) Rückgrat besitzt oder nicht. Dies steigert die Codegröße.

Zusätzlich ist zu unterscheiden, ob zusammenhängende Vektoren oder eingebettete Rückgrate unterstützt werden. Hierzu werden $fgc32c$ und $fgc32i$ verglichen. $fgc32c$ unterstützt zusammenhängende Vektoren. $fgc32i$ unterstützt eingebettete Rückgrate. Wird über ein eingebettetes Rückgrat auf die Elemente des Vektors zugegriffen, unterscheidet sich dies kaum vom Zugriff über ein ausgelagertes Rückgrat. Zugriffe auf zusammenhängende Vektoren müssen zusätzlich neben den Zugriff über das Rückgrat unterstützt werden. Damit werden mit $fgc32c$ im Vergleich zu $fgc32i$ 808 Byte $(2,7\%)$ an zusätzlichem Code erzeugt. Gleiches gilt für den Vergleich von $fgc16c$ mit $fgc16i$. Hier werden 804 Byte $(2,6\%)$ an zusätzlichem Code benötigt.

Bei $irr32$ entfällt der Aufwand der zum Feststellen des Rückgrats und für den Zugriff über das Rückgrat notwendig ist.

Codegröße des Speicherbereinigers

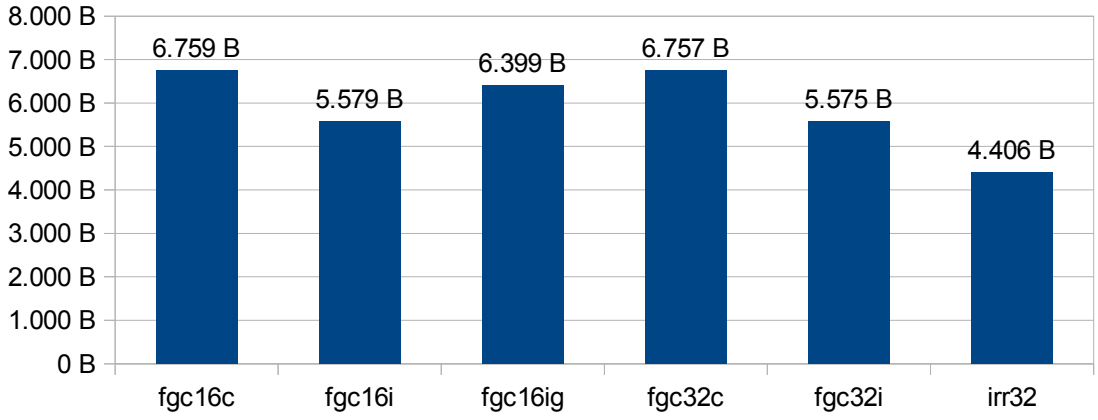

Abbildung 4.6: Codegrößen der Speicherbereiniger

Abbildung 4.6 zeigt die Codegrößen der verschiedenen Varianten des Speicherbereinigers.

FragGC-Varianten, die zusammenhängende Vektoren unterstützen, benötigen eine zusätzliche Allokationsfunktion, die das Anlegen zusammenhängender Fragmente unterstützt. Des Weiteren muss beim Markieren und Durchsuchen von Vektoren unterschieden werden, ob diese zusammenhängend oder fragmentiert vorliegen. Dies ist mit dem Zugriff auf Vektoren im Anwendungscode vergleichbar. FragGC-Varianten ohne zusammenhängende Vektoren benötigen diese Funktionen nicht. Dadurch erklärt sich, warum dies Varianten ($fgc16i$ und $fgc32i$) 1182 Byte weniger einnehmen. Damit nimmt die Behandlung zusammenhängender Vektoren 17% des Codes der entsprechenden Varianten ein.

Die Varianten $fgc16$ und $fgc32$ besitzen in etwa die gleiche Codegröße. Bei einer Fragmentgröße von 32 Byte, passen alle regulären Objekte des CDj in ein Fragment. Dies wird momentan jedoch bei der Generierung des Speicherbereinigers nicht berücksichtigt.

Bei Varianten mit Generationenkonzept ist ebenfalls ein Zuwachs der Codegröße zu verzeichnen. So muss zum Beispiel die Markierungsphase nachbereitet werden. Daher ist bei $fgc16ig$ im Vergleich zu $fgc16i$ ein Zuwachs von 820 Byte (14, 7%) festzustellen.

Zusammenfassend benötigen die fragmentierungstoleranten Speicherbereiniger 1169 bis 2358 Byte mehr Platz im Textsegment als der IdleRoundRobin. Dies entspricht einem Zuwachs von 26, 5% bis 53, 4%.

4.3.2 Datensegment

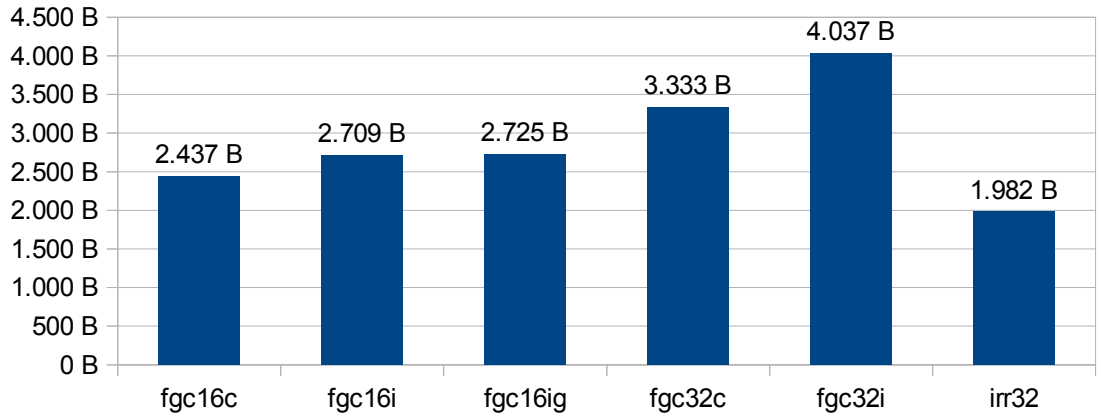

Abbildung 4.7: Größe des Datensegments

Das Datensegment enthält globale beziehungsweise statische Objekte, die vor der Ausführung mit bestimmten Werten initialisiert werden. Dies umfasst unter anderem die Task-Objekte, die den Einstiegspunkt der Anwendungsfäden bestimmen, sowie Alarm-Objekte. Auch Strings werden dort abgelegt (Ablage in „rodata“ kann konfiguriert wer-

den). Diese bestehen aus einem regulären Objekt und einem Vektor, der die eigentliche Zeichenkette enthält. Der CDj verfügt über sechs Task-Objekte, zwei Alarm-Objekte und 28 String-Objekte. Abbildung 4.7 zeigt die Größe des Datensegments in Abhängigkeit von Objektformat und Speicherbereiniger.

Der Zuwachs des Datensegments ist durch den Verschnitt zu erklären, der durch die Adressausrichtung („alignment“) entsteht. Ein String-Objekt ist beispielsweise 8 Byte groß. Bei einer Fragmentgröße von 16 Byte, entstehen 8 Byte Verschnitt. Bei 32-Byte-Fragmenten bleiben 24 Byte ungenutzt. Der Verschnitt des zugehörigen Vektors hängt davon ab, welcher Rest beim Teilen der Vektorlänge durch die Fragmentgröße entsteht. Zwischen $irr32$ und $fgc16c$ findet der Übergang von 4-Byte-Alignment auf 16-Byte-Alignment statt. Die Größe des Datensegments nimmt um 455 Byte (23%) zu. Bei $fgc32c$ steigt das Alignment auf 32 Byte. Im Vergleich zu $irr32$ ist damit eine Zunahme von 1351 Byte (68%) festzustellen.

Neben dem Alignment spielt auch das Vektorformat eine Rolle. Werden keine zusammenhängenden Vektoren unterstützt, müssen die statischen Vektoren fragmentiert angelegt werden. Damit wird neben dem Platz für die Nutzdaten auch noch Platz für das Rückgrat benötigt. Bei 16-Byte-Fragmenten ($fgc16c$ und $fgc16i$) steigt der Platzbedarf um 272 Byte (11%). Bei 32-Byte-Fragmenten ($fgc32c$ und $fgc32i$) steigt er sogar um 704 Byte (21%).

Hier bietet sich noch Optimierungspotential. Für einen Vektor, der aus 32-Byte-Fragmenten besteht, würden eigentlich weniger Rückgrateinträge benötigt werden. Es wurde jedoch Rücksicht auf das Alignment des ersten Nutzdatenfragments genommen. Da sich das Rückgrat zwischen Objektkopf und Nutzdaten befindet, entsteht auch hier Verschnitt. Da statische Vektoren, die primitive Daten enthalten, vom Speicherbereiniger nicht weiter untersucht werden, könnte auf das Alignment verzichtet werden.

Vergleicht man $fgc16i$ und $fgc16ig$ fällt auf, dass der Speicherbereiniger mit Generationionenkonzept 16 Byte mehr zur Speicherung seines Zustandes benötigt.

Zusammenfassend benötigt das Datensegment bei fragmentierungstoleranten Konfigurationen 455 bis 2055 Byte mehr Speicherplatz. Dies entspricht einem Zuwachs von 23% bis 104%.

4.4 Untersuchung der Laufzeit des Speicherbereinigers

In diesem Abschnitt wird die Laufzeit des Speicherbereinigers genauer untersucht. So wird die Aussage, dass die Speicherbereinigung aufwändiger ist, je mehr Objekte überleben, überprüft. Dabei wird auch die WCET, die zum Markieren der überlebenden Objekte notwendig ist, abgeschätzt. Des Weiteren wird die Freispeicherliste fragmentiert. Dadurch kann die WCET, die zur Markierung der Freispeicherliste und zum Einhängen von Freispeicherblöcken notwendig ist, abgeschätzt werden.

Hierbei liegt das Augenmerk auf dem Fragmentspeicher. Beim Markieren eines Vektors muss, zusätzlich zu den hier gemessenen Werten, das Kopieren des Rückgrates hinzu-

gezählt werden. Mit dem Löschen eines toten Rückgrats ist kein Aufwand verbunden.

4.4.1 Testanwendung: Abwechselnd verkettete Liste

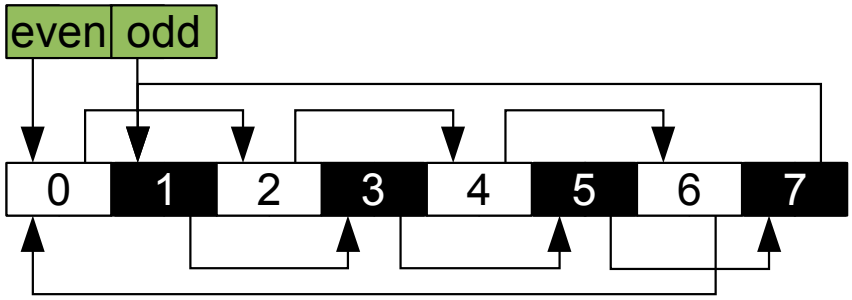

Abbildung 4.8: Sich abwechselnde vektettete Listen

Diese Testanwendung legt so lange Listenelemente an, bis der Speicherplatz der Halde erschöpft ist. Sie verfügt über zwei verkettete Listen. Die Elemente werden abwechselnd in die Liste der „geraden“ und „ungeraden“ Elemente eingereiht. Abbildung 4.8 verdeutlicht, wie die Elemente auf der Halde angelegt werden. Ein Listenelement nimmt dabei einen Slot der Halde ein.

Nachdem die beiden Listen angelegt wurden, wird der Speicherbereiniger aktiviert. Hierbei überleben alle Elemente. Dies dient dazu die WCET, die zum Markieren überlebender Objekte notwendig ist, grob abzuschätzen. Anschließend gibt es zwei Möglichkeiten, die Listen zu löschen:

1. Es wird erst die „gerade“ Liste gelöscht, dann die „ungerade“
2. Beide Listen werden gleichzeitig gelöscht

Bei der ersten Möglichkeit wird jeweils jeder zweite Slot freigegeben. Damit muss die maximale Anzahl an Freispeicherblöcken in die Freispeicherliste eingereiht werden. Könnte ein Block mehr freigegeben werden, würde er mit den beiden angrenzenden Blöcken verschmolzen werden. Die zweite Möglichkeit zeigt wie viel Zeit für das Nullen des Speichers verwendet wird.

Getestet wird neben FragGC auch IdleRoundRobin. Dadurch soll unter anderem überprüft werden, welche Auswirkungen der Verzicht auf das Verschmelzen der Freispeicherblöcke hat. Die Größe der Halde beträgt immer 640 Kibibyte.

4.4.2 Ergebnisse

Die Abbildungen 4.9 ff zeigen die Ausführungszeit der Speicherbereinigung. Diese ist nach den in Abschnitt 3.4 beschriebenen Phasen unterteilt. So teilt sie sich in die

	FragGC		IRR	
Ausführungszeit	903.404.333	ns	777.023.867	ns
Markierungsphase	902.237.266	ns	776.013.000	ns
Freispeicherliste	49.053	ns	48.987	ns
Fadenstapel	40.333	ns	36.947	ns
Wurzelmenge	72.400	ns	76.320	ns
Traversierung	902.147.880	ns	775.927.066	ns
Löschphase	1.058.294	ns	901.347	ns
Bitkarte zurücksetz.	108.773	ns	109.520	ns
Überlebend	640.016	B	640.016	B
Freigegeben	96	B	64	B
Markierungsrate	709	kB/s	824	kB/s
Löschrate	90	kB/s	71	kB/s

Abbildung 4.9: Alle Objekte überleben

Ausführungszeit der Lösch- und Markierungsphase auf. Die Markierungsphase unterteilt sich weiter in das Markieren der Freispeicherliste, das Durchsuchen der Fadenstapel, das Lesen der restlichen Wurzelmenge und das Besuchen und Markieren aller lebendigen Objekte. Des Weiteren ist zu sehen, wie viel Speicher freigegeben wurde und wie viel überlebt hat. Zur groben Bewertung der Geschwindigkeit, mit der Objekte markiert oder gelöscht werden, wird eine „Markierungs“- und eine „Löschrate“ berechnet. Diese bezeichnenen das Verhältnis zwischen überlebendem bzw. freigegebenem Speicherplatz zur Dauer der Traversierungs- bzw. Löschphase. Welcher Abschnitt die Dauer der Bereinigungsphase bestimmt, fällt meist schon auf Grund der Anzahl der Ziffern der Ausführungszeit ins Auge.

Abbildung 4.9 zeigt die erste Speicherbereinigungsphase nach dem Anlegen der Objekte. Keines der Objekte wird freigegeben. Alle Objekte müssen markiert werden. Die Traversierung der Objekte bestimmt die Gesamtausführungszeit maßgeblich. Die Ausführungszeit der Löschphase wird vom Durchsuchen der Bitkarte nach Freispeicherblöcken bestimmt. Durch das fragmentierte Objektformat benötigt FragGC etwas mehr Zeit zum Traversieren der Objekte. So erreicht er nur 86% der Markierungsrate des IRR.

Abbildung 4.10 zeigt die Speicherbereinigungsphase in der alle „geraden“ Objekte freigegeben werden. Der Aufwand der Traversierungsphase, die nun nur noch die „ungeraden“ Objekte besuchen muss, hat sich halbiert. In der Löschphase werden nun alle „geraden“ Objekte in die Freispeicherliste eingefügt. Daher fällt diese nun auch ins Gewicht. Es ist zu erkennen, dass es weniger als halb so teuer (39%) ist einen Block freizugeben, als ihn nach Referenzen zu durchsuchen. Der IRR benötigt zur Freigabe der Blöcke etwas länger, da er diese nach Adressen sortiert in die Liste einfügt und gegebenenfalls verschmilzt.

In Abbildung 4.11 läuft der Speicherbereiniger, während die „geraden“ Objekte bereits freigegeben wurden und die „ungeraden“ noch lebendig sind. Das Traversieren

	FragGC		IRR	
Ausführungszeit	625.782.293	ns	666.054.947	ns
Markierungsphase	451.095.973	ns	388.174.254	ns
Freispeicherliste	48.813	ns	48.707	ns
Fadenstapel	40.347	ns	36.973	ns
Wurzelmenge	64.013	ns	66.107	ns
Traversierung	451.006.813	ns	388.088.574	ns
Löschphase	174.577.574	ns	277.771.160	ns
Bitkarte zurücksetz.	108.746	ns	109.533	ns
Überlebend	320.016	B	320.112	B
Freigegeben	320.288	B	320.128	B
Markierungsrate	709	kB/s	824	kB/s
Löschrate	1834	kB/s	1152	kB/s

Abbildung 4.10: Löschung der „geraden“ Objekte

	FragGC		IRR	
Ausführungszeit	622.729.840	ns	744.796.506	ns
Markierungsphase	621.525.360	ns	524.830.773	ns
Freispeicherliste	170.475.534	ns	151.833.506	ns
Fadenstapel	40.386	ns	36.480	ns
Wurzelmenge	63.894	ns	60.000	ns
Traversierung	451.009.440	ns	372.960.787	ns
Löschphase	1.095.760	ns	219.379.147	ns
Bitkarte zurücksetz.	108.720	ns	109.520	ns
Überlebend	320.016	B	320.144	B
Freigegeben	96	B	32	B
Markierungsrate	514	kB/s	609	kB/s
Löschrate	87	kB/s	0	kB/s

Abbildung 4.11: „Ungerade“ Objekte überleben, „gerade“ wurden bereits gelöscht

der Überlebenden benötigt so viel Zeit wie vorher (Abb. 4.10). Auffällig ist die hohe Ausführungszeit, die zur Markierung der Freispeicherliste notwendig ist. Diese entspricht in etwa der, der vorherigen Löschphase. Der Grund dafür ist die fragmentierte Freispeicherliste, über die iteriert werden muss. Die Löschphase läuft bei FragGC, dadurch dass kaum Objekte freigegeben werden, schnell ab. IRR benötigt hierzu bedeutend länger. Dies liegt vermutlich daran, dass ein kleines Objekt freigegeben wird. Der dadurch entstandene Freispeicherblock muss in der fragmentierten Freispeicherliste an der richtigen Stelle eingefügt werden und verschmolzen werden.

In Abbildung 4.12 werden nun die „ungeraden“ Objekte freigegeben. Die Markierung der Freispeicherliste läuft wie vorher ab. Die Freigabe der „ungeraden“ Objekte ist bei FragGC ebenso aufwändig wie die Freigabe der „geraden“ (Abb. 4.10). Im Gegensatz

	FragGC		IRR	
Ausführungszeit	345.102.133	ns	571.919.920	ns
Markierungsphase	170.547.613	ns	127.276.000	ns
Freispeicherliste	170.375.400	ns	127.154.080	ns
Fadenstapel	40.333	ns	36.947	ns
Wurzelmenge	55.574	ns	55.853	ns
Traversierung	131.880	ns	84.973	ns
Löschphase	174.445.840	ns	444.534.413	ns
Bitkarte zurücksetz.	108.680	ns	109.507	ns
Überlebend	16	B	304	B
Freigegeben	320.288	B	320.032	B
Markierungsrate	0	kB/s	2	kB/s
Löschrate	1836	kB/s	719	kB/s

Abbildung 4.12: Löschung der „ungeraden“ Objekte

dazu fügt IRR die neuen Freispeicherblöcke sortiert ein und verschmilzt sie. In diesem speziellen Fall muss eine maximale Anzahl an Freispeicherblöcken (nämlich jeder einzelne Slot) zu einem großen Block verschmolzen werden, der die gesamte Halde umfasst. Die Ausführungzeit fällt dementsprechend hoch aus und übertrifft die Zeit, die zur Markierung der lebendigen „ungeraden“ Objekte notwendig war.

	FragGC		IRR	
Ausführungszeit	341.918.480	ns	1.162.573	ns
Markierungsphase	340.736.320	ns	960.906	ns
Freispeicherliste	340.564.027	ns	838.893	ns
Fadenstapel	40.346	ns	36.973	ns
Wurzelmenge	55.640	ns	55.867	ns
Traversierung	131.947	ns	85.040	ns
Löschphase	1.073.387	ns	91.960	ns
Bitkarte zurücksetz.	108.773	ns	109.707	ns
Überlebend	16	B	192	B
Freigegeben	96	B	64	B
Markierungsrate	0	kB/s	199	kB/s
Löschrate	89	kB/s	695	kB/s

Abbildung 4.13: Bereinigung nach der Fragmentierung der Freispeicherliste

In Abbildung 4.13 zeigen sich die Vorteile des Verschmelzens von Freispeicherblöcken. Es befinden sich weder „gerade“ noch „ungerade“ Objekte auf der Halde. Bei FragGC ist nun jeder Slot ein eigener Freispeicherblock. Um die Freispeicherliste zu markieren muss also über jeden einzelnen Slot iteriert werden. Entsprechend hoch entfällt die Ausführungszeit. IRR hat alle Freispeicherblöcke verschmolzen. Zur Markierung der Freispeicherliste muss nur ein Block besucht werden.

Anschließend wird der Speicher wieder mit Objekten gefüllt. Wird nun der Speicherbereiniger ausgelöst, verhält dieser sich wieder wie in Abbildung 4.9.

	FragGC		IRR	
Ausführungszeit	14.252.067	ns	91.093.133	ns
Markierungsphase	219.787	ns	171.080	ns
Freispeicherliste	47.614	ns	49.187	ns
Fadenstapel	40.333	ns	37.000	ns
Wurzelmenge	55.573	ns	55.773	ns
Traversierung	131.840	ns	84.893	ns
Löschphase	13.923.560	ns	90.807.067	ns
Bitkarte zurücksetz.	108.720	ns	114.986	ns
Überlebend	16	B	192	B
Freigegeben	640.480	B	640.384	B
Markierungsrate	72	kB/s	1122	kB/s
Löschrate	45999	kB/s	7052	kB/s

Abbildung 4.14: Freigabe aller Objekte

Abbildung 4.14 zeigt das Zeitverhalten des Speicherbereinigers, wenn alle Objekte freigegeben werden. Erwartungsgemäß fällt die Traversierungsphase nicht ins Gewicht, schließlich müssen kaum Objekte besucht werden. Auch die Löschphase läuft schneller ab, als bei der Freigabe der „geraden“ Objekte, da jetzt nur ein großer Freispeicherblock in die Liste eingefügt werden muss. Die Ausführungszeit ergibt sich aus dem Durchsuchen der Bitkarte nach dem Freispeicherblock und aus dem Nullen des Blocks.

4.4.3 Zusammenfassung

Maximiale Ausführungszeit nach Anzahl der überlebenden Objekte Die WCET hängt von der Zeit ab, die zur Markierung der Freispeicherliste, der Markierung lebendiger Objekte und der Freigabe von Blöcken notwendig ist. Ein Slot kann entweder frei, lebendig oder tot sein und somit in eine dieser drei Zeitspannen fallen. Die höchsten Kosten werden von lebendigen Objekten verursacht. In Abschnitt 2.2.2 wird erklärt, wie das Überleben von Objekten ausgeschlossen werden kann. Kann weder das Überleben der „geraden“ noch der „ungeraden“ Objekte ausgeschlossen werden, muss mit einer WCET von ca. $904ms$ gerechnet werden (Abb. 4.9). Kann das Überleben einer der beiden Listen ausgeschlossen werden, kann mit einer WCET von ca. $626ms$ gerechnet werden (Abb. 4.10). Kann das Überleben beider Listen ausgeschlossen werden, kann von einer WCET von $346ms$ ausgegangen werden, um die 640 Kibibyte große Halde zu bereinigen. Hierbei handelt es sich um des Maximum der Ausführungszeit, die zur Löschung (Abb. 4.12) und zur Markierung der Freispeicherliste (Abb. 4.13) nötig ist. Es kann nicht mit $15ms$ (Abb. 4.14) gerechnet werden, solange nicht ausgeschlossen werden kann, dass

eine Fragmentierung der Halde auftritt. Dies wäre dann der Fall, wenn zwischen zwei Bereinigungsphasen immer die gesamte Halde mit den beiden Listen belegt wird.

Natürlich handelt es sich nur um eine grobe Abschätzung, bei der bspw. Cacheeffekte nicht beachtet werden. Des Weiteren sollte ein Echtzeitsystem überdimensioniert werden, um etwaige Schwankungen in der Laufzeit verkraften zu können.

Im Vergleich zu IRR erreicht FragGC nur 86% der Markierungsrate, auf Grund des Objektformats. IRR erreicht in den vergleichbaren Fällen 62% von FragGCs Löschrate, da auf die Freispeicherblockverschmelzung verzichtet wurde.

Auswirkungen des Verzichts auf die Verschmelzung der Freispeicherliste Ob es vorteilhaft ist, auf die Verschmelzung der Freispeicherblöcke zu verzichten, ist hier nicht klar erkennbar. Wird sie verwendet, wird die WCET der Freispeicherlistenmarkierung von der Häfte statt von der gesamten Halde bestimmt. Jedoch müsste zusätzlich überprüft werden, wann die Kosten der Löschphase wie in Abb. 4.12 ansteigen können. Dazu müsste der Algorithmus, der zum Verschmelzen der Blöcke verwendet wird, genauer untersucht werden.

Wie in Abb. 4.13 zu sehen ist, wird die WCET der Markierung nur von der Größe der gesamten Halde beschränkt, wenn aneinanderliegende Freispeicherblöcke nicht verschmolzen werden. Würde beispielsweise nur ein Viertel der Objekte überleben, würde die WCET der Freispeicherlistenmarkierung die der Löschphase übersteigen. Schließlich wird der Aufwand der Löschphase maßgeblich von der halben Haldengröße beschränkt.

4.5 Fragmentierungstoleranz und Allokation

Ob Objekte trotz einer Fragmentierung des Freispeichers angelegt werden können, lässt sich mit einem einfachen Beispiel überprüfen. Hierzu wird eine Halde mit ca. 4 Kibibyte Fragmentspeicher gewählt. Anschließend werden vier Vektoren der Größe 1 Kibibyte angelegt, wovon jeder zweite wieder freigegeben wird. Nun wird versucht einen Vektor der Größe 2 Kibibyte anzulegen. FragGC kann dies erreichen, indem er den Vektor auf die zwei entstandenen Blöcke verteilt. Andere Speicherbereiniger lösen eine „OutOfMemory“-Ausnahme aus.

5 Schluss

5.1 Zusammenfassung

Das Ziel dieser Arbeit ist die Implementierung einer fragmentierungstoleranten, echtzeitfähigen, automatischen Speicherbereinigung.

Die Fragmentierungstoleranz wurde durch die Einführung des fragmentierten, bidirektionalen Objektformats erreicht. Dieses erlaubt es Objekte auf einzelne Fragmente verteilt anzulegen. Um eine vertretbare obere Schranke für die Ausführungszeit von Vektorzugriffen zu erlangen, werden die Fragmente des Vektors durch ein Rückgrat verwaltet. Die Verwaltung der Rückgrate, die eine variable Länge besitzen können, wird von einem replizierendem Speicherbereiniger übernommen. Zur Verwaltung der Fragmente wurden die Strukturen des bereits vorhandenen Speicherbereinigers IdleRoundRobin genutzt.

FragGC bietet obere Schranken für folgende Abschnitte der Java-Anwendung:

- Zugriffe auf Instanzfelder (Abs. 3.1.3)
- Zugriffe auf Vektorelemente (Abs. 3.1.4)
- Allokation regulärer Objekte (Abs. 3.3.4)
- Allokation von Vektoren (Abs. 3.3.5)

Des Weiteren erlaubt FragGC die Bestimmung der WCET der Bereinigungsphase (Abs. 3.4.5). Hierzu wurde beleuchtet, wie das Überleben bestimmter Objekte ausgeschlossen werden kann (Abs. 2.2.2). Schließlich wird die WCET der Bereinigung häuptsächlich von überlebenden Objekten bestimmt (Abs. 4.4).

Durch die Fragmentierungstoleranz und das Wissen über die WCET kann der Speicherbereiniger in ein Echtzeitsystem eingeplant werden. Weil Fragmentierung bei der Allokation nicht mehr zum Tragen kommt, können Speicheranforderungen immer erfüllt werden, solange genügend freier Speicherplatz vorhanden ist. Damit genügend Speicher frei ist, muss die Allokationsrate der Anwendung beachtet werden (Abs. 2.2.4). Diese lässt sich ermitteln, da der maximale Speicherverbrauch für jedes Objekt errechnet werden kann (Abschnitte 3.3.4 u. 3.3.5). Dem Speicherbereiniger muss dann, entsprechend der Allokationsrate und seiner WCET, genügend Schlupfzeit gegeben werden.

Trotz der Fragmentierungstoleranz bezüglich der Allokation ist FragGC nicht völlig unempfindlich gegenüber Fragmentierung. Werden zusammenhängende Vektoren unterstützt kann die durchnittliche Ausführungzeit der Vektorzugriffe sinken, wenn

Fragmentierung auftritt (Abs. 3.1.4). Dies ist vertretbar, da nur die durchschnittliche Ausführungszeit betroffen ist. Die WCET wird davon nicht beeinflusst. Die Ausführungszeit der Bereinigungsphase wird ebenfalls von der Fragmentierung der Freispeicherliste beeinflusst. Dies geschieht sowohl in der Löschphase als auch bei er Markierung der Liste (Abs. 4.4).

Die Fragmentierungstoleranz wurde auf Kosten von Laufzeit und Codegröße erkauft. FragGCs Code benötigt 27% bis 53% mehr Platz als der IdleRoundRobins. Die Größe des Anwendungscodes der Testanwendung *CDj* steigt um etwa 22% bis 27%. Der Durchsatz der Testanwendung sinkt auf etwa 67% bis 76%.

Letzten Endes wurde ein fragmentierungstoleranter, echtzeitfähiger, automatischer Speicherbereiniger für die KESO Multi-JVM implementiert.

5.2 Weiterführende Arbeiten

5.2.1 Steuerung der Speicherverwaltung

In einem harten Echtzeitsystem ist es nicht die Aufgabe der JVM, dem Anwendungsentwickler die Speicherverwaltung abzunehmen. Die Aufgabe der JVM ist es, Typsicherheit zu garantieren. Das bedeutet in Bezug auf die Speicherbereinigung, dass dem Entwickler typsichere Methoden zur Speicherfreigabe an die Hand gegeben werden müssen.

Daher sollte es dem Entwickler ermöglicht werden, die Einplanung des Speicherbereinigers genauer auf die Anwendung zuzuschneiden. Beispielsweise könnte der Speicherbereiniger periodisch oder nach der Abarbeitung bestimmter Aufgaben aktiviert werden. Im Rahmen dieser Arbeit wurde die explizite Auslösung durch einen Anwendungsfaden implementiert.

Um die maximale Ausführungszeit der Speicherbereinigung bestimmen zu können, muss klar sein, welche Objekte die Speicherbereinigung überleben können. Die Freigabe von Objekten zur nächsten Bereinigungsphase kann teilweise vom Übersetzer zugesichert werden. Bestimmten Objekten können Regionen zugewiesen werden, in denen sie existieren (engl. „region inference“ [16]). Kann die Speicherbereinigung innerhalb dieser Regionen nicht gestartet werden, sind die zugehörigen Objekte kurzlebig. Auch die Fluchtanlyse, die zur Stapelallokation genutzt wird, ist dazu fähig [9]. Zudem muss darauf geachtet werden, dass der Übersetzer die Lebensspanne von Objekten durch Umformungen nicht versehentlich verlängert.

Alternative Allokationsmechanismen können die maximale Ausführungszeit von Speicheranfragen verringern. Hier sind wieder Stapel- und Regionenallokation zu nennen (siehe Abschnitt 2.2.2). Dadurch, dass eine kleinere Halde verwendet werden könnte, sinkt damit auch der Aufwand der Löschphase. Jedoch wird der Aufwand der schwergewichtigen Markierungsphase nicht vermindert, da noch immer alle lebendigen Objekte nach Referenzen durchsucht werden müssen. Zumindest sofern sie die automatische Speicherbereinigung nicht gänzlich überflüssig machen.

5.2.2 Große reguläre Objekte

Bisher unterstützt FragGC Objekte mit bis zu sechs Fragmenten (Abs. 3.1.3). Passen Objekte nicht in diese sechs Fragmente, muss die Fragmentgröße erhöht werden. Dies kann zu Verschnitt bei kleineren Objekten führen, die möglicherweise häufiger angelegt werden. Daher könnten große reguläre Objekte implementiert werden. Diese verzweigen ab einem bestimmten Fragment in ein Rückgrat. Alle weiteren Fragmente werden über dieses Rückgrat adressiert. Damit könnten die Indirektionen, die bei Instanzfeldzugriffen durchschritten werden müssen, auch für große Objekte gering gehalten werden.

Literaturverzeichnis

[1] H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27:66–70, 1992.

[2] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline - an open source implementation of the osek/vdx rtos specification. In *11th Int. Conf. on Emerging Technologies and Factory Automation (ETFA'06)*, Prague, Tchèque, République, Sept. 2006. IEEE.

[3] E. M. Gagnon and L. J. Hendren. Sablevm: A research framework for the efficient execution of java bytecode. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 3–3, Berkeley, CA, USA, 2001. USENIX Association.

[4] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 150–156, New York, NY, USA, 2002. ACM.

[5] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TC1796 User's Manual (V2.0)*, July 2007.

[6] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. pages 275–288, 2002.

[7] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

[8] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cdx: A family of real-time java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM.

[9] C. Lang. Improved Stack Allocation Using Escape Analysis in the KESO Multi-JVM. Bachelorarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 2012.

[10] OSEK group. *OSEK/VDX Operating System Specification 2.2.3*, Feb. 2005.

[11] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-Tolerant Real-Time Garbage Collection. 2010.

[12] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM.

[13] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. A JVM for Soft-Error-Prone Embedded Systems. In ACM, editor, *Proceedings of the 14th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 21–32, 2013.

[14] M. Stilkerich. An OSEK Operating System Interface and Memory Management for Java. Diplomarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 2006.

[15] M. Strotz. Semi-Automatische Anwendung von graduellem software-basierten Speicherschutz in der KESO Multi-JVM. Bachelorarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 2012.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[17] C. W. A. Wawersich. KESO: Konstruktiver Speicherschutz für Eingebettete Systeme. Dissertation, Friedrich-Alexander Universität Erlangen-Nürnberg, 2009.

[18] C. Wimmer. Automatic Object Inlining in a Java Virtual Machine. Dissertation, Johannes Kepler University Linz, 2008.

Abbildungsverzeichnis