Fakultät für Mathematik, Informatik und Naturwissenschaft
Fachbereich Informatik

Ausarbeitung zum Seminar „Effiziente Programmierung in C“ im Wintersemester 2013/2014

Reference Counting

vorgelegt von

Moritz Gaack

Lehrende:

Nathanael Hübbe
Michael Kuhn

05. Dezember 2013

Inhalt

1 Einleitung

1.1 Motivation

Der enorme Fortschritt im Bereich Software-Development und –Engineering ermöglicht Programmierern heutzutage die Gestaltung komplexer IT-Systeme. Bei der Entwicklung solcher Systeme liegt der Fokus in der Regel auf der Entwicklung neuer Funktionalitäten oder Systemschnittstellen und weniger auf effizientem Speichermanagement. Insbesondere der fahrlässige Umgang mit korrekter Speicherverwaltung führt vermehrt zu Applikationsabstürzen oder miserabler Systemperformanz.

Diverse Programmiersprachen wie beispielsweise Java oder C# unterstützen Softwareentwickler mit Hilfe integrierter automatischer Speicherverwaltung. Der Einsatz sogenannter Tracing Garbage Collectors kann jedoch nach wie vor zu Performanzeinbußen führen, da Programmierer keinen Einfluss darauf haben, wann der Speicher wieder freigegeben wird. Ein weitverbreitetes, alternatives Konzept, das effizientes Speichermanagement ermöglicht, ist Reference Counting. Dieses kann auch in nicht objektorientierten Sprachen wie beispielsweise C eingesetzt werden.

1.2 Ziel

Das Ziel des Moduls „Seminar: Effiziente Programmierung in C“ ist das Kennenlernen verschiedener Techniken und Methoden zum Zeitsparen beim Programmieren sowie eine intensive Recherche und Bewertung von Nutzungsmöglichkeiten. Das konkrete Thema ist den Studierenden freigestellt. Ich habe mich für das vorgeschlagene Thema „Reference Counting“ entschieden. Das erste Ziel ist eine gründliche Recherche zu dem Vortragsthema durchzuführen und themenspezifische Fragen zu finden. Das Hauptziel ist die detaillierte Betrachtung, sowie die Bewertung hinsichtlich effizienter Programmierung. Am Ende des Vortrags sind Fragen des Plenums zu beantworten.

1.3 Aufbau

Kapitel 2 veranschaulicht an einem konkreten Beispiel die allgemeine Problemstellung des Speichermanagements. Zudem werden klassische Lösungsszenarien grob beschrieben und deren Schwachstellen aufgezeigt.

Daraufhin gibt Kapitel 3 einen umfassenden Überblick über das Thema *Reference Counting* orientiert am „Advanced Memory Management Programming Guide“ von Apple gemäß der Dokumentation [4]. Neben dem klassischen Konzept werden auch grundlegende Regeln des Speichermanagements, typische Anwendungsbereiche und aktuelle Handlungsfelder beschrieben.

In Kapitel 4 wird die Effizienz dreier Speichermanagementimplementierungen in der Programmiersprache *c* untersucht. Vor allem wird gezeigt, welchen Einfluss die Integration des *Reference Counting*-Konzeptes auf die Performanz hat.

In Kapitel 5 wird diese Ausarbeitung zusammengefasst.

2 Hintergrund

In diesem Kapitel wird zunächst an einem konkreten Beispiel die allgemeine Problemstellung des Speichermanagements erläutert. Anschließend werden universelle Lösungsszenarien eingeführt sowie deren Schwachstellen aufgezeigt.

2.1 Allgemeine Problemstellung

Das exemplarische Codebeispiel [1] veranschaulicht zwei Formen des inkorrekten Speichermanagements. Zum einen das Freigeben oder Überschreiben von Speicherblöcken, die noch in Verwendung sind und zum anderen das Nicht-Freigeben von Speicherblöcken. Ersteres wird als *Memory Corruption* bezeichnet und resultiert in Applikationsabstürzen oder korrupten Benutzerdaten. Speicherblöcke, die hingegeben nicht freigegeben werden, werden als *Memory Leaks* bezeichnet. Sie sind für das System blockiert und können nicht erneut allokiert werden. Dies hat auch zur Folge, dass die Applikation immer mehr Speicher benötigt. Die Konsequenz ist eine schlechte Systemperformanz sowie eine Beeinträchtigung der Systemstabilität.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    int * p1;
    int * p2 = (int *)calloc(1, sizeof(int));
    int * p3 = (int *)calloc(1, sizeof(int));

    p3 = p2;   // Original p3 has been lost.

    free(p1);  // Memory corruption.
    free(p2);  // p3 is invalid now.
    free(p3);  // Memory corruption.

    return 0;
}
```

Die c-Hauptroutine enthält drei Deklarationen für Variablen vom Typ Integer Pointer. Die erste Variable *p1* wird nicht initialisiert und zeigt auf einen beliebigen Speicherbereich. Für die Variablen *p2* und *p3* wird jeweils Speicher für ein Integer-Feld allokiert.

Durch die Zuweisung $p3 = p2$ wird die Referenz auf den zuvor reservierten Speicherblock von *p3* verloren. Im weiteren Programmverlauf kann das System diesen Speicherblock nicht wiederverwenden, das heißt, dass diese Ressource bis zum Programmende blockiert ist. Auf den allokierten Speicherblock der Variablen *p2* zeigen nun zwei Zeiger.

Im letzten Abschnitt der *main*-Routine wird der Speicher freigegeben. Die Freigabe der Variablen *p1* kann zu Memory Curroption führen, da der Zeiger nicht initialisiert worden ist und somit versucht wird, einen beliebigen Speicherbereich freizugeben. Der zweite *free*-Aufruf ist valide, da lediglich der zuvor resevierte Integer-Speicherblock freigegeben wird. Der Auf-

ruf *free(p3)* entspricht der Freigabe eines bereits freigegebenen Speicherblocks und ist somit nicht zulässig. Das Freigeben der Variable *p3* führt zwangsläufig zu Memory Curruption.

2.2 Klassische Konzepte für Speichermanagement

Um den, in Abschnitt 2.1 eingeführten, Problemen vorzubeugen, existieren allgemeingültige Speichermanagementkonzepte. Bei der manuellen Speicherverwaltung sind Programmierer dafür verantwortlich alle allokierten Ressourcen auch wieder freizugeben, jedoch keine Ressource freizugeben, die noch in Verwendung ist. Komplexe Systeme mit komplexen Schnittstellen, vielen Entwicklern und Parallelisierungen in Programmen enthalten mannigfaltige Fehlerquellen, sodass ein einwandfreies Speichermanagement in der Regel, ohne weitere Hilfsmittel, nicht gewähreistet werden kann.

Ein typisches, weitverbreitetes Hilfsmittel ist der Einsatz von *Garbage Collections*[1]. Nach [8] wird zwischen konservativer, automatischer und nicht-konservativer automatischer *Garbage Collection* unterschieden. Praxisrelevant sind jedoch ausschließlich nicht-konservative Ansätze, da diese eine zuverlässige Erkennung nicht-referenzierter Speicherbereiche erlauben. Die nicht-konservativen Ansätze unterscheiden zudem zwischen *Tracing Garbage Collectors* und *Reference Counting*.

Wann der Speicher in *Garbage Collections* wieder freigegeben wird ist implementierungsabhängig und zum Teil in Bibliotheken nicht transparent gekapselt. Die Effizienz solcher *Garbage Collection*-Bibliotheken ist zudem oftmals anwendungsspezifisch. Positiv ist jedoch, dass auch *c*-Bibliotheken (z. B. Boehm GC[2]) für die automatisierte Speicherverwaltung existieren. Ein weitverbreitetes, sehr effizientes und den Entwickler unterstützendes Konzept ist *Reference Counting*. Dieses wird im folgenden Kapitel ausführlich betrachtet.

[1] Automatische Speicherbereinigung.
[2] `http://www.hboehm.info/gc/`

3 Reference Counting

Dieses Kapitel gibt einen fundamentalen Überblick über das Konzept des *Reference Countings*. Anhand eines Beispiels wird die Idee erklärt und anschließend mit Hilfe von Regeln beschrieben. Abschließend werden typische Anwendungsbereiche und Handlungsfelder betrachtet.

Im Allgemeinen ist Speichermanagement der Prozess der Reservierung, Nutzung und Freigabe von Speicherressourcen zur Laufzeit des Programmes. Ein gut geschriebenes Programm verwendet dabei so wenig Speicher wie möglich und so viel Speicher wie nötig.

Speichermanagement-Modelle basieren auf dem Besitz eines Objektes. Der Besitz limitierter Speicherressourcen wird von verschiedenen Daten und Codefragmenten genutzt. Das Ziel ist Ressourcen, die nicht mehr benötigt werden, freizugeben. Somit werden nicht mehr Objekte im Speicher gehalten als notwendig. Dies ist insbesondere für Echtzeitanwendungen ein hinreichendes Kriterium.

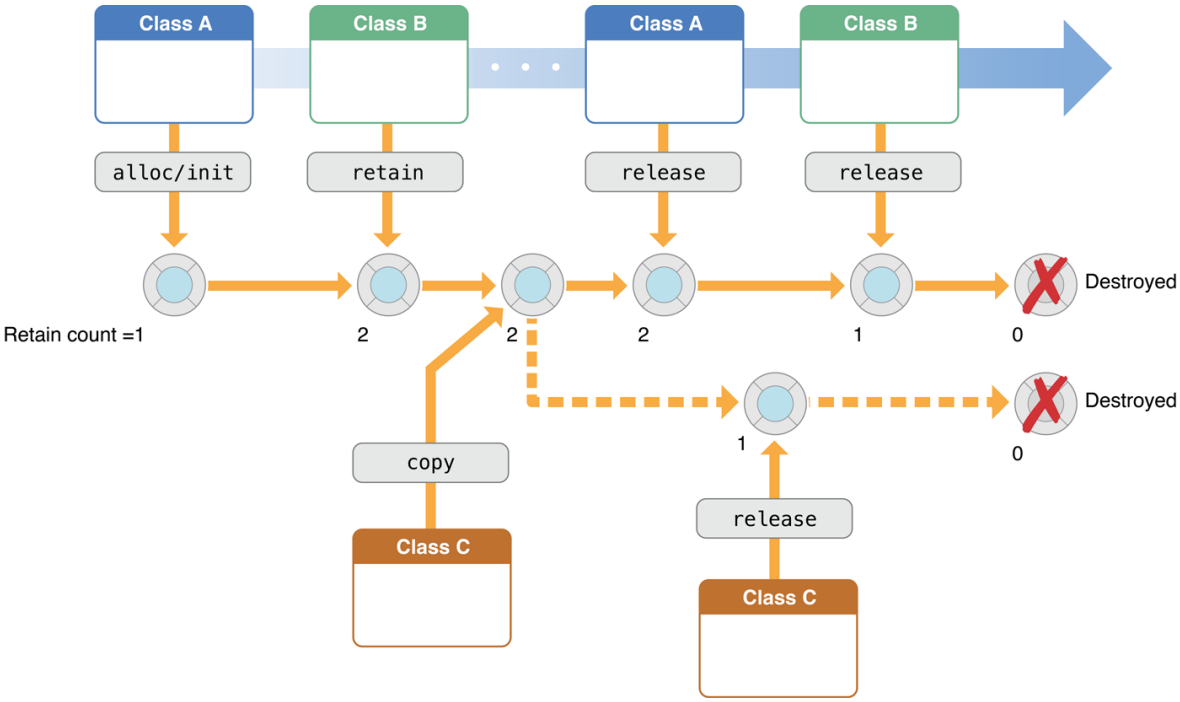


Abbildung 3.1: Speicherverwaltung mittels Reference Counting. [4]

Die Abbildung 3.1 zeigt einen exemplarischen Prozess der *Reference Counting* über die Zeit veranschaulicht. Das Beispiel verwendet Objective-C Syntax, kann aber auf andere Sprachen problemlos übertragen werden. Die Kreise mit blauem Zentrum beschreiben ein Objekt, das von verschiedenen Klassen verwendet wird. Dieses Objekt enthält einen Referenzzähler (engl. Retain count), der angibt, wie viele Referenzen auf dieses Objekt existieren.

Am Anfang erstellt die Klasse A (engl. Class A) mit Hilfe der Funktion *alloc/init* ein Objekt. Der Referenzzähler des Objektes wird hierbei mit dem Wert eins initialisiert. Die Klasse B (engl. Class B) erhält im Folgenden durch den *retain*-Funktionsaufruf auf das Objekt ebenfalls Besitz. Der Referenzzähler des Objektes wird gleichzeitig um den Wert eins inkrementiert.

Der *copy*-Funktionsaufruf der Klasse C (engl. Class C) erstellt eine Kopie des Objektes. Dabei ist zu beachten, dass der Referenzzähler des kopierten Objektes den Wert eins erhält, da ausschließlich eine Referenz auf das neue Objekt existiert. Wird nun auf das kopierte Objekt von der Klasse C die *release*-Funktion aufgerufen, so wird der Referenzzähler um den Wert eins dekrementiert und das zuvor kopierte Objekt freigegeben, da keine Referenz mehr auf dieses Objekt existiert.

Das ursprüngliche Objekt besitzt nach dem *copy*-Funktionsaufruf nach wie vor den Wert zwei, da die Klasse C keine Referenz auf das Objekt besitzt, sondern lediglich auf das kopierte Objekt. Sofern die Klassen A und B das Objekt nicht länger benötigen, rufen diese die *release*-Funktion auf und das Objekt wird ebenfalls freigegeben.

3.1 Grundlegende Speichermanagement Regeln

Nach [5] basieren Speichermanagement-Modelle grundsätzlich auf dem Besitz eines Objektes. Es gibt vier Regeln:

1. Jedes Objekt gehört dem Erzeuger (z.B. *malloc(..)*, *calloc(..)*, *copy(..)*).
2. Besitz kann geteilt/übernommen werden (z.B. *retain(..)*).
3. Besitz aufgeben, wenn Objekte nicht mehr benötigt werden (z.B. *release(..)*).
4. Keinen Besitz aufgeben, den man nicht besitzt.

Mit Hilfe der konsequenten Anwendung dieser vier Regeln sind Entwickler in der Lage Systeme zu entwickeln, die fehlerfrei von *Memory Corruption* und *Memory Leaks* sind.

3.2 Typische Anwendungsbereiche

Kevlin Henney erläutert drei Szenarios [3] in denen *Reference Counting* praktisch eingesetzt wird:

1. Objekt-Lebenszeit-Management für gemeinsam genutzte Objekte: Speichermanagement.
2. Einschränkungs-Management für Beziehungen zwischen gemeinsam genutzten Objekten: Limitierung der Anzahl an Referenzen (Biologie/Chemie).
3. Metadaten: Zählen der Instanzen einer Klasse (Debugging/Profiling).

3.3 Zyklische Datenstrukturen

Das größte Handlungsfeld des *Reference Countings* sind zyklische Datenstrukturen, da diese nach den grundlegenden Speichermanagement Regeln (Vgl. Abschnitt 3.1) einer rekursiven Besitzabhängigkeit entsprechen. Diese führt bei der Freigabe entsprechender Datenstrukturen zu Problemen. Exemplarische Datenstrukturen für diese Problematik sind beispielsweise hierarchische Baumstrukturen, da die Eltern- auf ihre Kindknoten und die Kind- auf ihre Elternknoten verweisen.

Ein Objekt sollte erst dann freigegeben werden, wenn dieses alle referenzierten Ressourcen freigeben hat. Bei der Freigabe von zyklischen Datenstrukturen gibt es nun ein Problem. Ein Elternobjekt, das freigegeben werden soll, ruft die *release*-Funktion aller Kindknoten auf. Die Kindknoten können gleichwohl nicht freigegeben werden, da diese noch eine Referenz auf ihr Elternobjekt besitzen, welches abermals nicht freigegeben werden kann.

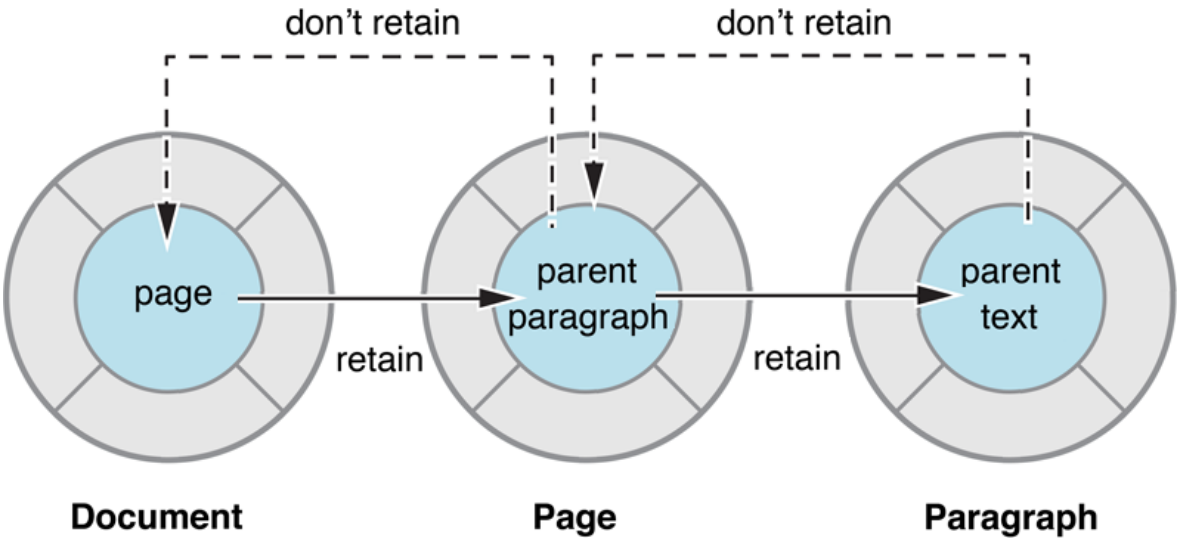


Abbildung 3.2: Zyklische Datenstrukturen beim Reference Counting. [6]

Die Abbildung 3.2 veranschaulicht solch eine problematische, zyklische Datenstruktur. Das *Document*-Objekt besitzt eine Referenz auf ein *Page*-Objekt und jedes *Page*-Objekt besitzt wiederum eine Referenz auf das *Document*-Objekt, sodass eine Zuordnung zum jeweiligen Dokument möglich ist. Diese rekursive Struktur ist für das *Page*-Objekt und das *Paragraph*-Objekt analog. Kein Objekt in der aktuellen Datenstruktur kann jemals freigegeben werden, da aufgrund der rekursiven Datenstruktur die jeweiligen Referenzzähler der Objekte niemals den Wert null erreichen können.

Die Lösung des Problems liegt in der Verwendung von starken und schwachen Referenzen (engl. strong-/weak-references) und der Einführung von neuen Regeln. Die bisher eingeführte Referenzierung entspricht der *strong*-Referenz und die *weak*-Referenz einer abgeschwächten Form. Die schwache Referenz beansprucht keinen Besitz an einem Objekt, sodass der Referenzzähler des referenzierten Objektes unverändert bleibt und die Referenz

nicht zwangsläufig valide sein muss. Der Vorteil ist, dass diese Objekte freigegeben werden können, ohne ihre referenzierten Objekte alle freizugeben.

Die neuen Regeln besagen, dass Eltern- ihre Kindobjekte mit *strong*-Referenzen referenzieren und Kind- ihre Elternobjekte ausschließlich mit *weak*-Referenzen referenzieren. Somit wird gewährleistet das rekursive Datenstrukturen freigegeben werden können.

3.4 Sprachunterstützung

Sprachen, die das *Reference Counting*-Konzept unterstützen [9]:

- c++11
- Cocoa (Automatic Garbage Collection)
- GObject (Atomare Operation für sicheres Multithreading)
- Python[1]
- VALA[2]
- File systems
- COM, Delphi, Perl, Tcl, u.v.m.

3.5 Anwendungsbeispiel

Das folgende praktische Beispiel veranschaulicht das *Singelton*-Entwurfsmuster mit Hilfe von *Reference Counting* und *GLib* [7]. Das *Singelton*-Entwurfsmuster stellt sicher, dass es von einer Klasse nur eine Instanz gibt und stellt diese über eine Fabrikmethode zur Verfügung. Dies stellt eine sehr einfache und effiziente Methode der Speicherverwaltung dar.

[1] `http://docs.python.org/2/extending/extending.html`
[2] `https://wiki.gnome.org/Projects/Vala/ReferenceHandling`

```
static MySingleton * the_singleton = NULL;
static GObject * my_singleton_constructor(GType type,
            guint n_construct_params,
            GObjectConstructParam * construct_params)
{
    GObject * object = NULL;
    if (!the_singleton)
    {
        object = G_OBJECT_CLASS(parent_class)->constructor(type,
                    n_construct_params, construct_params);
        the_singleton = MY_SINGLETON(object);
    }
    else
        object = g_object_ref(G_OBJECT(the_singleton));

    return object;
}
```

Zum Programmstart wird die statische Variable *the_singelton* mit dem Wert *NULL* initialisiert. Sofern ein anderes Objekt dieses *Singelton* verwenden will, bekommt es eine Referenz mit Hilfe der statischen *my_singelton_constructor*-Fabrikmethode. Wenn ein anderes Objekt bereits das *Singelton* nutzt wird lediglich der Referenzzähler inkrementiert und eine Referenz auf das bereits vorhandene Objekt zurückgegeben. Wenn noch kein *Singelton* existiert wird eines erzeugt und dessen Referenz zurückgegeben. Wenn alle Objekte ihren Besitzanspruch aufgegeben haben, wird das *Singelton*-Objekt freigegeben.

4 Effizientes Speichermanagement in C

In diesem Kapitel wird ein Performanzvergleich für drei Speichermanagementansätze durchgeführt. Folgende Ansätze werden untersucht:

1. Standard *c*: Manuelle Speicherverwaltung
   malloc(..), calloc(..), free(..)
2. Reference Counting nach Jean-David Gadina:
   alloc(..), retain(..), release(..)
3. Reference Counting mit *GLib*:
   my_object_new(..), g_object_ref(..), g_object_unref(..)

Der Performanzvergleich untersucht drei Funktionalitäten für eine int-Variable. Dabei werden detaillierte Zeitmessungen für Speicherallokation, Referenzieren und Speicherfreigabe durchgeführt. Für jede zu untersuchende Funktionalität werden dabei jeweils eine bis zehn Millionen Iterationsschleifen ausgeführt. Um Messungenauigkeiten und äußeren Fehlerquellen vorzubeugen, wird letztendlich der Mittelwert plus zugehöriger Standardabweichung berechnet und verglichen. Es werden keine Optimierungen durchgeführt.

4.1 System/Plattform

Alle Performanztests werden auf dem WR-Cluster[1] durchgeführt. Der WR-Cluster verfügt über zehn Knoten, wobei jeder Knoten zwei Prozessoren (Intel Xeon Westmere 5650 @ 2.67GHz) mit jeweils sechs Cores besitzt. Des Weiteren verfügen die Knoten jeweils über zwölf GByte DDR3 / PC1333 Hauptspeicher (System taktet mit 1333 MHz). Der Test besteht insgesamt aus zehn Batches, wobei jeweils ein Task pro Node läuft.

4.2 Standard C: Manuelles Speichermanagement

Pseudo Code für die Performanzmessung mit manuellem Speichermanagement:

[1] `http://wr.informatik.uni-hamburg.de/`

```
unsigned long i = 0;
int * referencePointer = NULL;
int ** ptrs = (int **)calloc(numIterations, sizeof(int *));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = (int *)calloc(1, sizeof(int));    // 1. Allocation

for(i = 0; i < numIterations; ++i)
    referencePointer = ptrs[i];                 // 2. Retain

for(i = 0; i < numIterations; ++i)
    free(ptrs[i]);                              // 3. Release

free(ptrs);
```

Die Implementierung des manuellen Speichermanagements beginnt mit der Deklaration einiger Hilfsvariablen. Die *i*-Variable ist eine Iterationsvariable für die Schleifendurchläufe, die *referencePointer*-Variable dient als Hilfszeiger um sich exemplarisch eine Referenz auf eine *int*-Variable aus dem *int*-Feld zu holen und die *ptrs*-Variable entspricht einem Zeiger auf ein Feld mit *int*-Zeigern. Es wird Speicher für das *int*-Zeigerfeld (*numIterations* x *int*-Zeiger) allokiert.

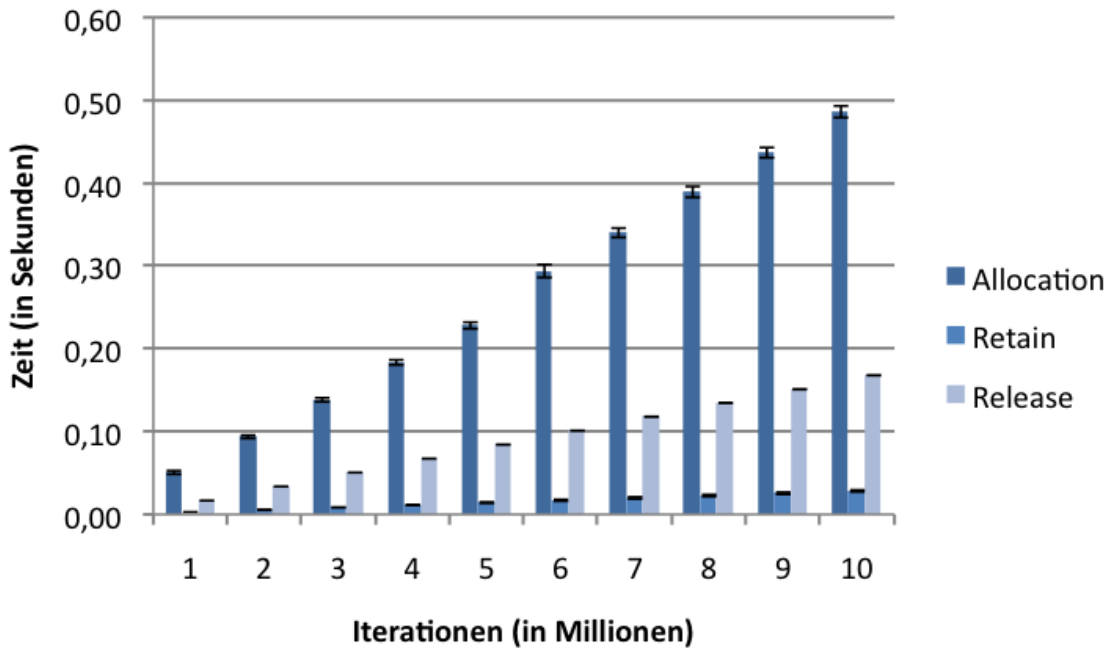


Abbildung 4.1: Performanz: Standard C ohne Reference Counting.

In der ersten Iterationsschleife wird *numIterations*-Mal Speicher für eine einzelne *int*-Variable allokiert und dem *int*-Zeigerfeld an entsprechendem Index zugewiesen. Die zweite Schlei-

fe durchläuft das gesamte *int*-Zeigerfeld und holt sich beispielhaft in jedem Durchlauf eine Referenz auf die aktuelle *int*-Variable. Im dritten Schleifendurchgang wird der Speicher für jede einzelne *int*-Variable freigegeben. Abschließend wird dann noch der Speicher für das *int*-Zeigerfeld wieder freigegeben.

Die Abbildung 4.1 veranschaulicht die drei gemessenen Zeiten für Allokation, Retain und Release pro Iterationszyklus. Die Allokation von einer Million *int*-Variablen hat beispielsweise circa $0,05$ Sekunden benötigt, für zehn Millionen *int*-Variablen in etwa $0,5$ Sekunden. Die benötigte Zeit ist linear proportional zur Anzahl der Iterationen. Die gemessene Standardabweichung ergibt sich aus der zehnfachen Wiederholung der Performanzmessung und ist im Balkendiagramm ebenfalls illustriert.

4.3 Reference Counting nach Jean-David Gadina [2]

Jean-David Gadina definiert vorab einen essentiellen Datentypen *MemoryObject*, der einen Referenzzähler und *void*-Zeiger kapselt. Die Kapselung ermöglicht Referenzzählung für jeden Datentypen. Der Datentyp *MemoryObject* wird folgendermaßen definiert:

```
typedef struct
{
    unsigned int retainCount;
    void * data;
} MemoryObject;
```

Mit Hilfe einer *alloc*-Funktion wird dem Programmierer eine Speicherallokationsroutine zur Verfügung gestellt. Diese Funktion hat die identische Signatur wie die Standardfunktion *malloc* und kann daher äquivalent verwendet werden. Der Funktionsrumpf wird wie folgt definiert:

```
void * alloc(size_t size)
{
    MemoryObject * o = NULL;
    char * ptr = NULL;

    o = (MemoryObject *)calloc(sizeof(MemoryObject) + size, 1);
    if(o != NULL)
    {
        ptr = (char *)o;
        ptr += sizeof(MemoryObject);
        o->retainCount = 1;
        o->data = ptr;
    }
    return (void *)ptr;
}
```

Die *alloc*-Funktion allokiert einen Speicherblock mit Hilfe der Standard *calloc*-Funktion in der geforderten Größe *size* plus der Größe des Datentyps *MemoryObject*. Der allokierte Speicherblock wird anschließend in einen *MemoryObject*-Zeigertypen gecastet und dessen *retainCount*-Variable mit dem Wert eins initialisiert (Vgl. Kapitel 3). Der *void*-Zeiger des *MemoryObjects* wird anschließend auf den vom Entwickler geforderten Speicherbereich umgebogen. Mittels Zeigerarithmetik wird der reservierte Speicher für das *MemoryObject* übersprungen (*ptr += sizeof(MemoryObject)*), sodass der *data*-Zeiger auf den eigentlichen Speicherbereich zeigt. Das Funktionsergebnis ist der *data*-Zeiger selbst, sodass Entwickler den geforderten Speicherblock zurück bekommen und das *Reference Counting* versteckt bleibt.

Um sich eine Referenz auf ein bereits existierendes Objekt zu holen, existiert die *retain*-Funktion. Sie ist wie folgt definiert:

```
void retain(void * ptr)
{
    MemoryObject * o = NULL;
    char * cptr = NULL;

    cptr = (char *)ptr;
    cptr -= sizeof(MemoryObject);
    o = (MemoryObject *)cptr;

    o->retainCount++;
}
```

Das Referenzieren eines bereits existierenden *int*-Objekts entspricht dem Inkrementieren der versteckten *retainCount*-Variable. Die *MemoryObject*-Informationen stehen im Speicher vor dem eigentlichen Speicherbereich des Entwicklers. Daher wird mittels inverser Zeigerarithmetik der *cptr*-Zeiger auf den Speicheranfang des *MemoryObjects* gesetzt (*cptr -= sizeof(MemoryObject)*) und der Zeiger anschließend in einen *MemoryObject*-Datentypen gecasted, sodass die *retainCount*-Variable um den Wert eins inkrementiert werden kann.

```
void release(void * ptr)
{
    MemoryObject * o = NULL;
    char * cptr = NULL;

    cptr = (char *)ptr;
    cptr -= sizeof(MemoryObject);
    o = (MemoryObject *)cptr;

    if(--o->retainCount == 0)
        free(o);
}
```

Der *release*-Funktionsrumpf ähnelt dem der *retain*-Funktion, da lediglich die *retainCount*-Variable dekrementiert anstatt inkrementiert wird. Die Besonderheit der *release*-Funktion ist, dass wenn die *retainCount*-Variable den Wert null erreicht, das gesamte Speicherobjekt freigegeben wird.

Pseudo Code für die Performanzmessung der *Reference Counting*-Implementierung von Jean-David Gadina:

```
unsigned long i = 0;
int * referencePointer = NULL;
int ** ptrs = (int **)calloc(numIterations, sizeof(int *));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = (int *)alloc(sizeof(int));    // 1. Allocation

for(i = 0; i < numIterations; ++i) {
    referencePointer = ptrs[i];
    retain(referencePointer);               // 2. Retain
}

for(i = 0; i < numIterations; ++i)
    release(ptrs[i]);                       // 3. Release

free(ptrs);
```

Für den Performanztest von Jean-David Gadina's Implementierung werden die gleichen Hilfsvariablen wie zuvor verwendet (Vgl. Abschnitt 4.2). Für die Speicherallokation der einzelnen *int*-Variablen wird, die anfangs definierte *alloc*- anstatt der standard *calloc*-Funktion verwendet. Somit besitzt jede allokierte *int*-Variable einen eigenen Referenzzähler. Dieser wird beim Holen der Referenz durch die *retain*-Funktion inkrementiert. Die anschließende Freigabe wird in der *release*-Funktion gekapselt und muss nicht explizit aufgerufen werden. Dabei ist zu beachten, dass der Referenzzähler vor dem letzten Schleifendurchlauf den Wert zwei besitzt und somit der für die *int*-Variablen allokierte Speicher beim *release*-Funktionsaufruf nicht freigegeben wird. Daher wird im Performanztest zwischen der angegebenen *retain*- und der *release*-Iterationsschleife eine weitere *release*-Schleife durchgeführt (Vgl. Codebeispiel[2]), sodass der Referenzzähler vor der letzten *release*-Schleife den Wert eins hat und somit der gesamte Speicher wieder freigegeben wird.

Die Abbildung 4.2 zeigt, wie Abbildung 4.1 auch, die benötigten Zeiten für Allokation, Retain und Release in Abhängigkeit der Anzahl der Schleifendurchgänge. Auffällig ist, dass im Vergleich zum manuellen Speichermanagement (Vgl. Abschnitt 4.2) mehr Zeit für die *retain*- und *release*-Funktion benötigt wird. Dies ist darauf zurückzuführen, dass die *Reference Counting*-Implementierung mehr Code ausführen muss, um die *retainCount*-Variable zu setzen.

[2] `http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2013_2014/epc-1314-gaack-reference-counting-code.zip`

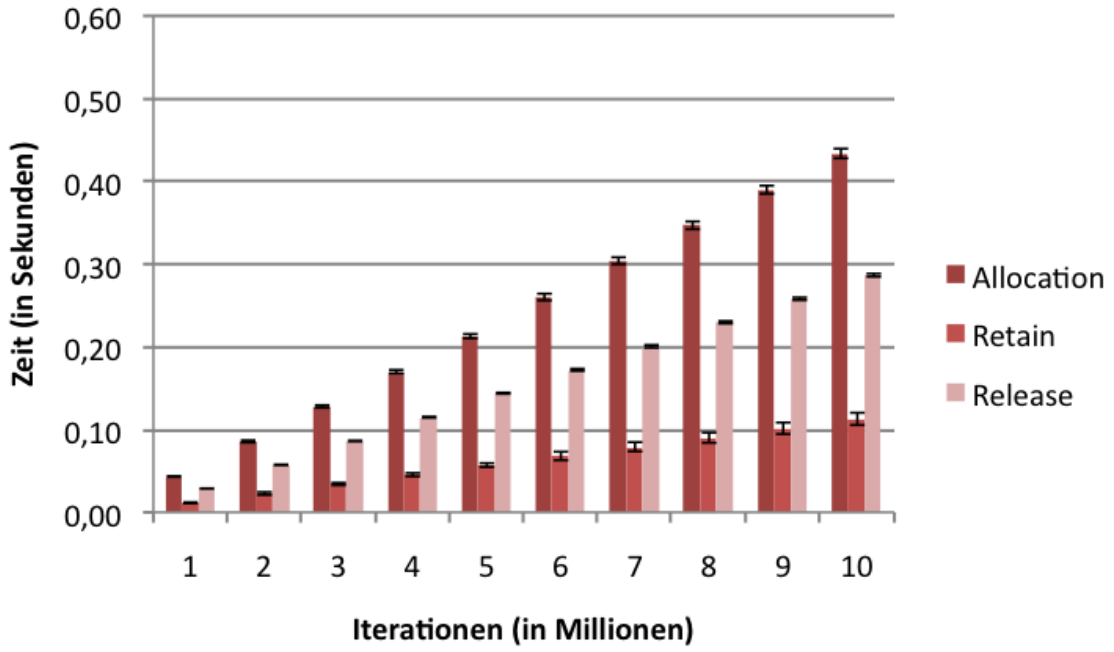


Abbildung 4.2: Performanz: Reference Counting nach Jean-David Gadina.

4.4 Reference Counting mit GLib

Der *Reference Counting* Performanztest mit Hilfe von *GLib* zeigt, dass es auch freie Bibliotheken gibt, die das Konzept des *Reference Countings* in der Programmiersprache *c* unterstützen. Zu Beginn werden die gleichen Hilfsvariablen wie zuvor deklariert. Allerdings bietet *GLib* integrierte Datentypen, wie beispielsweise *gpointer* an, um eine bessere Kapselung und Lesbarkeit zu unterstützen.

Pseudo Code für die Performanzmessung der *Reference Counting*-Implementierung mit Hilfe von *GLib*:

```
unsigned long i = 0;
gpointer referencePointer = NULL;
gpointer * ptrs = (gpointer *)calloc(numIterations,
                                     sizeof(gpointer));

for(i = 0; i < numIterations; ++i)
    ptrs[i] = my_object_new(0);              // 1. Allocation

for(i = 0; i < numIterations; ++i)           // 2. Retain
    referencePointer = g_object_ref(G_OBJECT(ptrs[i]));

for(i = 0; i < numIterations; ++i)
    g_object_unref(G_OBJECT(ptrs[i]));       // 3. Release

free(ptrs);
```

Um *GLib Reference Counting* für beliebige Datentypen einzusetzen müssen entsprechende Daten in einem *GObject*-Datentyp gekapselt werden. Im Rahmen des Performanztests wurde eine Klasse namens *MyObject* erzeugt, die exemplarisch eine *int*-Variable enthält. In der ersten Iterationsschleife wird der Konstruktur der *MyObject*-Klasse aufgerufen und somit eine Instanz erzeugt, deren Referenzzähler mit dem Wert eins initialisiert wird. In der zweiten Iterationsschleife wird dann mittels der *g_object_ref*-Funktion eine Referenz auf das jeweilige Objekt geholt. Die *g_object_unref*-Funktion wird als *release*-Funktion in der letzten Iterationsschleife verwendet. Auch hier wird zuvor eine zusätzliche Schleife durchlaufen (Vgl. Codebeispiel[2]), sodass der Referenzzähler im letzten Durchlauf den Wert null erreicht und das Objekt freigegeben wird (Vgl. Abschnitt 4.3).

Die Abbildung 4.3 veranschaulicht, dass insbesondere die Allokation sowie das Freigeben des *GObject*-Datentypen viel Zeit benötigt. Die Ordinatenachse ist im Vergleich zu den vorherigen Messungen in größere Zeitintervalle eingeteilt. Der höhere Zeitaufwand ist darauf zurückzuführen, dass *GLib* ein objektorientierter Ansatz ist und somit viel mehr Speicher für die Daten benötigt. Folglich entsteht ein großer Overhead für das *Reference Counting*.

4.5 Evaluierung

Im Folgenden werden die drei Performanztests hinsichtlich Allokations-, Retain- und Releasezeiten verglichen.

4.5.1 Allokation Performanztest

Die Abbildung 4.4a zeigt deutlich, den in Abschnitt 4.4 erwähnten Overhead des objektorientierten Ansatzes von *GLib*. Während der Zeitunterschied zu den ersten beiden Ansätzen bei einer Million Iteration noch etwa eine Sekunde beträgt, steigt die zeitliche Differenz zu-

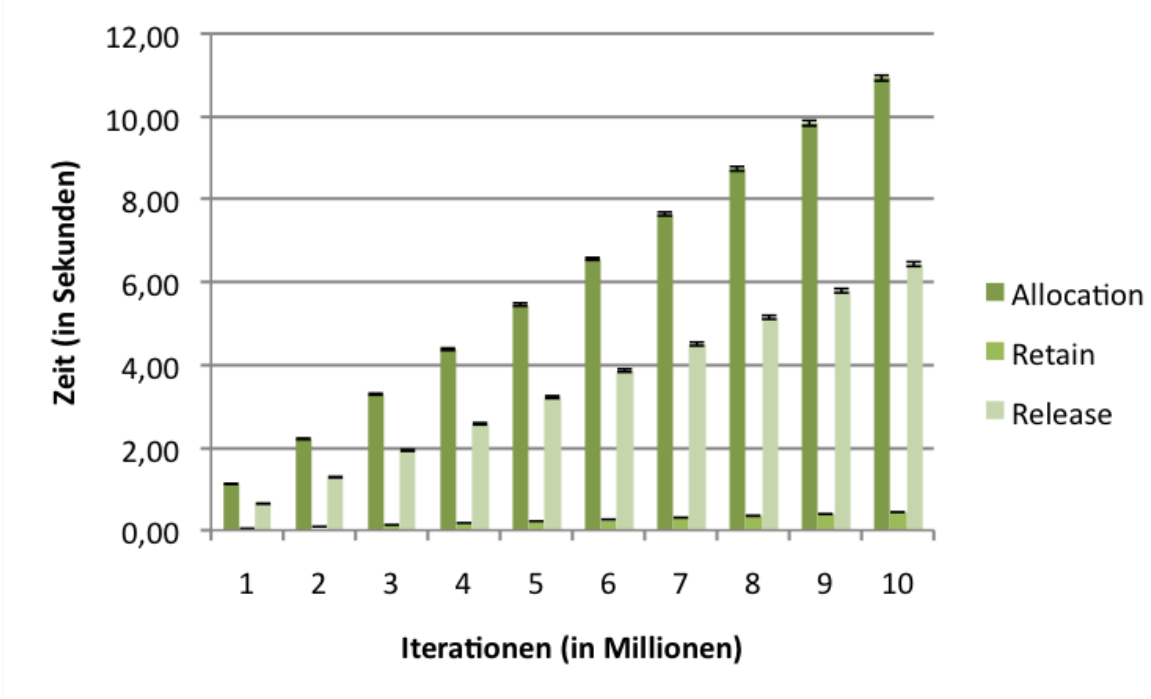


Abbildung 4.3: Performanz: Reference Counting mit GLib.

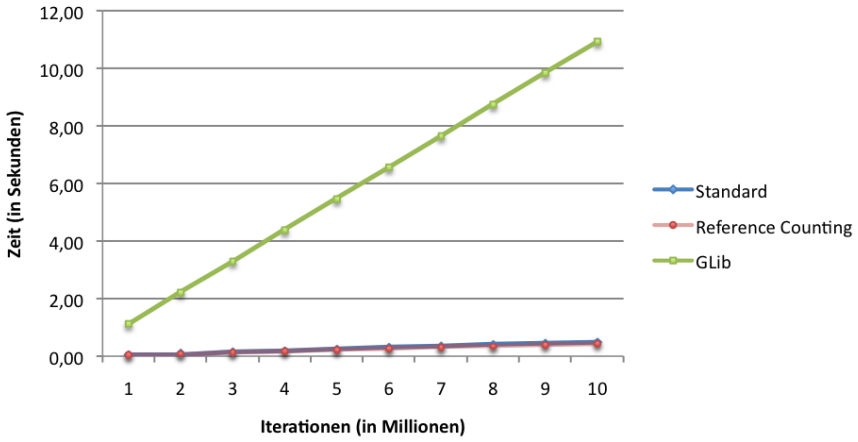
nehmend mit der Iterationsanzahl, sodass beispielsweise bei zehn Millionen Iterationen der Unterschied größer als zehn Sekunden ist.

Bei dem detaillierten Vergleich zwischen der Standard- und der *Reference Counting*-Implementierung nach Jean-David Gadina gibt es nur geringfügige Zeitdifferenzen. Außergewöhnlich ist jedoch der deutlich steigende Zeitunterschied zwischen fünf und sechs Million Iterationen, der sich ab dort an fortsetzt. Zu erwarten ist eigentlich, dass der Ansatz mit manueller Speicherverwaltung weniger Zeit benötigt, da dieser auch weniger Speicher als der *Reference Counting*-Ansatz braucht. Äußere Einflussfaktoren bzw. alternative Fehlerquellen sind durch die zehnfache Wiederholung des Performanztests sowie der Betrachtung der Standardabweichung (Vgl. Abbildung 4.1) größtenteils ausgeschlossen. Warum kommt es dann aber zu dem Laufzeitunterschied?

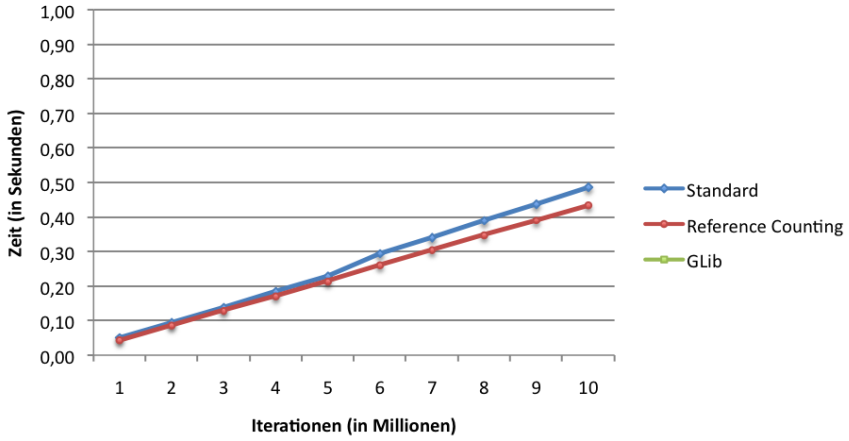
Bei genauerer Betrachtung unterscheiden sich die beiden Allokationsaufrufe grundlegend nur in ihrer Parameterreihenfolge beim *calloc*-Funktionsaufruf:

```
// Manual memory management
.. = (int *)calloc(1, sizeof(int));

// Reference Counting (Jean-David Gadina)
.. = (MemoryObject *)calloc(sizeof(MemoryObject) + size, 1);
```



(a) Messwerte für Speicherallkoation.



(b) Detailbetrachtung der Messwerte für Speicherallkoation.

Abbildung 4.4: Performanz: Allokation.

Die *calloc*-Funktion ist laut Definition[3] eine Speicherallokationsroutine für ein Array mit x Elementen der Größe y, wobei alle allokierten Bits mit null initialisiert werden. Die Vertauschung der Parameter, wie es in Jean-David Gadina's Implementierung der Fall ist, führt höchst wahrscheinlich zu einer effizienteren Allokation und/oder Initialisierung der Bits.

4.5.2 Retain Performanztest

Das Referenzieren mittels einer *retain*-Funktion benötigt den geringsten Zeitaufwand. Beim manuellen Speichermanagementansatz entspricht dieser beispielsweise der Zeit für einen Arrayzugriff und eine Zuweisung. Die Abbildung 4.5 zeigt: Je komplexer die jeweilige *re-*

[3] `http://www.cplusplus.com/reference/cstdlib/calloc/`

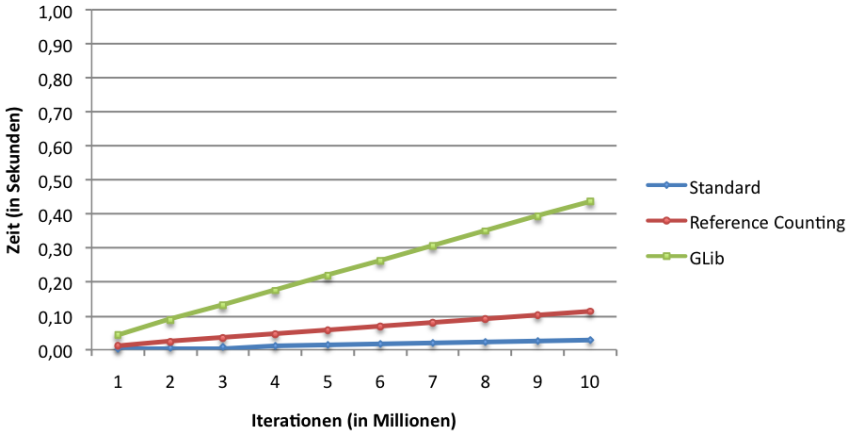


Abbildung 4.5: Performanz: Retain.

tain-Funktion ist, desto mehr Zeit wird für die entsprechende Referenzierung benötigt. Während bei einer Million Iterationen die Zeitdifferenz zwischen dem objektorientierten Ansatz und den ersten beiden Ansätzen noch marginal ist, steigt diese mit zunehmenden Iterationen deutlich an.

4.5.3 Release Performanztest

In der Abbildung 4.6 ist deutlich zu erkennen, dass neben der Allokation (Vgl. Unterabschnitt 4.5.1) auch das Freigeben in Abhängigkeit vom reservierten Speicher zu betrachten ist. Der objektorientierte *GLib*-Ansatz benötigt beispielsweise bei acht Millionen Iterationen bereits annähernd fünf Sekunden mehr als die alternativen Ansätze (Vgl. Abschnitt 4.2 und Abschnitt 4.4).

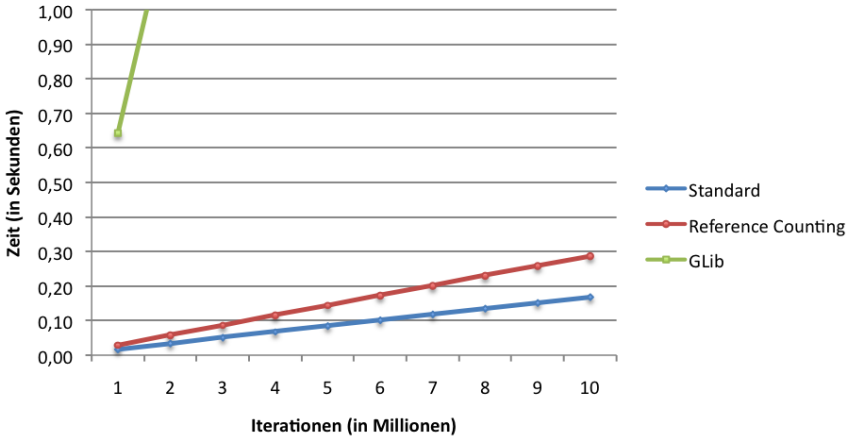


Abbildung 4.6: Performanz: Release. Detailbetrachtung der Messwerte für Speicherallkoation.

Bemerkenswert ist, dass der Zeitunterschied zwischen manueller Speicherverwaltung und Jean-David Gadina's *Reference Counting*-Ansatz bei zehn Millionen Iterationen lediglich circa eine zehntel Sekunde beträgt. Das heißt, dass der Overhead für die Freigabe von Speicherblöcken beim *Reference Counting* sehr gering ist.

5 Zusammenfassung

Reference Counting ist ein weitverbreitetes, unterstütztes und zum Teil integriertes Konzept. Auch Implementierungen in der Programmiersprache *c* sind möglich und in Form von *c*-Bibliotheken, wie beispielsweise *GLib* frei verfügbar.

Reference Counting kann ein sehr effizientes Hilfsmittel für Speichermanagement sein. Der Performanztest hat differenziert gezeigt, dass der bedingte, zusätzliche zeitliche Aufwand sehr gering ausfällt und somit das Konzept insbesondere für Echtzeitanwendungen oder Systeme mit limitierten Speicher ideal geeignet ist. Darüber hinaus dient *Reference Counting* als unterstützendes Konzept für Entwurfsmuster und ist somit flexibel in beliebigen Softwareprojekten integrierbar.

Allerdings entspricht der Einsatz von *Reference Counting* keiner Universallösung für Speichermanagement, sodass *Memory Corruption* und *Memory Leaks* nach wie vor möglich sind. Ein großes Handlungsfeld, das von Entwicklern stets zu beachten ist, sind zyklische Datenstrukturen.

Literatur

[1] Batov, V. Extending the Reference-Counting Pattern. http://www.drdobbs.com/extending-the-reference-counting-pattern/184403543, 9 1998. Zugriff erfolgt: 13-03-2014.

[2] Gadina, J.-D. Reference counting in ANSI-C. http://www.xs-labs.com/en/archives/articles/c-reference-counting/, 2013. Zugriff erfolgt: 13-03-2014.

[3] Henney, K. C++ patterns-reference accounting. In in Proceedings of the EuroPLoP 2002 conference,(Irsee), Citeseer (2002).

[4] Library, M. D. About Memory Management. Advanced Memory Management Programming Guide (07 2012). https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html. Zugriff erfolgt: 13-03-2014.

[5] Library, M. D. Basic Memory Management Rules. Advanced Memory Management Programming Guide (07 2012). https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html. Zugriff erfolgt: 13-03-2014.

[6] Library, M. D. Practical Memory Management. Advanced Memory Management Programming Guide (07 2012). https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html. Zugriff erfolgt: 13-03-2014.

[7] Project, T. G. GObject Reference Manual. GNOME DEV CENTER (2012). https://developer.gnome.org/gobject/stable/gobject-The-Base-Object-Type.html. Zugriff erfolgt: 13-03-2014.

[8] Wikipedia. Garbage Collection. http://de.wikipedia.org/wiki/Garbage_Collection, 2 2014. Zugriff erfolgt: 13-03-2014.

[9] Wikipedia. Reference Counting: Sprachunterstützung. http://en.wikipedia.org/wiki/Reference_counting#Examples_of_use, 1 2014. Zugriff erfolgt: 13-03-2014.