FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Florian Fischer

A Survey and Evaluation Framework of Memory Allocators on Many-Core Systems

Bachelorarbeit im Fach Informatik

Please cite as:
Florian Fischer, "A Survey and Evaluation Framework of Memory Allocators on Many-Core Systems", Bachelor's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, October 2018.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

A Survey and Evaluation Framework of Memory Allocators on Many-Core Systems

Bachelorarbeit im Fach Informatik

vorgelegt von

Florian Fischer

geb. am 30. August 1995
in München

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Florian Schmaus, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Februar 2018**
Abgabe der Arbeit:	**27. September 2018**

Erklärung

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.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Florian Fischer)
Erlangen,

ABSTRACT

Dynamic memory management is a fundamental technique for all programmers. The endless amount of different programs and the simple interface are the cause of huge internal complexity. That is why there are numerous competing implementations and extensive ongoing scientific interest. Choosing the right memory allocator for the used hard- and software environment is a difficult task.

This work gives a short summary of some common techniques. In its course a benchmark framework was implemented, which is suitable to benchmark any allocator with any given workload. This framework is used to obtain and present a comparison of the four memory allocators: *glibc*, *jemalloc*, *tcmalloc* and *Hoard* on a multi-core system.

This comparison shows, that thread local caches which are the new standard to avoid synchronization all together, clearly improve the throughput and scalability of the allocator. Differences in how external fragmentation is handled can be spotted but which technique is the best must be decided for each actual program behavior. The arrangement shows that benchmarks can never cover the whole program behavior and often really test only one aspect. That's why the actual application has to be benchmarked with the feasible allocators. Additionally it stands out that memory allocators ignore hardware topologies like NUMA which are an essential part of future many-core systems.

KURZFASSUNG

Dynamische Speicherverwaltung ist ein nicht wegzudenkendes Werkzeug für jeden Programmierer. Das unbegrenzte Spektrum an auf sie angewiesenen Anwendungen, sowie ihre einfache Schnittstelle führen zu enormer inneren Komplexität. Diese Komplexität drückt sich im großen Umfang der über die Jahre entstandene Forschung und die vielen konkurrierenden Implementierungen aus. Die Wahl der geeignetsten Implementierung für verwendete Soft- und Hardware ist deshalb nicht einfach.

In der vorliegenden Bachelorarbeit soll ein kurzer Überblick über die verwendeten Techniken gegeben sowie eine Umgebung vorgestellt werden, die es ermöglicht Speicherverwaltungen für unterschiedliche Anwendungen und nach einzelnen Kriterien zu untersuchen. Außerdem wird mit Hilfe der entstandenen Umgebung ein Vergleich der vier Implementierungen: *glibc*, *jemalloc*, *tcmalloc* und *Hoard* auf einem Mehrkernsystem präsentiert.

Der durchgeführte Vergleich zeigt, dass der aktuelle Stand der Technik, die weitgehende Vermeidung von Synchronisation durch fadenlokale Zwischenspeicher, zu einer deutlichen Steigerung des Durchsatzes und der Skalierbarkeit der dynamischen Speicherverwaltung führt. Unterschiede sind darüber hinaus im Umgang mit externer Fragmentierung zu erkennen, welche Herangehensweise allerdings für den wenigsten zusätzlichen Speicherbedarf sorgt muss im Einzelfall bestimmt werden. Die aus dem Vergleich gewonnene Erkenntnis ist, dass generische Tests niemals die Realität sondern nur einen Teilaspekt abbilden können und somit für eine optimale Entscheidung immer die betrachtete Anwendung selbst Untersuchungsgegenstand sein muss. Zusätzlich ist aufgefallen, dass Hardwaretopologien wie NUMA Architekturen, die in der Zukunft für Vielkernprozessoren eine entscheidende Rolle spielen werden noch nicht im Fokus der dynamischen Speicherverwaltungen angelangt sind.

INHALTSVERZEICHNIS

1 EINLEITUNG

Jedes nicht triviale Programm, dass die Größe seiner Eingabedaten nicht im Vornherein kennt, muss in der einen oder anderen Form seinen ihm zur Verfügung stehenden dynamischen Speicher, auch als Halde oder engl. Heap bezeichnet, während der Ausführung organisieren. Dies ist Aufgabe der dynamischen Speicherverwaltung, im Englischen memory allocator kurz allocator genannt. Trotz der oft ausgesprochenen Ratschläge, die Verwendung von dynamischem Speicher auf das benötigte Minimum zu reduzieren, weil die manuelle Bedienung des Allokators eine eigene Kategorie an oft schwer zu findenden Fehlern mit sich bringt, ist sie dennoch nicht aus den grundlegenden Werkzeugen der Informatik wegzudenken.

Jeder Programmierer kommt mit dynamischer Speicherverwaltung in Berührung entweder durch einen expliziten Aufruf wie in den Programmiersprachen C und C++ über `malloc` bzw. `new` oder vor ihm verborgen durch die Laufzeitumgebung, der von ihm verwendeten Programmiersprache.

Die Notwendigkeit einer abstrakten und einheitlichen Schnittstelle für dynamische Speicherverwaltung zur Laufzeit eines Programms wurde bereits früh erkannt. Bereits in den 1960er Jahren beschäftigte sich Fachliteratur mit der Verwaltung von Hauptspeicher. Trotz der zunächst hauptsächlichen Betrachtung von segmentiertem Speicher im Betriebsystemkern entstanden bereits viele Konzepte, die sich auch heute noch in Speicherverwaltungsstrategien wiederfinden, wie zum Beispiel verkettete Listen von Speicherblöcken [Wil+95]. Die Bibliotheksaufrufe `malloc`, um in einem Programm mehr Speicher anzufordern, und `free`, um diesen wieder an das System zurückzugeben, tauchten bereits 1979 in UNIX Version 7 der Bell Labs auf [Lab79] und sind seit dem ersten C Standard von 1989, auch als ANSI C oder ISO C bekannt, fester Bestandteil der C Standard Bibliothek.

Seit 1979 ist die Welt der Informatik nicht stehen geblieben. Computer sind deutlich schneller geworden und auch die grundsätzliche Architektur hat sich von einem verbauten Prozessor zu mehreren oder gar hunderten geändert. Die dynamische Speicherverwaltung hat diese Entwicklung nicht verschlafen und es gibt eine Vielzahl an verschiedenen Lösungen, von allgemeinen bis hin zu sehr spezifischen Ansätzen.

Die Forschung beschäftigt sich nach wie vor intensiv mit dem Thema der dynamischen Speicherverwaltung. Auch wenn vorerst keine fundamentalen und radikalen Neuerungen zu erwarten sind, reicht das Interesse aus, damit Internet und Software Giganten wie Facebook und Google sich dem Thema annehmen, ihre eigenen Lösungen entwickeln und pflegen. Motivation dafür ist, dass die angebotenen Software Dienste von Google und Facebook gut skalierbar sein müssen und folglich auch das benutzte Subsystem für die dynamische Speicherverwaltung. Verschwendeter Speicher oder länger als nötige Antwortzeiten können hier deutlich Kosten verursachen, wenn deshalb, um die geforderte Leistung zu erzielen, zum Beispiel mehr Hardware angeschafft werden muss.

Die Veröffentlichungen von *tcmalloc*, Google's Speicherverwaltung, und *jemalloc*, der Implementierung, der sich Facebook angenommen hat, lassen leider bei dem Thema Evaluation deutlich zu wünschen übrig. *Tcmalloc* präsentiert einen einzigen synthetischen Vergleich und auch nur mit *ptmalloc2*. In der Veröffentlichung von *jemalloc* findet sich ebenfalls nur ein einziger Vergleich jedoch mit tatsächlich bei Facebook gemessenen Anwendungsdaten. Die Motivation für diese Arbeit ist das eben genannte Fehlen von dokumentierten Vergleichen und die Tatsache, dass ich bei der Suche nach einem einfachen Weg, meine eigene Implementierung zu testen, keine befriedigende Lösung gefunden habe. In der betrachteten Literatur sind die verwendeten Vergleichstests entweder, wie oben erwähnt, sehr einseitig, der Quelltext sowie Beschreibung der Tests auf diverse Publikationen verteilt, oder sehr veraltet.

Deswegen soll in dieser Arbeit eine Vergleichsumgebung vorgestellt werden, die folgende drei Punkte erfüllt:

- **Hohe Flexibilität**: Es soll möglich sein jeden beliebigen Allokator mit jedem beliebigen Anforderungsprofil, egal ob synthetisch oder real, zu vermessen.
- **Einfachheit**: Trotz der gebotenen Flexibilität soll es ohne viel Aufwand und Komplexität möglich sein, neue Allokatoren und Anwendungsprofile in die Umgebung aufzunehmen.
- **Breites Analysespektrum**: Außerdem soll der Standardumfang bereits möglichst viele nicht funktionale Eigenschaften der untersuchten Allokatoren betrachten und zur Analyse aufbereiten.

Eine derartige Vergleichsumgebung ist geeignet, eine bestehende Anwendung mit unterschiedlichen *Allokatoren* zu testen, um eine gute und fundierte Entscheidung treffen zu können, welche Implementierung für die eingesetzte Hard- bzw. Software, sowie für das angestrebte Anwendungsverhalten die Richtige ist. Darüber hinaus ist es auch denkbar, verschiedene Änderungen im Allokator oder der Anwendung zu evaluieren, um mit Hilfe der Ergebnisse den Entwicklungsprozess zu gestalten.

Außerdem soll ein Vergleich von vier aktuellen dynamischen Speicherverwaltungen gezeigt werden, um sowohl einen Überblick über den Stand der verfügbaren Implementierungen als auch über die Testumgebung zu erhalten. Die vier ausgewählten Allokatoren sind: *tcmalloc* von Google, *jemalloc* von FreeBSD bzw. Facebook, den Allokator der GNU's Not Unix (GNU) C Bibliothek (*glibc*) sowie den Forschungsallokator *Hoard* von Emery Berger [Ber+00].

Der durchgeführte Vergleich legt dabei einen Fokus auf Mehrprozessorsysteme, da dies den aktuellen Trend sowohl der Hard- als auch der Softwareentwicklung widerspiegelt und ausgeschriebenes Einsatzgebiet der Allokatoren *tcmalloc* und *jemalloc* ist. Außerdem sind Speicherverwaltungen für den Einkernbetrieb ausgiebig untersucht und analysiert. Gerade deshalb ist der Vergleich zwischen *tcmalloc* bzw. *jemalloc* und der über Jahrzehnte gewachsenen, gewandelten und im Einsatz erfolgreich bewährten Implementierung in der *glibc* interessant.

Im Sinne einer Studie gliedert sich diese Arbeit in drei Abschnitte. In Kapitel 2 erfolgt kurz eine Einführung der Grundlagen. Darauf aufbauend werden die in dieser Arbeit verglichenen Allokatoren und deren Aufbau vorgestellt. Im Anschluss erfolgt in Kapitel 3 eine Betrachtung der entwickelten Vergleichsumgebung *allocbench*. Dabei wird kurz ihre Architektur und die zugrundeliegende Modellierung eines Tests sowie die fünf in ihr enthaltenen Vergleichstests vorgestellt. Zum Schluss werden die so entstandenen, konkreten Ergebnisse der Vergleichstests in Kapitel 4 einer umfassenden Analyse präsentiert. Die zwei Hauptmetriken sind dabei die Speicher- und Laufzeitkosten der unterschiedlichen Allokatoren. Kapitel 5 betrachtet, ob die eingangs gestellten Ziele erreicht wurden und zeigt neue Untersuchungsschwerpunkte sowie unbetrachtete Aspekte eines Allokators auf.

2 GRUNDLAGEN

Um die Schnittstelle (engl. Application Programming Interface (API)) der Speicherverwaltung so einfach und allgemein wie möglich zu halten, besteht sie aus nur vier Grundfunktionen:

- `malloc(size)`, um Speicher einer bestimmten Größe anzufordern
- `free(pointer)`, um erhaltenen Speicher zurück zu geben
- `calloc(nmemb, size)`, um ein mit Null initialisiertes Speicherstück bestimmter Größe zu erhalten
- `realloc(pointer, size)`, um erhaltenen Speicher zu vergrößern, ohne dessen Inhalt zu verändern
- `memalign(alignment, size)`, um Speicher einer bestimmten Ausrichtung zu erhalten

Die Simplizität der Schnittstelle führt aber zu höherer interner Komplexität für allgemeine Speicherverwaltungen, da die unterschiedlichsten Anforderungen und Nutzungsprofile berücksichtigt werden müssen. Implementierungen lassen sich unterscheiden durch die verfolgten **Strategien**, den daraus resultierenden **Vorgehensweisen** und die verwendeten **Mechanismen** [Wil+95].

Strategien versuchen Wissen über das Aufrufprofil zu nutzen. Wohingegen Vorgehensweisen Implementierungsentscheidungen sind, wie und wo Speicherblöcke platziert werden, die sich in den Mechanismen bestehend aus Algorithmen und verwendeten Datenstrukturen manifestieren.

Eine mögliche Strategie, den Zwischenspeicher möglichst effektiv auszunutzen, wäre, dass unterschiedliche Fäden immer auf denselben Speicherbereichen arbeiten. Da dies aber nicht möglich ist, weil Fäden untereinander Speicher austauschen können, ist ein denkbares Vorgehen, immer den zuletzt freigegeben Speicher nach Möglichkeit wieder auszugeben, was auch tatsächlich oft durch den Einsatz von Last In First Out (LIFO) Warteschlangen für freie Speicherblöcke umgesetzt wird [LK98].

Im Folgenden werden einige mögliche Zielsetzungen, die auch beim nachfolgenden Vergleich betrachtet werden, vorgestellt und wie sich diese Zielsetzungen in den vier untersuchten Implementierungen widerspiegeln.

2.1 Ziele und Eigenschaften

Im Laufe der Zeit kamen durch neue Technologien auch neue Dimensionen für Speicherverwaltungen hinzu. So waren 1995 für Wilson noch die zwei einzigen wirklich relevanten Ziele für Allokatoren ihre Geschwindigkeit und ihr Speicherverbrauch [Wil+95]. Wenige Jahre später untersuchten Larson und Krishnan die Skalierbarkeit von Allokatoren für mehrere Fäden und Prozessoren (engl. Central

Processing Unit (CPU)), die entscheidend für die Performanz von mehrfädigen Serveranwendungen ist [LK98].

Unterschiedliche Ziele schließen sich nicht unbedingt aus, allerdings müssen oft Kompromisse eingegangen werden vor allem zwischen Speicherverbrauch und Geschwindigkeit. Eine ausführlichere Buchhaltung oder eine aggressivere Zwischenspeicherstrategie können zu kürzeren Laufzeiten der API Funktionen führen jedoch mit höherem Speicherverbrauch.

2.1.1 Geschwindigkeit

Die Geschwindigkeit ist die intuitivste Eigenschaft jeder Bibliotheksfunktion. Vor allem bei Anwendungen, deren Arbeit zu einem großen Teil aus Speicherverwaltung besteht, wie Serveranwendungen, die für jede eingehende Anfrage mehrere kleine Speicherblöcke aufbauen, die Anfrage beantworten und diese danach wieder abbauen, ist die Geschwindigkeit der einzelnen Funktionsaufrufe von großer Bedeutung.

Die Geschwindigkeit ist gut experimentell messbar, das Problem am experimentellen Messen ist jedoch, dass die Geschwindigkeit das komplexe Ergebnis vieler kleiner Faktoren ist und damit stark vom verwendeten Experiment und den gewählten Rahmenbedingungen abhängt. Faktoren, die die Geschwindigkeit eines `malloc` Aufrufs beeinflussen können sind unter anderem:

- Die Größe des angeforderten Speicherblocks
- Die Anzahl gleichzeitiger Anfragen
- Der innerer Zustand der Speicherverwaltung (ist noch genug bzw. passender Speicher vorrätig oder muss Neuer vom System angefordert werden)
- Hardware spezifische Eigenschaften
- Irrige Mitbenutzung interner Strukturen (mehr dazu in Abschnitt 2.1.5)

Ebenfalls hängt die Geschwindigkeit davon ab, wie viel Speicher man ihr zu Gunsten eintauschen möchte. Der schnellste vorstellbare Allokator ist ein sogenannter “bump allocator”, er gibt für jede Anfrage einen internen Zeiger heraus und erhöht diesen danach um die Größe des angeforderten Blocks. Das Zurücknehmen eines Blocks ist eine Nulloperation [Wil+95]. Dieser Allokator verdankt seine Geschwindigkeit der Tatsache, dass er Speicher grundsätzlich nicht wiederverwendet, was in der Realität natürlich nicht praktikabel ist, da die Anwendung schnell den Rahmen des ihr zur Verfügung stehenden Hauptspeichers sprengen würde und das System unbenutzbar wird.

Von besonderem Interesse sowohl für die aktuelle Forschung als auch für diese Arbeit ist der zweite Punkt. Das Problem von gleichzeitigen Anfragen ist es, den inneren Zustand vor paralleler Nutzung zu schützen, um zu jedem Zeitpunkt einen korrekten Zustand zu gewährleisten. Dies führt uns zum nächsten Ziel.

2.1.2 Skalierbarkeit

Normalerweise werden, um einen konstanten inneren Zustand zu gewährleisten, Zugriffe auf diesen sequentialisiert, heißt es darf nur eine Anfrage gleichzeitig damit arbeiten. Wenn jedoch mehr und mehr parallele Kontexte hinzugefügt werden, um mehr Arbeit zu bewältigen, diese sich dann aber in der Speicherverwaltung sequentialisieren müssen, können sie nicht mehr parallel arbeiten und es kann kein Zugewinn erzielt werden.

Um bessere Skalierbarkeit der Speicherverwaltung zu erhalten, muss versucht werden den für eine Anfrage benötigten inneren Zustand so klein wie möglich zu halten oder dafür zu sorgen, dass es überhaupt keinen geteilten inneren Zustand gibt. Diesen Ansatz verfolgten Larson und Krishnan

bereits 1998 mit ihrem *lkmalloc* genannten Allokator, bei dem sie den gesamten inneren Zustand vervielfachten und ihn in kleinst mögliche Abschnitte (128 Warteschlangen unterschiedlicher Anfragegrößen) unterteilten. Aktuelle Allokatoren für große skalierbare Serversysteme, wie sie zum Beispiel bei Google und Facebook eingesetzt werden, vermeiden sogar für bestimmte *kleine* Anfragegrößen jeglichen geteilten inneren Zustand [Eva11] [GM07]. Dies erreichen sie durch die Separation und exklusive Zuteilung von Speicherbereichen zu einzelnen Fäden (siehe. Abschnitt 2.2.4.1 und Abschnitt 2.2.4.4), was allerdings mit der Gefahr von Speicherverschwendung erkauft werden muss. Um dieses Risikio zu minimieren, werden weitere Techniken eingesetzt, die es ermöglichen zwischen den Fäden Speicher zu verschieben.

2.1.3 Speicherverbrauch

Speicherverbrauch ist die zweite große intuitive Eigenschaft einer Speicherverwaltung. Eine ideale Speicherverwaltung fordert nicht mehr Speicher vom System als die Anwendung maximal zu einem Zeitpunkt benötigt. Aufgrund der Schnittstelle ist dies jedoch nicht möglich, da der Aufruf von `free` keine Informationen über den Speicher beinhaltet und die Speicherverwaltung diese Informationen intern speichern muss.

Das experimentelle Messen der Speichereffizienz verhält sich ähnlich wie das Messen der Geschwindigkeit. Es ist relativ einfach einen Wert zu erhalten, da das Betriebssystem genau weiß, wie viele Speicherseiten es dem Programm zur Verfügung gestellt hat. Jedoch ist es nahezu unmöglich, von außen zu bestimmen, wo und warum eigentlich überflüssiger Speicher benötigt wird.

Neben den Metadaten und Datenstrukturen, die angelegt und verwaltet werden müssen, gibt es noch zwei andere Arten der “Speicherverschwendung”: **Blowup** und **Fragmentierung**.

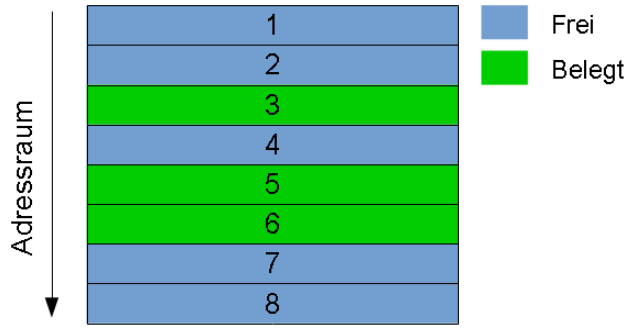
Blowup kann bei Speicherverwaltungen entstehen, die ihren Speicher separieren, diesen Bereichen einen eindeutigen Besitzer zuweisen und keine Möglichkeit haben, Speicher zwischen den getrennten Bereichen zu verschieben. Dies kann dazu führen, dass der Speicher in einem Bereich, dessen Besitzer sein Anfrageprofil geändert hat, nicht wiederverwendet und somit verschwendet wird. Wenn dieses Muster wiederholt auftritt, kann der Speicherverbrauch der Anwendung schnell überproportional wachsen [Ber+00].

Fragmentierung bezeichnet, dass eigentlich freier Speicher nicht wiederverwendet werden kann, auf Grund der Platzierungsstrategie des Allokators. Sie wird klassisch in zwei Kategorien unterteilt.

2.1.3.1 Interne Fragmentierung

Unter interner Fragmentierung versteht man den Verschnitt, der durch den Größenunterschied zwischen Anforderung und herausgegebenen Speicherstück entsteht [Wil+95]. Dieser Effekt tritt bei jeder Implementierung auf, da die Schnittstelle garantiert, dass herausgegebener Speicher geeignet ausgerichtet sein muss, um ein beliebiges darin enthaltenes Objekt über einen Zeiger auf den Beginn des Blocks verwenden zu können [Gro13]. Der Allokator der GNU C Standardbibliothek zum Beispiel gibt keine kleineren Blöcke heraus als das Vierfache der Größe eines `size_t`, wobei ein `size_t` pro Block für Metadaten benötigt wird [DO]. Das bedeutet, wenn viele Objekte kleiner als die Mindestgröße angefordert werden, wird deutlich mehr Speicher vom System angefordert und ausgegeben als tatsächlich benötigt. Die interne Fragmentierung beträgt dann $N * (S - s)$ wobei N die, Anzahl der Blöcke, S die passende Blockmindestgröße und s die geforderte Größe ist.

Abbildung 2.1 – Die Platzierung der belegten Blöcke verhindert, eine Anfrage, die drei oder mehr Blöcke benötigt, zu erfüllen, obwohl ausreichend Speicherplatz vorhanden wäre (externe Fragmentierung).



2.1.3.2 Externe Fragmentierung

Als externe Fragmentierung bezeichnet man die Zerstückelung des zur Verfügung stehenden Speichers, die durch die fixe Platzierung von unterschiedlich großen Speicherblöcken im Adressraum entsteht [Wil+95]. Belegte Speicherblöcke dürfen nicht angefasst werden, um sie etwa besser zu platzieren, weil die Anwendung die Garantie hat, dass sich die Adresse einmal erhalten nicht verändert. Sie entsteht, wenn die Anwendung ihr Anfrageprofil verändert oder immer nur vereinzelte Blöcke wieder freigegeben werden [Wil+95].

Abbildung 2.1 zeigt wie durch externe Fragmentierung ein erhöhter Speicherbedarf als eigentlich nötig entsteht. Strategien, um externe Fragmentierung zu vermeiden, werden in Abschnitt 2.2 genauer erläutert.

2.1.4 Erhöhung der Datenlokalität

Lokalität von Daten bedeutet, dass durch eine geschickte Platzierung im Speicher die bei der Benutzung enstehenden Zugriffskosten möglichst gering sind. Oft heißt das, dass zusammen genutzte Daten nahe beieinander im Adressraum liegen. Der Vorteil dieser Nähe im Speicher ist, dass Zwischenspeicherfehlzugriffe und Seitenfehler reduziert werden. Dies führt, ähnlich wie die Vermeidung der irrigen Mitbenutzung, zu höheren Geschwindigkeiten sowohl der Anwendung als auch des Allokators. Über die reine Nähe im Adressraum hinaus spielt für die Datenlokalität die weitere Topologie der Hardware eine nicht zu vernachlässigende Rolle. Larson und Krishnan haben gezeigt, dass die Geschwindigkeit und Skalierbarkeit eines Allokators auf mehreren Prozessoren nicht nur durch die Größe und Anzahl der kritischen Abschnitte sondern auch durch die gemeinsame Nutzung von Daten auf unterschiedlichen Prozessoren geprägt wird. Unabhängig davon ob der innere Zustand des Allokators im Speicher eng beieinander liegt, können die Zwischenspeicher der Hardware nicht optimal genutzt werden, da Maschinenwörter häufig zwischen den Speichern der verschiedenen Prozessoren hin und her flattern [LK98]. Mitunter um diesen von ihnen als “cache slothing” bezeichneten Effekt zu vermeiden, vervielfältigen sie den gesamten inneren Zustand der Speicherverwaltung. Über die Betrachtung von physikalischen Zwischenspeichern hinaus könnten auch Unterschiede in den Zugriffskosten wie bei der im Testsystem verwendeten Non-Uniform Memory Access (NUMA) Architektur in die Strategie eines Allokators einfließen. Die Autoren der

Speicherverwaltung der *glibc* sprechen sich allerdings gegen derartige Maßnahmen aus mit der Hoffnung, dass der Betriebsystemkern ausreichende Vorkehrungen trifft [DO].

Die Datenlokalität des Allokators und der Anwendung lässt sich über die durch das Betriebssystem gemessene Anzahl an Zwischenspeicher- und Seitenfehlern experimentell bestimmen.

2.1.5 Vermeidung von irriger Mitbenutzung

Die letzte untersuchte Eigenschaft ist die irrige Mitbenutzung (engl. false sharing). Sie entsteht dann, wenn Fäden auf unterschiedlichen Prozessoren mit von einander unabhängigen Daten in derselben physikalischen Zwischenspeicherzeile schreibend arbeiten, wie in Abbildung 2.2 zu sehen. Dann müssen die Prozessoren, um den Besitz der Zeile konkurrieren und sicherstellen, dass eine Änderung auch im Zwischenspeicher aller beteiligten Prozessoren sichtbar ist. Dieses Austauschen von Zwischenspeicherzeilen benötigt Zeit, die sich sowohl auf die Geschwindigkeit des Allokators, wenn es sich um allokatorinterne Daten handelt, als auch auf die Geschwindigkeit des Programms auswirkt.

Ein Programm kann selbst für eine geteilte Zwischenspeicherzeile sorgen, indem sie Teile einer Zeile an einen anderen Faden weitergibt. Dies der Allokator nicht verhindern, außer dadurch Blöcke nur an der Zeilengröße ausgerichtet auszugeben, was aber zu enormer Fragmentierung führen würde [Ber+00]. Allerdings kann die Speicherverwaltung sicher stellen, dass sie *selbst* nicht aktiv für irrige Mitbenutzung sorgt, indem sie Teile einer Zeile nicht an mehrere Fäden ausgibt. Ebenfalls kann sie verhindern, dass Zeilenteile nach einem `free` nicht an einen anderen Faden ausgegeben werden. Dies würde zu einer passiven Verursachung von irriger Mitbenutzung führen [Ber+00].

2.2 Verbreitete Vorgehensweisen

Es sind einige wichtige grundlegende Vorgehensweisen eines Allokators zu kennen, um sein tatsächlich beobachtetes Verhalten zu verstehen und zu vergleichen.

- **Muster der Speicherwiederverwendung**: Werden gerade freigegebene Speicherstücke bevorzugt wiederverwendet? Werden Blöcke aus derselben Speicherregion für ähnliche Grö-

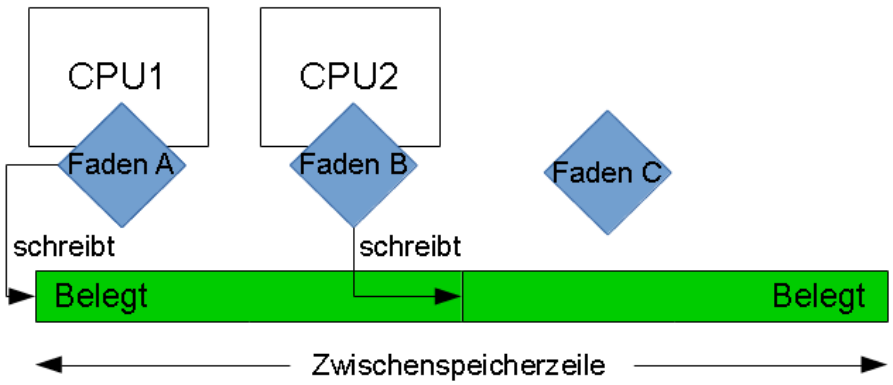


Abbildung 2.2 – Zwei Speicherblöcke von Fäden auf unterschiedlichen Prozessoren teilen sich eine physikalische Zwischenspeicherzeile. (In Anlehnung an [Eva06])

ßenanfragen genutzt? Werden freie Speicherblöcke aus einem ähnlichen Speicherbereich bevorzugt?

- **Teilen und Verschmelzen**: Werden Blöcke geteilt und wieder verschmolzen? Zu welchem Zeitpunkt (Abschnitt 2.2.3)?
- **Auswahl des Blocks**: Wird immer ein Block mit möglichst genau passender Größe verwendet oder bevorzugt Blöcke, die irgendwie anders mit der Anfragegröße in Verbindung stehen?
- **Schwellenwerte**: Wie oft werden Blöcke aufgeteilt? Wie viel Speicher wird im Allokator vorgehalten?

Oft sind diese Vorgehensweisen ein Kompromiss aus der Simplizität der Implementierung, der Geschwindigkeit sowie der akzeptierten Fragmentierung.

Viele grundlegende Mechanismen finden sich in nahezu allen Allokatoren wieder und ermöglichen es, bestimmte Vorgehensweisen sehr effizient umzusetzen.

2.2.1 Grundlegende Mechanismen

- **Kopfvermerke (engl. header)**: Wie bereits erwähnt muss die Speicherverwaltung selbst buchhalten, wie groß die verwendeten Speicherblöcke sind. Dazu bieten sich je nach Architektur ein oder zwei Maschinenworte zu Beginn des jeweiligen Blocks an. In ihnen wird meist die Größe des Blocks und noch weitere oft boolsche Werte hinterlegt wie Abbildung 2.3 zeigt. Gespeicherte Informationen können die Zugehörigkeit zu einem Speicherbereich, der aktuelle Zustand und die Beziehung zu den Nachbarblöcken sein [DO; Wil+95]. Allerdings nimmt bei dieser Art der Metadatenspeicherung der zusätzliche Speicherverbrauch proportional mit der Anzahl an Allokationen zu. Deshalb externalisieren moderne Allokatoren die Metadaten, indem sie größere Speicherbereiche in kleinere Stücke fester Größe zerlegen. Die Größe eines spezifischen Blocks wird dann über die Maskierung seiner Adresse und den Vermerk im beherbergenden Speicherstück bestimmt.
- **Grenzvermerk (engl. Boundary tag)**: Um zwei Blöcke einfach miteinander verschmelzen zu können, ist es sinnvoll den Kopfvermerk noch einmal an das Ende des Blocks zu setzen, damit man leicht vom darunterliegenden Nachbarn den Start finden kann. Grenzvermerke werden praktischerweise nur bei freien Blöcken benötigt und führen so nicht zu zusätzlichem Speicherverbrauch.

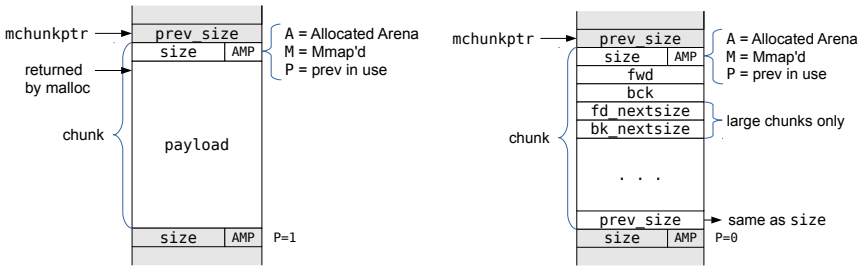


Abbildung 2.3 – Der linke Speicherblock ohne Grenzvermerk ist in Benutzung durch die Anwendung. Der Rechte ist frei und beinhaltet für die Speicherverwaltung nützliche Informationen [DO].

- **Referenzierung innerhalb von Speicherblöcken**: Da jegliche Speicherverwaltung irgendwie vermerken muss, welche freien Speicherstücke existieren, bietet es sich an, den Speicher innerhalb der freien Blöcke für eine Verzeigerung zu nutzen. Ob die Verzeigerung nun für eine verkettete Liste oder eine Baumstruktur verwendet wird, ist dabei unrelevant, da grundsätzlich kein zusätzlicher Speicheraufwand entsteht. Allerdings wirkt sich diese Verzeigerung auf die Mindestgröße der Blocks aus. Zum Beispiel ist die bereits erwähnte Mindestgröße der *glibc* durch zwei Vermerke und zwei Zeiger für eine doppeltverkettete Liste auf $4 * \texttt{size\_t}$ limitiert [Wil+95; DO].

- **Unterschiedliche Handhabung von Blöcken verschiedener Größenordnungen**: Nahezu alle Allokatoren setzen eine Obergrenze für die von ihnen verwalteten Blöcke und lassen alle Anforderungen überhalb dieser Marke direkt vom System bedienen und geben diese bei einer Deallokation auch direkt wieder an das System zurück. Auch die Tatsache, dass Anwendungen normalerweise deutlich mehr kleinere als größere Speicherblöcke anfordern [Wil+95], führt dazu, dass unterschiedliche Größen fast immer auf verschiedene Weisen behandelt werden. Um dieses Muster auszunutzen, werden zum Beispiel kleinere Blöcke aggressiver zwischengespeichert und nicht so schnell verschmolzen.

- **Sonderbehandlung des letzten Blocks**: Diese folgende Technik wird nur von Halden benötigt die mit Hilfe des Systemaufrufs `brk()` vergrößert oder verkleinert werden, da mit `brk` nur das Ende der Halde verändert werden kann. Um die Buchhaltung des zur Verfügung stehenden Speichers zu erleichtern und zusammenhängenden Speicher wieder ans System zurückgeben zu können, wird der letzte Teil der Halde verschmolzen und als einzelner Block gesondert verwaltet. Von diesem Block wird bei Bedarf Speicher für Allokationen der Anwendung abgeschnitten oder Endstücke, wenn eine gewisse Größe erreicht wurde, wieder ans System zurückgegeben.

2.2.2 Auffinden eines geeigneten Blocks

Um in den verzeigerten Datenstrukturen einen Block auszuwählen, gibt es mehrere mögliche Vorgehensweisen:

- **Best-Fit**: Der am besten zur Größenanforderung passende Block wird gesucht und benutzt. Je nach Datenstruktur müssen eventuell alle verfügbaren Blöcke untersucht werden, um den Besten zu finden.

- **First-Fit**: Der erste gefundene Block, der die Größenanforderung erfüllt, wird verwendet. Dies führt eventuell zu erhöhter Fragmentierung oder zu mehr Aufwand für das Teilen und Einsortieren des übrigen kleineren Blocks.

- **Next-Fit**: Eine gängige Optimierung von First-Fit, bei der sich die Position des zuletzt verwendeten Blocks gemerkt und ab ihr weiter gesucht wird.

Die Verfahren wurden für die Verwendung mit einer einzelnen verketteten Liste entwickelt und untersucht [Wil+95]. Mit der Auswahl der richtigen Datenstruktur bezeichnen diese Verfahren oft dasselbe. So wird zum Beispiel bei einer der Größe nach sortierten Suchstruktur First-Fit und Best-Fit denselben Block liefern.

2.2.3 Teilen und Verschmelzen

Die Grundidee von Teilen und Verschmelzen ist es, überflüssig große Blöcke für kleinere Anfragen aufzuteilen und zurückgegebene Blöcke mit ihren ebenfalls unbelegten Nachbarblöcken zu einem größeren Block zu verschmelzen. Der Vorteil von großen Blöcken ist, dass sie ein breiteres Spektrum an Anfragen erfüllen können. Außerdem wird es für den Designer des Allokators einfacher, eine gewünschte Strategie umzusetzen, dadurch dass immer größt mögliche Speicherblöcke zur Verfügung stehen [Wil+95]. Sehr bekannte und in der Lehre oft vermittelte Verfahren, um Blöcke zu teilen und wieder zu verschmelzen, sind die sogenannten **Buddy Systeme**. Diese Verfahren halbieren stets die ihnen zur Verfügung stehenden Blöcke. Die resultierenden zwei Blöcke werden Buddies zu deutsch Freunde oder Partner genannt. Verschmolzen werden nur Partnerblöcke, was den Schritt des Teilens rückgängig macht. Der Vorteil dieses Vorgehens ist, dass der Partnerblock sehr leicht durch eine einfache Operation auf der Speicheradresse gefunden werden kann und das Verschmelzen damit schnell und einfach funktioniert. Außerdem benötigt dieses Verfahren theoretisch kaum mehr Speicher für Metadaten, da die Adresse quasi die Größe enthält und nur vermerkt werden muss, ob der Speicher gerade in Benutzung ist. Dies kann durch ein einzelnes Bit pro Block gewährleistet werden. In der Praxis wird jedoch mindestens ein Maschinenwort als Kopfvermerk verwendet, da kein einzelnes Bit von einem Objekt verwendet werden darf, um die garantierte Ausrichtung des Objekts nicht zu verletzten. Blöcke bestimmter Größe werden in einer Liste gehalten, um schnell bei einer Anfrage Blöcke aufgerundet auf die nächst größere Größenklasse zu finden [Wil+95]. Dieses Verfahren wird trotz seiner Beliebtheit in der Lehre von keinem untersuchten Allokator verwendet, da es zu größerer Fragmentierung führt als andere flexiblere Teile-Verschmelze-Verfahren [Wil+95].

2.2.4 Separierter Speicher

Das vermutlich grundlegendste Prinzip aller Allokatoren, die aus mehr als einer verketteten Liste bestehen, ist die logische Separation von Speicher. Das muss nicht, kann aber bedeuten, dass dieser Speicher auch physikalisch von einander getrennt ist. Die Separation erstreckt sich von einzelnen nebeneinander liegenden Blöcken bis hin zu gänzlich multiplizierten Allokatoren.

2.2.4.1 Größenklassen

Die gängigste Unterscheidung von Speicher (alle untersuchten Allokatoren nutzen sie) findet in zur Übersetzungszeit festen Größenklassen statt, die in ihren eigenen Strukturen verwaltet werden. Diese Teilung ermöglicht es mehrere parallele Anfragen für unterschiedliche Größenklassen auch tatsächlich parallel zu beantworten, da nicht zwingend geteilte Strukturen genutzt werden müssen [LK98]. Außerdem werden die Suchstrategien von passenden Blöcken deutlich schneller, weil die Anzahl an zu überprüfenden Blöcken durch den systematischen Ausschluss unpassender Größenklassen deutlich geringer wird. Falls auch eine physikalische Trennung der unterschiedlichen Größenklassen vorliegt, das heißt, dass Anfragen einer Größenklasse aus kontinuierlichen Speicherstücken bedient werden, kann auf Kopfvermerke verzichtet werden. In diesem Fall reicht es aus, die Größenklasse des kontinuierlichen Speicherstücks zu vermerken, woraufhin die Adresse des Blocks seine Größe verrät. Bei diesem Vorgehen ist allerdings ein Verschmelzen und Teilen von Blöcken nicht mehr möglich.

2.2.4.2 Arenen

Larson und Krishnan haben in ihrer Untersuchung zu Serveranwendungen gezeigt, dass es nicht ausreicht, den kritischen Abschnitt auf eine Größenklasse zu reduzieren, um für gute Skalierbarkeit zu

sorgen, da gemeinsame Daten zwischen den Zwischenspeichern der CPUs wandern müssen [LK98]. Ihre Lösung für dieses Problem ist es, den gesamten inneren Zustand des Allokators, das heißt alle verschiedenen Freispeicherstrukturen, zu vervielfachen und jedem eine exklusive Halde zuzuweisen. Auf die resultierenden Konstrukte, Arenen genannt, werden die Fäden aufgeteilt und müssen somit nur noch mit den anderen Fäden in ihrer Arena um die kritischen Abschnitte konkurrieren. Dieses Konzept findet sich in *jemalloc* sowie in der *glibc*.

2.2.4.3 Prozessor lokaler Speicher

Eine Weiterführung des Arenenprinzips ist es, eine oder mehrere Arenen pro Prozessor zu verwalten. Das funktioniert sehr gut in Kontexten, in denen es nicht mehr Fäden geben kann als CPUs, dies ist besipielsweiße der Fall im Betriebssystemkern. Jedoch scheitert dieser Ansatz im Anwendungskontext, da diese Annahme nicht zutrifft und Fäden mitunter auch Prozessor wechseln können [LK98]. *Hoard* schafft zumindest auf *Solaris* Systemen eine kollisionsfreie und eindeutige Zuordnung, indem die Light-weight Process (LWP) Identifikationsnummer genutzt wird, die normalerweise die Nummer des benutzen Prozessors ist [Ber+00].

Die Trennung in prozessorlokale Speicherbereiche bietet die Möglichkeit strukturell durch den Allokator eingeführte irrige Mitbenutzung zu vermeiden, weil Speicherzeilen nur an Fäden auf demselben Prozessoren ausgehändigt werden [Ber+00]. Außerdem wird somit die Lokalität der Daten erhöht, was aufgrund der besseren Zwischenspeichernutzung zu höheren Geschwindigkeiten führt.

2.2.4.4 Faden lokaler Speicher

Die wohl bedeutendste logische Trennung von Speicher für die Geschwindigkeit und Skalierbarkeit dynamischer Speicherverwaltungen ist die Separation in fadenlokale Bereiche. Fadenlokalität wird mit speziellem Speicher, Thread-local Storage (TLS) genannt, ermöglicht, von dem der Übersetzer mit Hilfe der Laufzeitumgebung durch automatisch erzeugte Programmteile garantiert, dass er einmalig und exklusiv für jeden Faden ist. In C und C++ ist diese Art von Speicher mit dem Attribut `thread_local` benutzbar. Alternativ bieten oft auch Bibliotheken für parallele Fäden wie *pthread* mit der Funktion `pthread_key_create` ähnlichen Speicher an, dessen Benutzung jedoch mit mehr Aufwand verbunden und daher teurer ist. In TLS kann nicht nur ein Zeiger auf die zugeordnete Arena gespeichert werden, was die Implementierung einer festen Zuordnung deutlich effizienter macht [Eva06], sondern auch eigens für diesen Faden angelegte Suchstrukturen. Da TLS nicht mit anderen Fäden geteilt wird, kann gänzlich auf Synchronisation innerhalb dessen verzichtet werden. Das Wegfallen von Synchronisation beschleunigt laut den Autoren von *tcmalloc* die durchschnittliche Zeit eines `malloc` Aufrufs auf unter 50 Nanosekunden auf einem 2.8 Ghz Pentium 4 Prozessor. Ein einzelner Sperr- und Entsperrzyklus dauert im Vergleich dazu bereits 100 Nanosekunden [GM07].

2.3 Untersuchte Implementierungen

Im Folgenden werden kurz die Ziele, Eigenschaften und inneren Funktionsweisen der untersuchten Speicherverwaltungen vorgestellt. *Glibc* und *jemalloc* sind über die Zeit gewachsen und vereinen in sich verschiedene Entwicklungen und neue Erkenntnisse der Forschung. *Tcmalloc* und *Hoard* sind von Grund auf nach bestimmten Motivationen designte Systeme. Wie gut sie die von einem Allokator gewünschten oder angestrebten Ziele umsetzen, ist in Kapitel 4 zu finden.

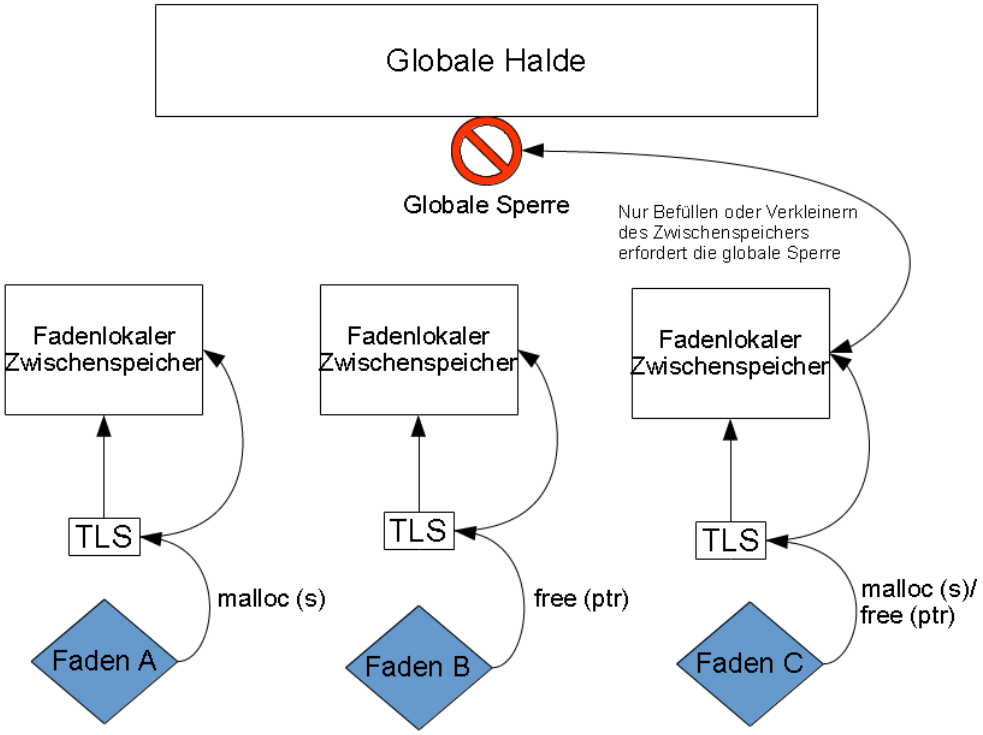


Abbildung 2.4 – Unsynchronsierte Verwendung einer Speicherverwaltung mit fadenlokalen Zwischenspeichern durch drei parallele Fäden

2.3.1 Hoard

Hoard ist ein Allokator der besonders mit Blick auf die Vermeidung von durch die Speicherverwaltung verursachte irrige Mitbenutzung, eine garantierte Obergrenze für den Speicherverbrauch bei parallelen Anwendungen sowie gute Skalierbarkeit designend wurde.

All diese Ziele werden durch die Entscheidung erreicht, Speicher separiert in einer globale Halde und einer bzw. zwei pro Prozessor zu verwalten. Jedem Faden wird per Hash-Funktion eindeutig eine Halde zugeordnet. Diese Halden bestehen aus Superblöcken fester Größe (ein Vielfaches der Seitengröße), einer Sperre sowie mehreren Listen von Superblöcken und einem Zähler für die Menge an in ihr enthaltenen Speichers.

Superblöcke wiederum sind Felder aus Blöcken einer bestimmten Größenklasse, einer Sperre und einer verketteten LIFO Liste (für höhere Lokalität) von freien Speicherblöcken. Des weiteren besitzen sie eine Füllstandsmarke und sind mit anderen Superblöcken ähnlichen Füllstands verkettet.

Allokationen größer als ein halber Superblock werden direkt vom System angefordert. Andernfalls wird zuerst die eigene Halde gesperrt und nach einem Superblock mit passender Größenklasse durchsucht. Der vollste Gefundene wird verwendet. Wenn kein passender Superblock gefunden wurde, wird falls vorhanden aus der globalen Halde oder vom System ein neuer Superblock in die eigene Halde überführt und die Buchhaltung der Halde aktualisiert. Danach wird aus dem neuen oder gefundenen Superblock ein Block zurückgegeben und die Füllstandsmarke des Superblocks angepasst.

Große Deallokationen gehen direkt wieder an das System zurück. Andernfalls wird zuerst der passende Superblock über die Adresse des zurückgegebenen Speichers und dessen Halde ermittelt. Dann wird beides gesperrt, der Block wieder in den Superblock eingehängt und dessen Füllstandsmarke aktualisiert. Falls es sich bei der Halde um eine Prozessor-Halde handelt und sie daraufhin einen zu geringen Füllstand aufweist, wird ein leerer oder fast leerer Superblock an die globale Halde zurückgegeben und die Buchhaltung aktualisiert. Schließlich werden noch die beiden Sperren aufgehoben.

Die Buchhaltung der Füllstände und das Wiederzurückgeben von Superblöcken an die globale Halde verhindert, dass sich viel ungenutzter Speicher in den Prozessor-Halden sammelt. Die Autoren beweisen, dass der maximal vorgehaltene Speicher dadurch linear ($O(1)$) und somit unabhängig von der Anwendung ist. Dadurch kann es nicht zu Blowup kommen.

Die exklusive Zuweisung von Halden an Fäden führt zu geringem Wettbewerb um die Sperren und somit zu keinen oder geringen Wartezeiten vor den kritischen Abschnitten, wodurch gute Skalierbarkeit erreicht wird.

Außerdem soll damit die irrige Mitbenutzung vermieden werden, da nur ein Faden Speicher aus demselben Superblock erhält und zurückgegebener Speicher immer in den ursprünglichen Superblock eingefügt wird [Ber+00]. Diese Annahme stimmt jedoch nur, wenn es eine 1:1 Zuordnung von Fäden auf Halden gibt, was nicht immer möglich ist, wenn zum Beispiel mehr Fäden als Prozessor-Halden existieren oder die Hash-Funktion zu Kollisionen führt. Leider gehen die Autoren von *Hoard* nur sehr kurz auf diese als Designziel ausgeschilderte Eigenschaft ein.

2.3.2 tcmalloc

tcmalloc wurde im Rahmen einer Sammlung von Werkzeugen zur Leistungssteigerung von hauptsächlich Serveranwendungen (gperftools) bei Google entwickelt. Die Motivation für eine Eigenkreation war es, einen schnelleren und besser skalierenden Allokator als den der *glibc* zu haben. Darüberhinaus sollten weiter für Google wichtige Werkzeuge integriert werden. So bietet *tcmalloc* zum Beispiel die Möglichkeit, Speicherfehler zu finden oder das Anforderungsprofil einer Anwendung mit zu schneiden. Außerdem leidet *tcmalloc* nicht unter dem Blowup, der bei Google für manche Anwendungen mit dem Allokator der *glibc* in der Version 2.23 festgestellt wurde [GM07].

Die fundamentalen Verhaltensweisen von *tcmalloc* sind, dass Speicher sowohl in Größenklassen als auch in fadenlokale Bereiche separiert wird sowie, dass Speicher aus fadenlokalen Bereichen nach einer anpassbaren Heuristik wieder für andere Fäden zur Verfügung gestellt wird.

Die verwaltete Halde besteht aus kontinuierlichen Speicherstücken, die aus einer oder mehreren Speicherseiten bestehen. Diese Stücke (engl. span) werden, falls sie unbelegt sind, in einem globalen Allokator in 128 Listen von spans mit $i + 1$ umspannten Seiten oder einem Rot-Schwarz-Baum für größere Stücke verwaltet. Belegte spans sind entweder als große Objekte in Benutzung oder aufgeteilt in mehrere kleinere Blöcke, deren feste Größe im span vermerkt ist. Eine Zuordnung von Speicherseiten zu umspannenden span erfolgt über einen Patricia-Trie.

Tcmalloc unterscheidet grundlegend drei Arten von Allokationen:

- **Klein ($s < 256K$)**: Der Art wie kleine Allokationen verwaltet werden, verdankt der Allokator seinen Namen. Sie werden nämlich in fadenlokalen Zwischenspeichern (engl. thread cache → tc) gehalten und nur falls dieser leer oder zu voll ist, aus beziehungsweise in den globalen Zwischenspeicher bewegt. Diese Zwischenspeicher bestehen aus Listen für jede von ca. 88 verschiedenen Größenklassen, welche mit zunehmender Größe auch zunehmende Abstände aufweisen. Jede Liste merkt sich außerdem ihre maximal erlaubte Länge. Für jede Allokation wird zuerst die passende Größenklasse ermittelt und im eigenen fadenlokalen

Zwischenspeicher gesucht. Falls ein Block gefunden wurde, wird dieser zurückgegeben und die Maximalgröße der Liste erhöht. Zu beachten ist, dass dieser Pfad im Algorithmus keine Sperre benötigt, was einen signifikanten Geschwindigkeitsvorteil darstellt und für gute Skalierbarkeit sorgt. Ein leerer lokaler Zwischenspeicher wird mit einer festen Anzahl an Blöcken der passenden Größe aus dem globalen Zwischenspeicher gefüllt und, falls dieser ebenfalls leer ist, wird aus dem Seitenallokator ein Span angefordert und dieser in die entsprechende Größenklasse unterteilt sowie in den globalen Speicher eingefügt.

- **Mittel (**$256K <= s <= 1MB$**)**: Diese werden direkt aus den Listen des Seitenallokators erfüllt. Falls in allen Listen geeigneter Größe kein span zu finden ist, wird die Anfrage als *groß* behandelt.

- **Groß (**$s > 1MB$**)**: Große Anfragen werden direkt per Best-Fit aus dem Baum des Seitenallokators bedient und überflüssige Seiten in die kleineren Listen eingehängt. Falls nicht genug Speicher vorhanden ist, wird zusätzlicher Speicher vom System angefordert.

Um *Blowup* zu vermeiden, wird bei jeder Deallokation von kleinen Objekten überprüft, ob die tatsächliche Länge größer ist als ihre maximal erlaubte Länge, wenn dies zutrifft, werden wieder dieselbe feste Anzahl an Blöcken in den globalen Zwischenspeicher bewegt. Ein Fall bei dem dieses Verhalten eintritt, ist zum Beispiel ein Erzeuger-Verbraucher-Szenario.

Außerdem findet eine automatische Speicherbereinigung statt, falls ein lokaler Zwischenspeicher über eine festgelegte Gesamtspeichergröße wächst. Während dieser Speicherbereinigung werden neue Maximalgrößen für die einzelnen Zwischenspeicher ausgehandelt, um allen Fäden einen passend großen Zwischenspeicher bereitzustellen und gleichzeitig nicht zu viel Speicher zu verschwenden. [GM07]

Tcmalloc sorgt dadurch, dass die einzelnen Blöcke keine Kopfvermerke benötigen, da ihre Größe im span hinterlegt ist und sie niemals verschmolzen werden, für geringen zusätzlichen Speicherbedarf. Das Ziel des hohen Durchsatzes auch bei vielen parallelen Fäden wird durch die Vermeidung von Sperren erreicht. Diese Technik der fadenlokalen Zwischenspeichern wird mittlerweile auch von vielen anderen Allokatoren in geringerem Umfang verwendet. Durch die automatische Speicherbereinigung und das Zurückgeben von Speicher bei der Deallokation wird das Problem des Blowups erfolgreich verhindert.

2.3.3 glibc's malloc

Der in der aktuellen GNU C Bibliothek (Version 2.28) verwendete Allokator ist von allen Untersuchten mit Sicherheit der am häufigsten Verwendete, da die glibc die Standard C Bibliothek für die meisten Linux Systeme ist. Ebenfalls ist er der Älteste. Er geht auf den Allokator *dlmalloc* in Version 2.7 von Doug Lea aus dem Jahr 2001 zurück, welcher später von Wolfram Gloger unter dem Namen *ptmalloc2* sicher für die Verwendung durch mehrere parallele Fäden gemacht wurde. Seit 2001 ist der Code immer wieder verändert worden und weist immer weniger Gemeinsamkeiten mit dem ursprünglichen *ptmalloc2* auf. Die aktuelle Version nutzt Kopfvermerke und Grenzvermerke wie in Abbildung 2.3 zu sehen, Arenen, fadenlokale Zwischenspeicher[1] sowie vier verschiedene Arten von Freispeicherlisten für Blöcke unterschiedlicher Größen.

Kleine Blöcke liegen in 64 festen Größenklassen der Form $idx * 2 * \texttt{sizeof(size\_t)} + 3 * \texttt{sizeof(size\_t)}$ Bytes (maximal 1032 64bit / 516 32bit) vor. Große Blöcke sind Speicherstücke

[1] Fadenlokale Zwischenspeicher wurden 2017 mit dem git commit `d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc` von DJ Delorie eingeführt.

beliebiger Länge, die weder klein noch größer als der Schwellenwert (standardmäßig 131072 Bytes) für `mmap` sind.

Arenen werden aufgebaut, wenn ein Faden für eine Allokation keine ungesperrte Arena vorfindet. Da es keinen Mechanismus gibt einmal angelegte, aber danach schlecht oder unbenutzte Arenen aufzuräumen oder zu verkleinern, entsteht hier die Möglichkeit für Blowup. Arenen besizen eine eigene Halde und die folgenden Freispeicherlisten (engl. bins):

- **Fast**: Eine einfach verkettete Liste pro kleiner Größenklasse. Auf diese "schnellen" Listen kann atomar zugegriffen werden, ohne die Arena sperren zu müssen.

- **Unsorted**: Eine Liste, in der zuerst alle Blöcke, die zurückgegeben werden, landen, um später weiter sortiert und verschmolzen zu werden.

- **Small**: Eine doppelt verkettete Liste pro kleiner Größenklasse. Doppelte Verkettung wird genutzt, da Blöcke in diesen Listen verschmolzen werden. Dafür müssen sie einfach aus der Mitte einer Liste entfernt werden können.

- **Large**: Weitere 64 doppelt verkettete Listen für große Blöcke. Die Blöcke sind nach aufsteigender Größe sortiert, um effizient Best-Fit zu ermöglichen. Um schnell die passende Größe zu finden, sind außerdem die ersten Blöcke unterschiedlicher Größe miteinander verzeigert. Dazu werden die Felder `df_nextsize` und `bk_nextsize`, wie in Abbildung 2.3 zu sehen, verwendet [DO].

Die Unterscheidung in kleine und große Blöcke und deren unterschiedlichen Listen, sowie das Teilen und Verschmelzen wird bereits in dem ursprünglichen *dlmalloc* verwendet [LG96].

```
1:  if block ist klein then
2:     if Platz in fadenlokaler Zwischenspeicher then
3:        Füge Block in Zwischenspeicher ein
4:     else
5:        Füge Block in fast Liste ein
6:     end if
7:  else if not Block ist Speicherabbildung per mmap then
8:     Verschmelze Block mit benachbarten freien Blöcken
9:     if not Block ist letzter Block then
10:       Füge Block in unsorted List ein
11:    end if
12:    if Block > Verschmelz Schwellenwert (65536 Bytes standartmäßig) then
13:       Verschmelze Blöcke in "fast" Listen
14:    end if
15:    Versuche Teile des letzten Block ans System zurückzugeben
16: else
17:    unmap Block
18: end if
```

Algorithmus 2.1 – `free(block)` der *glibc* [DO]

```
1: if size > mmap Schwellenwert then
2:    return Block per mmap vom System
3: else if size ist klein then
4:    if Block in fadenlokalem Zwischenspeicher then
5:       return Block
6:    else if Block in passender "small" Liste then
7:       Befülle den Zwischenspeicher vorsorglich mit Blöcken gleicher Größe falls vorhanden
8:       return Block
9:    end if
10: else
11:    Bewege alle Blöcke aus den "fast" Listen in die "unsorted" Liste
12:    Verschmelze dabei falls möglich
13: end if
14: while Blöcke in "unsorted" Liste and Iterationen < 10000 do
15:    Sortiere Blöcke aus der "unsorted" Liste in die passenden "small"/"large" Listen
16:    Verschmelze dabei falls möglich
17:    if Block == s gefunden then
18:       return Block
19:    end if
20: end while
21: if size ist groß then
22:    while not Block > s gefunden do
23:       Suche Block per Best-Fit in "large" Listen
24:       Teile gefundenen Block und sortiere übrigen Teilblock in "small"/"large" Listen ein
25:       return Block
26:    end while
27: end if
28: if Blocks in "fast" Liste (möglich wenn s klein) then
29:    Springe zu Zeile 11
30: end if
31: Trenne s vom letzten Block der Arena und vergrößere diesen falls nötig mit Speicher vom System
```

Algorithmus 2.2 – `malloc(size)` der *glibc* [DO]

Wie in Algorithmus 2.2 und Algorithmus 2.1 zu sehen ist, verfügt der Allokator der *glibc* über mehrere Stufen der Geschwindigkeitsoptimierung. Messungen (mehr zu den verwendeten Vergleichstest in Abschnitt 3.2.2.2) des Entwicklers DJ Delorie zeigen, dass fadenlokale Zwischenspeicher und "fast" Listen nicht redundant sind, sondern sich ergänzen und zu gesteigerter Leistung im Vergleich zu einem oder keinem von beiden Verfahren führen[Del17].

Bei einer Reallokation kann auf das teure Kopieren des Speicherinhalts in einen neuen größeren Speicherbereich verzichtet werden, falls der Block entweder per `mmap` vom System stammt und das System `mremap` unterstützt oder wenn der Block mit ausreichend benachbarten freien Blöcken verschmolzen werden kann, um die neue Größe zu erreichen. Der zweite Fall funktioniert trotz der Separation in Größenklassen, da diese nur logisch, nicht aber physikalisch im Speicher separiert sind. Dies macht es möglich, dass ein Block seine Größenklasse ändert.

2.3.4 jemalloc

Die Designziele von *jemalloc* ähneln sehr denen von *tcmalloc*. Zusätzlich soll *jemalloc* aber auch sehr deterministisch sein und enge Grenzwerte für den zusätzlichen Speicherverbrauch gewährleisten. Außerdem soll möglichst bereits benutzter (engl. dirty) Speicher wieder verwendet werden, bevor noch ungenutzter Speicher angefasst wird.

Für Geschwindigkeit sorgen dieselben fadenlokalen Zwischenspeicher wie bei *tcmalloc* mit dem Unterschied, dass automatische Speicherbereinigung nicht bei Bedarf sondern periodisch durchgeführt wird. Dadurch entfällt viel Buchhaltung. Außerdem ist die Anzahl an auf einmal bewegten Objekten nicht statisch, sondern wird zur Laufzeit an die Anwendung angepasst [LJR14].

Die Zuweisung von Fäden auf die vier Arenen pro verfügbarer CPU ist fest und nicht dynamisch wie bei der *glibc* und im Gegensatz zu *Hoard* erfolgt sie nicht über eine Hash-Funktion, sondern reihum, um eine möglichst gleichmäßige Nutzung der Arenen zu garantieren [Eva06]. Arenen bestehen aus Blöcken (engl. chunks) fester Größe (standardmäßig 4MiB). Diese Blöcke werden je nach Bedarf in kleinere Stücke (engl. runs) bestehend aus einer oder mehreren Speicherseiten unterteilt. “Runs” können entweder genutzt werden um große Objekte ($4KiB <= s < 4MiB$) darin abzulegen oder sie werden in noch kleinere Stücke einer festen Größenklasse ([8], [16, 32, ..., 128], [192, 256, ..., 512], [768, 1024, ..., 3840]) unterteilt. Anfragen größer als vier Mebibyte werden grundsätzlich direkt vom System angefordert.

Jeder “chunk” hat einen Kopfvermerk mit Zeiger auf die Arena sowie Metadaten. Diese Metadaten enthalten eine Abbildung, die existierende “runs”, deren Verwendung samt Größeninformation vermerkt, sowie weitere Buchhaltung, welche Speicherstücke bereits angefasst wurden und welche noch nicht. Durch diese Speicherung der Metadaten wird der Mehrspeicherverbauch auf maximal 2% reduziert und es ist ein Zugriff auf die Größeninformation eines Blocks durch Zeigerarithmetik in konstanter Zeit möglich [Eva11].

Arenen verfügen über eine Sperre sowie zwei Rot-Schwarz-Bäume für bereits verwendete und noch frische “runs”, aus denen per Best-Fit bei Bedarf ein neuer “run” gesucht wird. Außerdem verwalten sie pro kleiner Größenklasse eine Sperre sowie einen weiteren Rot-Schwarz-Baum, der die “runs” der entsprechenden Größenklasse nach den Startadressen sortiert. Damit kann immer der Block mit der aktuell niedrigsten Adresse verwendet werden. Abbildung 2.5 zeigt den Aufbau und die Zusammenhänge der verwendeten Datenstrukturen.

Um unnötige Wartezeiten von parallelen Fäden zu vermeiden, wurde darauf geachtet, dass alle Sperren abgegeben werden, bevor ein Systemaufruf getätigt wird [Eva11].

Außerdem wurden durch Facebook ähnlich wie bei *tcmalloc* noch weitere Werkzeuge integriert, um das Speicherverhalten des Allokators und der Anwendung zu beobachten und zu analysieren.

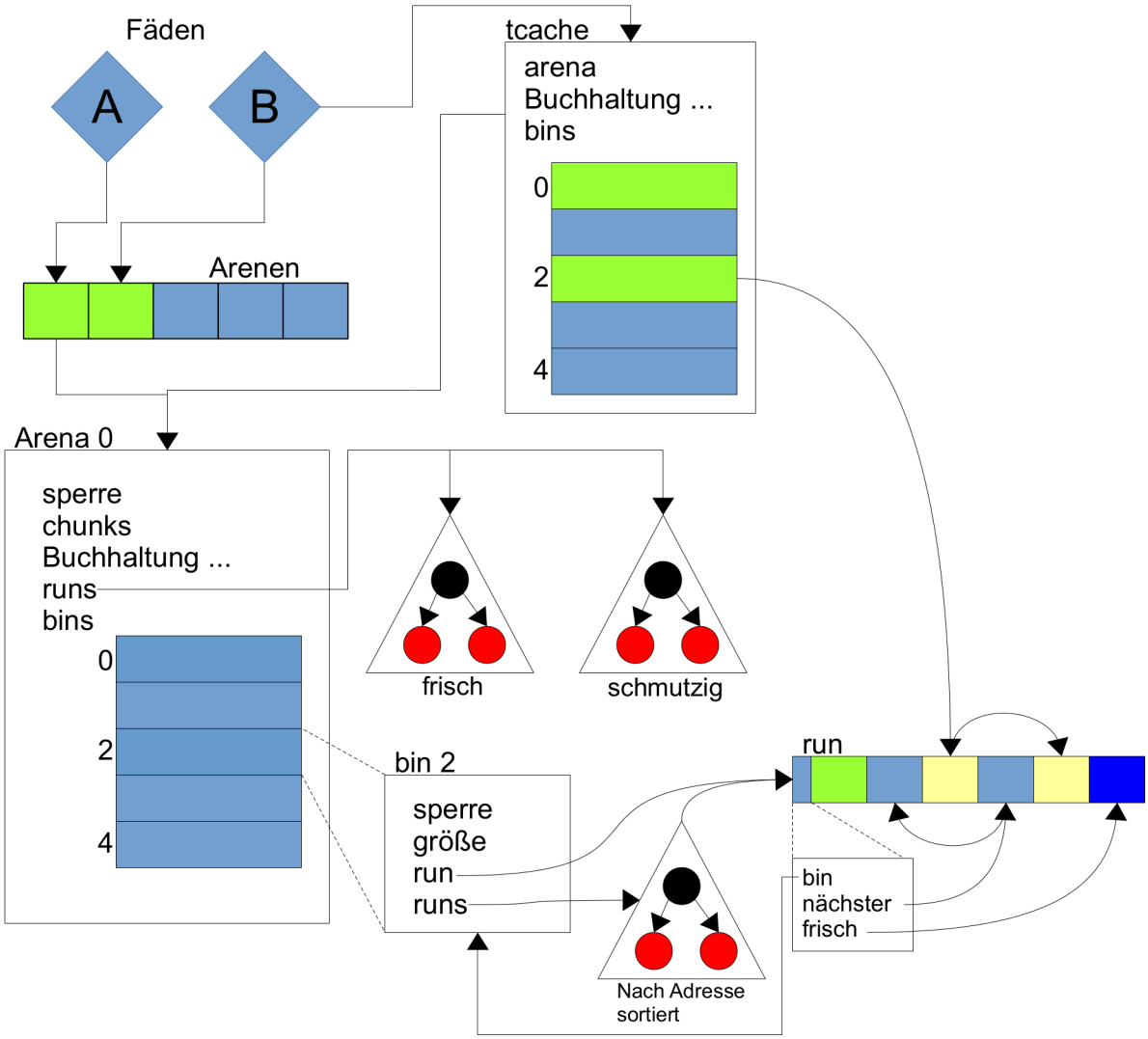


Abbildung 2.5 – Aufbau der internen Datenstrukturen von *jemalloc* (in Anlehnung an [Eva11])

2.3.5 Resümee

Tabelle 2.1 – Zusammenfassung der nichtfunktionalen Eigenschaften der vier Allokatoren

Allokator	Speicherlokalität	Blowup Vermeidung	Teilen und Verschmelzen	Metadaten im Block	Seperation der Größenklassen
Hoard	Prozessor	Ja	Nein	Nein	Physisch
tcmalloc	Faden und Global	Ja	Nein	Nein	Physisch
glibc	Faden und Arenen	Nein	Ja	Ja	Logisch
jemalloc	Faden und Arenen	Ja	Nein	Nein	Physisch

In der Tabelle 2.1, die markante nicht-funktionale Eigenschaften der beschriebenen Allokatoren aufführt, ist das Muster zu erkennen, dass neuere Allokatoren eine andere Herangehensweise an externe Fragmentierung haben, als es lange Stand der Forschung war. Um externe Fragmentierung gänzlich zu umgehen, entstammen unterschiedlich große Blöcke völlig disjunkten Speicherstücken. Damit kann auf das Teilen und Verschmelzen sowie die dafür benötigten Metainformationen in jedem Block verzichtet werden. Allerdings hat dieses Vorgehen den Preis, dass eventuell Speicher für wenig verwendete Größenklassen verschwendet wird, da der für sie reservierte Speicher nicht von Blöcken anderer Klassen belegt werden kann. Ein anderer Nachteil der physischen Trennung der Größenklassen ist, dass gemeinsam genutzte Daten unterschiedlicher Größe nicht wie bei der *glibc* mit hoher Wahrscheinlichkeit in derselben Seite liegen und somit höhere Zugriffskosten aufweisen, welche zum Beispiel in einem NUMA System beträchtlich sein können.

Darüber hinaus ist zu erkennen, dass alle noch aktiv weiterentwickelten Allokatoren durch TLS implementierte Fadenlokalität von Speicher nutzen, um häufig komplett auf Synchronisation verzichten zu können.

Ob und bei welchen Anwendungsszenarien diese Techniken vorteilhaft sind, wird in Kapitel 4 diskutiert. Um die dafür erforderlichen Messwerte zu erhalten, wurde die im folgenden Kapitel vorgestellte Vergleichsumgebung und die in ihr enthaltenen Tests verwendet.

3 ARCHITEKTUR

Die erstellte Vergleichsumgebung *allocbench* soll die in der Einleitung erwähnten Ziele - Flexibilität, einfache Benutz- sowie Erweiterbarkeit und großen Standardumfang - umsetzen. Der folgende Abschnitt soll einen kurzen allgemeinen Überblick über den gewählten Aufbau von *allocbench* geben und die getroffenen Entscheidungen erklären. Danach wird genauer auf den verwendeten Aufbau eines Vergleichstests und jene, die im Standardumfang enthalten sind, eingegangen.

Um das Implementieren einfach und schnell zu gestalten und außerdem leichte nachträgliche Erweiterbarkeit durch Dritte zu gewährleisten, wurde die sehr mächtige und weitverbreitete Skriptsprache *Python* verwendet. Die Umgebung ist darüber hinaus nicht monolithisch, sondern besteht aus mehreren von einander unabhängigen Testdateien und anderen zur Vermessung nützlichen Werkzeugen.

3.1 Aufbau der Vergleichsumgebung

Verwendet wird *allocbench* mit einem Skript für die Kommandozeile namens `bench.py`, mit dem das Verhalten, wie in Algorithmus 3.1 zu sehen ist, festgelegt und einzelne oder mehrere Tests durchgeführt werden können.

Das Skript `bench.py` verwaltet eine Liste von verfügbaren Testobjekten, die mindestens über die Funktion `run()` und je nach gewählten Aufrufoptionen über

- `load()`, um bereits vorhandene Testergebnisse in die Analyse mit aufzunehmen
- `save()`, um die gemessenen Testergebnisse zu speichern
- `summary()`, um die Testergebnisse zu verarbeiten

verfügen müssen. Darüber hinaus kann ein Testobjekt die Funktionen `analyse`, um das Anwendungsverhalten gesondert vom Test aufzuzeichnen, `prepare`, um Vorbereitungen vor der Ausführung zu treffen, sowie `cleanup`, um die mit `prepare` getroffenen Vorbereitungen wieder rückgängig zu machen, implementieren. Da jedes beliebige Pythonobjekt, das die erforderten Funktionen besitzt, verwendet werden kann, ist die Umgebung beliebig erweiterbar. Dabei müssen nicht die in *allocbench* verwendeten Techniken, beispielsweise *matplotlib* zur Erzeugung von Graphen, verwendet werden. Die einfache und flexible Schnittstelle ermöglicht es, jeden Test, der in Python programmiert werden kann, in die Analyse zu integrieren. Damit das Einfügen eines neuen Tests nicht zwingend gleichbedeutend ist mit dem Schreiben eines neuen *Python* Programms und, um die Implementierung der bereits enthaltenen Tests zu vereinfachen, bietet *allocbench* die Klasse `Benchmark` an. Diese Klasse implementiert alle oben genannten Methoden bis auf `summary` in einer möglichst generischen und einfach zu benutzenden Form. Um die bereits implementierten Methoden zu benutzen, muss

Require: Liste von durchzuführenden Vergleichstests
1: Erzeuge Ausgabeverzeichnis
2: **for all** Test ∈ Liste von Tests **do**
3: **if** Ladeoption (`-l`) gesetzt **then**
4: Lade alte Ergebnisse
5: **end if**
6: **if** Durchläufe (`-r n`) > 0 **or** Analysedurchlauf (`-a`) **then**
7: Bereite Test vor
8: **end if**
9: Führe Test durch
10: **if** Test nicht erfolgreich **then**
11: **continue**
12: **end if**
13: **if** Speicheroption (`-s`)gesetzt **then**
14: Speicher Ergebnisse
15: **end if**
16: **if not** Nicht-Zusammenfassungsoption (`-ns`) gesetzt **then**
17: Verarbeite Testergebnisse und fasse Test in Ergebnisordner (`-sd Pf/ad/`) zusammen
18: **end if**
19: **if** Aufräumen erforderlich **then**
20: Räume hinter Test auf
21: **end if**
22: **end for**

Algorithmus 3.1 – Grober Ablauf und Aufrufoptionen von *allocbench*

von `Benchmark` geerbt werden und mindestens die drei Attribute, `name`, `cmd` sowie `args` definiert werden. Das Attribut `name` wird von `bench.py` benutzt, um das richtige Testobjekt zu finden. Die beiden Anderen definieren den durch die in `Benchmark` enthaltenen Methoden durchgeführten Test.

Abbildung 3.1 zeigt eine mögliche Konfiguration mit drei unterschiedlichen Testklassen und deren Objekten, die von jedem Test benötigten, sowie die von `Benchmark` bereitgestellten Methoden. In der gezeigten Konfiguration erben zwei von drei Tests von `Benchmark` und “Test1” implementiert alle benötigten Methoden eigenständig.

Um die in Abschnitt 3.1.2.1 beschriebenen von `Benchmark` bereitgestellten Methoden und Abstraktionen besser verstehen zu können, betrachten wir zuerst wie ein allgemeiner Vergleichstest in *allocbench* modelliert wird.

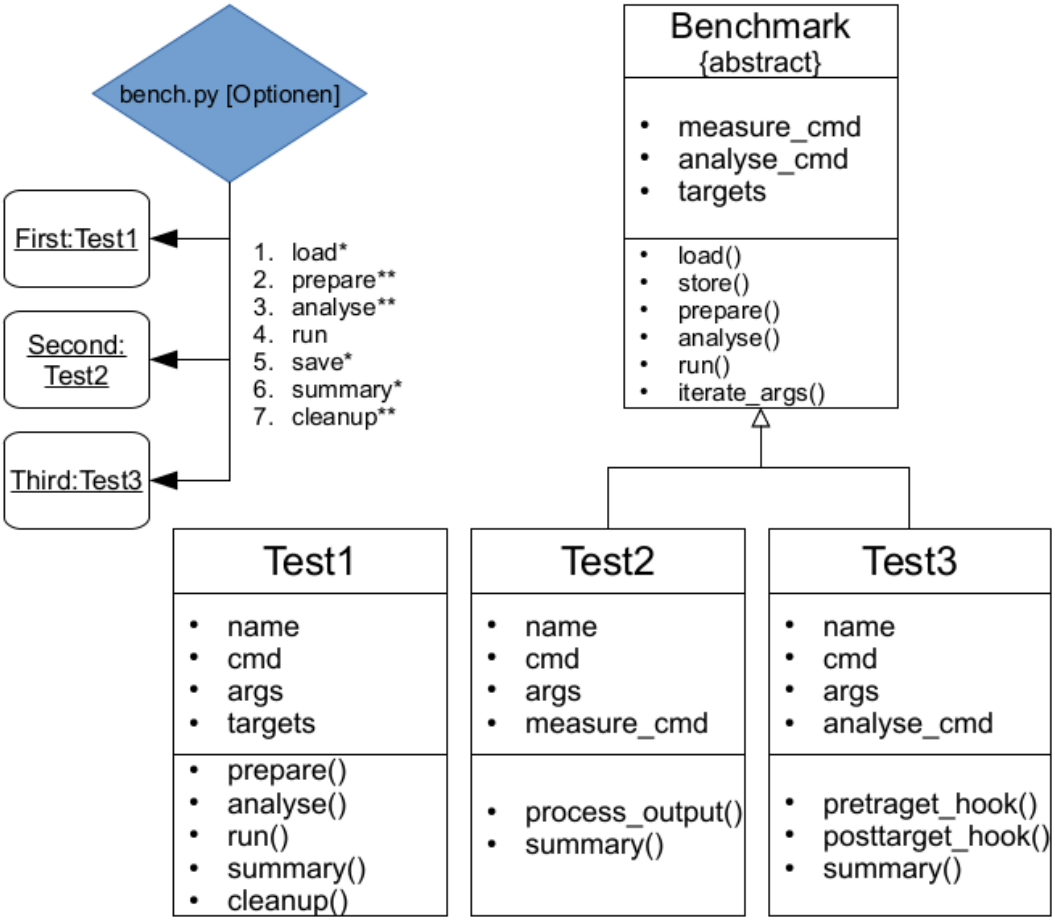
3.1.1 Modellierung eines Vergleichstests

Ein Vergleichstest ist definiert durch eine Schablone, aus der eine per `exec()` ausführbare Zeichenkette mit den entsprechenden Argumenten erzeugt werden kann. Das erzeugen des eigentlich Befehls geschieht mit einer Abbildung von Argumentnamen auf eine Liste von möglichen Ausprägungen. Ein Test, der beispielsweise die Zeit misst, die benötigt wird Inhalte unterschiedlicher Systemverzeichnisse aufzulisten, kann wie folgt definiert werden:

```
cmd = "ls {dir}"
args = {"dir" : ["/bin/", "/usr/bin"]}
```

Zudem wird eine weitere Abbildung von zu testenden Allokatoren auf Informationen, wie diese zu benutzen sind, benötigt. Für die vier in dieser Arbeit untersuchten Allokatoren liegt eine

Abbildung 3.1 – Beispielskonfiguration von *allocbench* mit drei verschiedenen Tests.
** Muss bei entsprechender Option vorhanden sein.*
*** Wird auch bei entsprechender Option nur ausgeführt falls vorhanden.*



solche Abbildung in der Datei `common_targets.py` vor, welche standardmäßig von der Klasse `Benchmark` benutzt wird, falls keine eigene definiert wurde. Diese Abbildung erwartet Zeiger oder die tatsächlichen dynamischen Bibliotheken der Allokatoren im Unterverzeichnis `targets/`.

Die `run` Methode der `Benchmark` Klasse führt mit dieser Definition den in Algorithmus 3.2 dargestellten Algorithmus aus. Innerhalb jedes Testlaufs wird pro Allokator die Einschubmethode `pretarget_hook` mit der Abbildung des aktuell untersuchten Allokators sowie der Nummer des Testdurchlaufs aufgerufen, falls das Testobjekt diese definiert hat. Diese Einschubmethode wird benötigt, wenn vor den einzelnen Durchläufen einmalig für jeden Allokator Arbeit erledigt werden muss. Der in `allocbench` enthaltene Test `mysql` beispielsweise startet mit ihr einen *MySQL* Server für den aktuellen Allokator, an den während des Testlaufs unterschiedliche Anfragen gestellt und vermessen werden.

Der Kern eines jeden Testlaufs ist eine Schleife über alle möglichen Permutationen der Argumente. Aus der aktuellen Permutation werden mit Hilfe der Abbildung an den passenden Stellen die

Require: Liste von zu testenden Allokatoren
Require: Schablone des auszuführenden Befehls
Require: Liste von Argumenten mit allen ihren möglichen Ausprägungen
Require: Anzahl an Durchläufen
1: Erzeuge Ausgabeverzeichnis
2: **for** Anzahl an Durchläufen **do**
3: **for all** Allokator ∈ Liste von Allokatoren **do**
4: Füge Bibliothek, die `/proc/self/status` bei Prozesstermination sichert und Allokatorbibliothek, falls benötigt zur `LD_PRELOAD` Umgebungsvariable hinzu
5: **if** Präausführungs-Einschubmethode vorhanden **then**
6: Führe Präausführungs-Einschubmethode aus
7: **end if**
8: **for all** Permutation ∈ alle Permutationen der Argumente **do**
9: Baue ausführbaren Befehl aus Schablone mit aktueller Permutation und Allokator spezifischen Teilen
10: **if** Messbefehl vorhanden **then**
11: Füge Messbefehl zu tatsächlichen Befehl hinzu
12: **end if**
13: Führe tatsächlichen Befehl aus
14: **if not** Befehl erfolgreich war **then**
15: Gib verfügbare Informationen zum Fehler aus
16: **return** Fehlercode
17: **end if**
18: Lese `VmHWM` aus gesicherter `status` Datei
19: Lösche gesicherte Datei
20: **if** Ausgabe-Einschubmethode vorhanden **then**
21: Führe Ausgabe-Einschubmethode aus
22: **end if**
23: Zerteile Messausgabe
24: Speichere alle Messergebnisse in interne Ergebnisdatenstruktur
25: **end for**
26: **if** Postausführungs-Einschubmethode vorhanden **then**
27: Führe Postausführungs-Einschubmethode aus
28: **end if**
29: **end for**
30: **end for**

Algorithmus 3.2 – Ablauf eines Vergleichstests

konkreten Werte in die Schablone eingesetzt. Der so enthaltene ausführbare Befehl wird noch mit dem `measure_cmd` Präfix versehen, das es ermöglicht ein externes Werkzeug, wie `perf` oder `time` zum Messen zu verwenden. Außerdem wird vor der Ausführung des Programms noch die Umgebungsvariable `LD_PRELOAD` entsprechend vorbereitet, dass die dynamische Bibliothek des Allokators statt der Standardbibliothek vom nun gestarteten Kindprozess verwendet wird.

Nachdem der mit dem Befehl gestartete Kindprozess terminiert ist, werden seine Ausgaben, das aktuelle Ergebnisobjekt, der Name des Allokators sowie die aktuelle Permutation an die Einschubmethode `process_output`, falls diese existiert, übergeben. Diese Einschubmethode wird immer dann benötigt wenn der ausgeführte Befehl selbst die gewünschten Messwerte liefert. Um keinen Standard für die Ausgaben der Testbefehle zu erzwingen muss diese Einschubmethode definiert werden, um die Messergebnisse aus den spezifischen Ausgaben des Kindprozesses zu extrahieren und im Ergebnisobjekt zu speichern.

Wenn alle Permutationen abgearbeitet wurden, wird die letzte Einschubmethode `posttarget_hook` ausgeführt. Sie ist dafür gedacht nach dem Test eines Allokators aufzuräumen und die Umgebung bereit für den nächsten zu machen. Im bereits erwähnten Test *mysql* terminiert und vermisst sie den gestarteten *MySQL* Server.

Nun da der Grundablauf und die Modellierung eines Tests bekannt sind, wird im Folgenden nun genauer auf die von *allocbench* bereit gestellte Funktionalität eingegangen und gezeigt wie, mit wenig Programmieraufwand ein neuer Test hinzugefügt werden kann.

3.1.2 Bereitgestellte Funktionalität

Nahezu alle nicht testspezifische Funktionalität ist in der *Python* Klasse `Benchmark` enthalten. Sie ermöglicht es, eine neue Testklasse, ähnlich wie sie Listing 3.1 zeigt, ohne viel Programmieraufwand zu erstellen. Der Ansatz, eine Klasse zu verwednen, hat den Vorteil, dass er nicht die Freiheit des Programmierers einschränkt, da jede Funktion überschrieben und somit beliebig angepasst werden kann. Die testspezifischen Aspekte, die nicht allgemein gelöst werden können ohne einen Teststandard zu erzwingen, wie relevante Messdaten aus der Ausgabe des Testprogramms zu extrahieren, muss der Testersteller selbst in einer der passenden Einschubmethoden implementieren.

Über den grundsätzlichen Aufbau und die Durchführung eines Test hinaus bietet *allocbench* außerdem Funktionalität, das Verhalten eines Tests aufzuzeichnen. Zu den erfassten Eigenschaften gehört die Größe jeder Allokation sowie ein zeitlicher Überblick, wie viel Speicher zu jedem Zeitpunkt angefordert wurde. Diese Informationen können nützlich sein, um das Speicherverhalten der eigenen Anwendung besser zu verstehen und um so eine Hilfestellung für die Interpretation der eigentlichen Testergebnisse zu liefern.

Um allerdings ein eigenes Programm untersuchen zu können, muss es zuerst in eine für *allocbench* nutzbare *Python* Klasse verpackt werden. Das dazu erforderliche Vorgehen erklärt der folgende Abschnitt.

3.1.2.1 Erstellen eines Vergleichstests

Um einen neuen Test in die Umgebung zu integrieren, muss eine neue Klasse definiert werden, dazu bietet es sich an, eine bereits Bestehende zum Beispiel die in Listing 3.1 Gezeigte in eine neue Datei zu kopieren und den Namen sowie die den Test beschreibenden Attribute anzupassen. Anschließend muss ein globales Objekt dieser Klasse erzeugt und in die Liste aller Tests in `bench.py` eingehängt werden.

Beim Erstellen eines neuen Tests muss darauf geachtet werden, dass der verwendete Befehl deterministisch ist, damit die resultierenden Ergebnisse interpretiert werden können. Das heißt,

```
import multiprocessing

from benchmark import Benchmark

class Benchmark_Loop( Benchmark ):
    def __init__(self):
        self.name = "loop"
        self.descrition = """This benchmark makes n allocations in t concurrent threads.
                             How allocations are freed can be changed with the benchmark
                             version""",

        self.cmd = "build/bench_loop{binary_suffix} {nthreads} 1000000 {maxsize}"

        self.args = {
                        "maxsize" : [2 ** x for x in range(6, 16)],
                        "nthreads" : range(1, multiprocessing.cpu_count() * 2 + 1)
                    }

        self.requirements = ["build/bench_loop"]
        super().__init__()

    def summary(self, sumdir):
        # Speed
        self.plot_fixed_arg("perm.nthreads / (float({task-clock})/1000)",
                            ylabel = '"MOPS/cpu-second"',
                            title = '"Loop: " + arg + " " + str(arg_value)',
                            filepostfix="time",
                            sumdir=sumdir)

        # Memusage
        self.plot_fixed_arg("int({VmHWM})",
                            ylabel='"VmHWM in kB"',
                            title= '"Loop Memusage: " + arg + " " + str(arg_value)',
                            filepostfix="memusage",
                            sumdir=sumdir)

        self.plot_fixed_arg("({L1-dcache-load-misses}/{L1-dcache-loads})*100",
                             ylabel='"L1 misses in %"',
                             title= '"Loop l1 cache misses: " + arg + " " + str(arg_value)',
                             filepostfix="l1misses",
                             sumdir=sumdir)

        # Speed Matrix
        self.write_tex_table("perm.nthreads / (float({task-clock})/1000)",
                             filepostfix="memusage.matrix",
                             sumdir=sumdir)

loop = Benchmark_Loop()
```

Listing 3.1 – Vollständige *Python* Datei eines in *allocbench* enthaltenen Vergleichstests (Loop Test, beschrieben in Abschnitt 3.2.1.2).

dass bei mehrmaligen Ausführen des Testbefehls die gleichen Allokatoraufrufe stattfinden und, falls die gemessene Zeit interpretiert werden soll, auch die von der Anwendung erledigte Arbeit nicht variiert. Falls diese Anforderung nicht erfüllt werden kann, gibt es die Möglichkeit, Werkzeuge von DJ Delorie[2] zu verwenden. Sie ermöglichen es, die Verwendung der Speicherverwaltungs API aufzuzeichnen, diese Aufzeichnung in eine Anweisungsdatei umzuwandeln und von einem speziellen Programm erneut ausführen zu lassen. Dadurch wird garantiert, dass das aufgezeichnete und wieder abgespielte Verhalten bei jedem Durchlauf identisch ist. Die zeitliche Komponete des Aufrufprofil wird dadurch zwar eliminiert, aber das Verhalten des Allokators kann zumindest beim Ausführen des Anwendungsprofils deterministisch und von anderen Effekten isoliert untersucht werden.

Falls ein eigenes Programm verwendet werden soll, können die Quelldateien in dem Ordner `benchmarks/` gespeichert und mit Hilfe des vorhandenen Makefiles übersetzt werden. Übersetzte Programme werden, wie auch andere enthaltene Hilfsbibliotheken, im Ordner `build/` erzeugt.

Einen Allokator aus einer C-Standardbibliothek zu testen, ist oft mit zusätzlichem Aufwand verbunden, da der Allokator beliebig mit seiner C-Bibliothek verzahnt sein kann. Es werden zwei Möglichkeiten unterstützt, eine andere C-Standardbibliothek zu verwenden als die des Systems. Die Erste wird für selbsterzeugte Testprogramme benutzt. Dabei wird von dem zu testenden Programm eine Kopie erzeugt und mit dem Programm *patchelf* werden der verwendete Lader sowie der hinterlegte Bibliothekspfad in der Kopie geändert. Die zweite Möglichkeit wird zum Beispiel im `mysql` Test genutzt und besteht darin, pro zu testenden Allokator den auszuführenden Befehl beliebig am Anfang zu erweitern. So wird, um eine andere Variante der *glibc* zu verwenden, der in ihr enthaltene Lader aufgerufen, welcher den eigentlichen Befehl ausführt. Bei beiden Möglichkeiten ist zu beachten, dass sich für den Testlauf nicht nur der Allokator, sondern womöglich auch andere benutzte Funktionen der Standardbibliothek ändern. Deswegen sind die erzeugten Ergebnisse immer mit Blick auf mögliche Auswirkung dieser Änderungen zu betrachten.

Nachdem eine neue Klasse erzeugt wurde und der zu testende Befehl deterministisch für alle Allokatoren ablaufen kann, muss zusätzlich, falls der Standardablauf nicht passt oder ausreicht, die nötige fehlende Funktionalität implementiert werden.

3.1.2.2 Analyse des Anwendungsverhaltens

Im Gegensatz zu vielen synthetischen Tests ist das Anfragenprofil einer Anwendung aussagekräftig, da sie geschrieben ist, ein tatsächliches Problem zu lösen. Das bedeutet auch, dass ihre Speicheranforderungen nicht willkürlich sind, sondern häufig Muster und charakteristische Phasen aufweisen [Wil+95]. Diese Muster kann ein Allokator für seine Strategie nutzen. *allocbench* bietet mit der Aufrufoption `-a`, die die Funktion `analyse` des Testobjekts aufruft, die Möglichkeit das Speicherverhalten des Tests mitzuschreiben.

Die Implementierung in der `Benchmark` Klasse benutzt zwei verschiedene Ansätze. Der Standardweg ist es, ein kleines Skript der *glibc* namens *memusage* zu benutzen. Dieses Skript führt den übergebenen Befehl aus und lädt zusätzlich per `LD_PRELOAD` die Bibliothek *libmemusage*, die das Allokationsverhalten aufzeichnet. Aus der Aufzeichnung wird anschließend mit dem Programm *memusagestat* eine Graphik des angeforderten Speichers über die Laufzeit des Programms sowie ein Histogramm der angeforderten Speichergrößen erzeugt. Die zweite Möglichkeit wird verwendet, wenn *memusage* nicht installiert ist oder explizit nicht benutzt werden soll. Das Speicherprofil der Anwendung wird mit Hilfe der dynamischen Bibliothek *chattmyalloc.so* in eine Datei geschrieben. Diese Bibliothek definiert eigene Versionen der Speicherverwaltungs API und dokumentiert für jeden Aufruf die übergebenen Parameter sowie den Rückgabewert. Aus dieser Aufzeichnung wird danach von dem *Python* Skript *chattyparser.py* ebenfalls ein Histogramm, ein Zeitverlauf des gesamten ange-

[2] Der branch dj/malloc des *git* repository der *glibc* enthält die erwähnten Werkzeuge im Unterordner malloc.

forderten Speichers sowie der fünf am häufigsten angeforderten Größen erzeugt. Da das Generieren dieses Profils sehr naiv implementiert ist und daher viel Hauptspeicher benötigt, ist es für große Aufzeichnungen nicht geeignet. Bei der Verarbeitung werden pro Funktionsaufruf sechs Integer zum Verlauf hinzugefügt. Für eine Aufzeichnung von 190 Millionen Aufrufen (3.4 GB Aufzeichnung) werden über neun GB nur für die Datenstruktur des Profils benötigt.

Mit den über die Anwendung erhaltenen Informationen lassen sich die beobachteten Auswirkungen der Allokatoren leichter verstehen sowie den schließlich Ausgewählten besser auf die Anforderungen des verwendeten Programms einstellen.

3.1.2.3 Erhebung der Messdaten

Die zwei für jeden Test nützlichen Kennzahlen sind die benötigte Ausführungszeit und der maximale Speicherverbrauch, den der Linuxkern in der Datei `/proc/self/status/` mit dem Wert `VmHWM` zur Verfügung stellt. Diese Datei existiert allerdings nur während der Laufzeit des Programms. Um nicht wärend der Programmausführung ständig die Datei lesen zu müssen, enthält *allocbench* die Bibliothek `print_status_on_exit.so`. Sie sorgt in ihrem Konstruktor per `atexit` dafür, dass nach Beendigung der `main` Funktion die Datei `/proc/self/status` zur späteren Verwendung kopiert wird. Durch dieses Vorgehen wird kein externes Polling benötigt, was zur Verzerrung des Tests führen könnte.

Die Laufzeit und viele andere nützliche Kennzahlen, wie die Anzahl an Zwischenspeicher- oder Seitenfehlern wird standardmäßig mit dem Programm *perf* aufgezeichnet. Dabei besteht allerdings die Gefahr, dass durch die Arbeit von *perf* im Hintergrund die Ergebnisse verzerrt werden. Beim Vergleich der Messungen, die durch das weitaus simplere Programm *time* erzeugt, und denen, die mit *perf* aufgezeichnet wurden, war kein Unterschied und vorallem keine Diskriminierung eines einzelnen Allokators zu erkennen. Dennoch sollte kein externer Befehl benutzt werden, wenn der Testbefehl bereits selbst seine Ergebnisse liefert, um die Gefahr einer Beeinflussung generell auszuschließen. Der verwendete Messbefehl kann verändert oder entfernt werden, in dem das Attribut `measure_cmd` der Testklasse gesetzt wird.

Um aus der Ausgabe des Testprogramms Messwerte zu extrahieren, wird die Einschubmethode `process_output` angeboten. Sie bekommt als Eingabeparameter beide Standardausgaben des Testbefehls sowie das aktuelle Ergebnisobjekt und wird normalerweise benutzt, um mit regular expressions die Daten aus der Ausgabe zu extrahieren und im übergebenen Ergebnisobjekt zu speichern.

3.1.2.4 Verarbeitung der Messdaten

Die von der Klasse `Benchmark` genutzte Ergebnisdatenstruktur speichert nicht nur die einzelnen Messwerte nach Allokatoren sortiert, sondern auch den Befehl, die Argumente sowie die verwendete Liste der Allokatoren, um die Auswertung unabhängig von der Durchführung des Tests zu machen. Diese Separation ist nützlich, wenn das System, auf dem die Tests durchgeführt wurden, nicht die Mittel hat, auch die Auswertung durchzuführen (kein X-Server, keine passenden Bibliotheken).

Um aus den gemessenen Ergebnissen einfacher interpretierbare Graphen zu generieren, werden die Funktionen `plot_fixed_args` und `plot_single_arg` angeboten. Sie erzeugen mit der Bibliothek *matplotlib* Graphen für ein einzelnes Argument oder einen Graphen pro Ausprägung eines oder mehrer Argumente. Allerdings kann auch jede andere Form der Datenverarbeitung benutzt werden, da die Funktion `summary` nicht vorimplementiert ist und auch völlig frei von der in `Benchmark` angebotenen Funktionalität implementiert werden kann.

Einen guten Anhaltspunkt bei der Integration eines neuen Tests bieten die fünf bereits Enthaltenen: `mysql`, `loop`, `falsesharing` `dj_trace`, `larson`.

3.2 Ausgewählte Vergleichstests

Um ein ausgewogenes Bild der Allokatoren zu erhalten, wurde bei der Auswahl der enthaltenen Vergleichstests darauf geachtet, nicht nur synthetische probabilistische Tests zu verwenden. Zufällige Tests bilden nicht das Verhalten tatsächlicher Programme ab und bergen daher die Gefahr, einzelne Allokatoren besser aussehen zu lassen, obwohl die erzeugte Last nicht in der Realität vorzufinden ist. Deshalb werden nur synthetische Tests benutzt, die sehr simpel sind und explizit eine spezifische Eigenschaft des Allokators testen. Um einen realitätsnäheren Blick auf die Allokatoren zu erlangen, sind zwei Tests enthalten, die entweder reale Lastmuster wiederholen oder versuchen, derartige zu simulieren.

Die vom Allokator verursachte Fragmentierung kann leider nicht befriedigend untersucht werden, da dafür Unterstützung der Allokatoren nötig wäre. Als Maß für die entstehende Fragmentierung kann deshalb nur der Vergleich zwischen dem maximal und dem idealerweise benötigten Speicher betrachtet werden.

3.2.1 Synthetische Tests

Die enthaltenen synthetischen Tests prüfen die zwei unterschiedlichen Arten der vom Allokator beeinflussbaren irrigen Mitbenutzung sowie die Skalierbar- beziehungsweise Geschwindigkeit der Allokatoren.

3.2.1.1 Tests der irrigen Mitbenutzung

Ob ein Allokator aktiv irrige Mitbenutzung verursacht, indem er Blöcke innerhalb derselben Zwischenspeicherzeile an unterschiedliche Fäden heraus gibt, wird von dem Test *cache-thrash* überprüft. Dieser Test alloziert in einer Schleife pro Faden ein Objekt kleiner als eine Zwischenspeicherzeile, führt darauf eine große Anzahl an Schreib- und Leseoperationen durch und giebt es schließlich wieder frei. Die Zeit, die für die wenigen Allokationen benötigt wird, verschwindet im Vergleich zu der Zeit, die die Schreib- und Leseoperationen benötigen. Das bedeutet, dass die Laufzeit des Tests davon abhängt, wie schnell diese Operationspaare durchgeführt werden können. Die hängt wiederum davon ab, wie viele Zwischenspeicherzeilen zwischen unterschiedlichen CPUs wandern beziehungsweise wie viel Zeit durch das Zwischenspeicherprotokoll im Hintergrund benötigt wird. Wenn keine Zwischenspeicherzeilen geteilt werden, müssten mehr Prozessoren bei gleich bleibender Anzahl an Schreib- und Leseoperationen einen linearen Speedup erzeugen. Je größer die irrige Mitbenutzung ist, desto geringer der Speedup [Ber+00].

Ein zweiter Test, namens *cache-scratch* prüft, ob der Allokator passiv oder aktiv irrige Mitbenutzung verursacht. Beim Start wird zentral pro Faden ein Objekt alloziert und eines an jeden Faden übergeben, welcher das Objekt unverzüglich freigibt und danach wie *cache-thrash* verfährt. Das sofortige Freigeben des ersten Objekts prüft, ob der Allokator ermöglicht, dass der freigebende Faden sein Objekt, das von einem anderen Faden stammt, bei der nächsten Allokation wiedererhält und somit passive irrige Mitbenutzung ermöglicht [Ber+00].

Beide Tests allozieren hundertmal ein Objekt der Größe acht Byte und führen auf jedem eine Million Schreib- und Leseoperationen geteilt durch die Anzahl an Fäden durch, bevor sie es wieder freigeben.

3.2.1.2 Simpler Geschwindigkeitstest

Der *loop* Test macht pro Faden eine Million mal eine Allokation pseudozufälliger Größe, die unmittelbar danach wieder frei geben wird. Auf ein Lesen oder Beschreiben der Allokation würde versichtet um nicht die durch die berets vorgestellten Tests vermesse irrige itbenutzung erneut zu Messen. Dieser Test ist durch den in *ptmalloc2* enthaltenen und für die Evaluation von *tcmalloc* verwendeten Test `t-test1` [GM07], sowie durch *threadtest* aus der Veröffentlichung von Hoard [Ber+00] inspiriert. Um Unterschied zu `t-test1` sind viel weniger Zufallsentscheidungen enthalten, die Lebensdauer einer Allokation ist immer gleich und es werden nur die Funktionen `malloc` und `free` verwendet. Dieser simplifizierte Test überprüft ausschließlich die Skalierbarkeit eines Allokators für eine zunehmende Anzahl an parallelen Fäden. Da die angeforderte Allokation direkt wieder freigegeben wird, hat die Zwischenspeicherstrategie der Allokatoren den größten Einfluss auf die gemessene Laufzeit. Bei diesem Test wird die Gesamtlaufzeit und der benötigte Speicherverbrauch aufgezeichnet und untersucht.

3.2.2 Reale Tests

Um ein gesamtheitliches Bild der untersuchten Allokatoren und all ihrer nicht-funktionalen Eigenschaften zu erhalten, ist ein Test enthalten, der versucht die reale Last einer Serveranwendung zu erzeugen. Außerdem ist eine Sammlung von Lastprofilen enthalten, die DJ Delorie aufgezeichnet und benutzt hat, um seine Änderungen an der *glibc* zu vermessen.

3.2.2.1 MySQL/sysbench Test

Dieser Test ist inspiriert von einen Artikel von Alexey Stroganov einem Datenbankentwickler des Unternehmens Percona, welches verschiedene Datenbanksysteme entwickelt und Support für diese verkauft [Str12]. Dieser Test benutzt den externen System- und Datenbankvergleichstest *sysbench*, der verschiedene Testlasten für Datenbankserver erzeugen kann. Die generierte Datenbank besteht aus fünf Tabellen mit je eine Million Einträgen. *sysbench* startet mehrere Fäden, die parallel für sechzig Sekunden unterschiedliche `SELECT` Anfragen an den Server stellen. Gemessen wird dabei die Anzahl an durchgeführten Transaktionen sowie der vom *MySQL* Server benötigte Speicher. Es wurde ein rein lesender Modus gewählt, um möglichst wenig Einfluss des Ein- und Ausgabesubsystems zumessen. Pro Allokator wird ein *MySQL* Server gestartet und mehrere Durchläufe des *sysbench* Tests `oltp_read_only` mit steigender Anzahl an Fäden durchgeführt. Nach jedem Durchlauf wird der maximale Speicherverbrauch des Servers gemessen. Das per `analyse` für 10 Fäden erhaltene Histogram zeigt das 50% aller Allokationen kleiner als 1024 Byte und 82% kleiner als eine Seitengröße sind. Größere Allokationen fallen nahezu ausschließlich in die Größenintervalle: 5136-5151 mit 2.5%, 65536-65551 mit 5% und 114400-114415, 258160-258175, 524224-524239 mit jeweils 2.5%. Von den kleinen Allokationen sind ca. 2.5% kleiner als eine Zwischenspeichezeile.

3.2.2.2 DJ Delorie's Tests

Dieser Test besteht aus einer Sammlung von neun unterschiedlichen Lastprofilen, die mit den in Abschnitt 3.1.2.1 erwähnten Werkzeugen aus Aufzeichnungen sowohl synthetischer als auch realer Programme erzeugt wurden. Diese Lastprofile werden von einem Simulator, der die Lastdatei einliest und die Allokationen nochmal in entsprechenden Fäden durchführt, erneut durchgeführt. Die Fäden synchronisieren sich dabei nur an Stellen, an denen ein Block zwischen zwei Fäden der Originalanwendung gewechselt ist, das dadurch festgestellt werden kann, dass ein anderer Faden einen Block zurückgibt als der, dem der Block ausgehändigt wurde. Es wird keine weitere

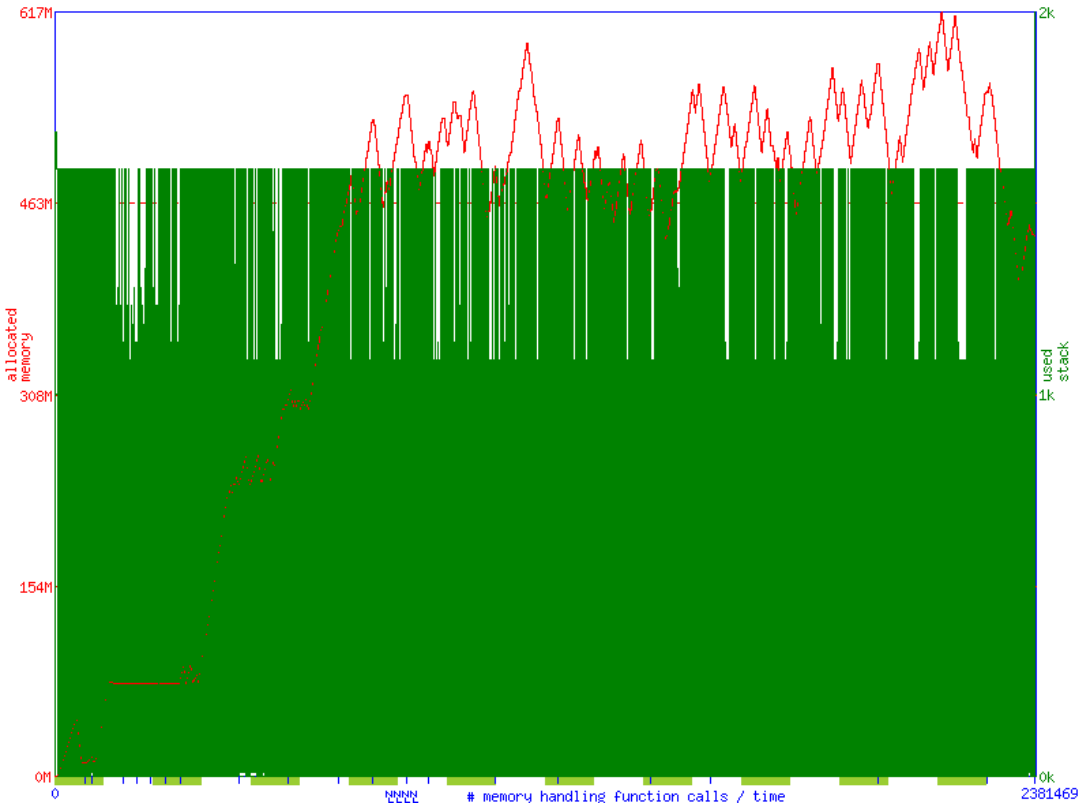
Arbeit als die Allokationen und Deallokationen durchgeführt. Während der Simulation bestimmt der Simulator die durchschnittlich benötigte Zeit der verschiedenen API Funktionen sowie den idealen Speicherverbrauch, den ein perfekter Allokator aufweisen würde. DJ Delorie stellt diese Lastprofile, die er auch benutzt hat, um die neuen fadenlokalen Zwischenspeicher der *glibc* zu vermessen[Del17], auf seiner Webseite[3] zur Verfügung. Die enthaltenen Lastprofile stammen von Desktopanwendungen wie *Open Office Calc*, der virtuellen Maschine *qemu*, verschiedensten Serveranwendungen, darunter beispielsweise der Lightweight Directory Access Protocol (LDAP) Server *389* von Red Hat, sowie synthetischen Vergleichstests, die Worst-Case-Verhalten für die *glibc* erzeugen. Das breite Spektrum an Lastprofilen ermöglicht ein gutes Gesamtbild über die Stärken und Schwächen der untersuchten Allokatoren. Im Folgenden werden kurz die einzelnen Tests und ihre charakteristischen Eigenschaften vorgestellt.

- **mt_test_one_alloc**: Dieser synthetische Test führt in einem einzelnen Faden immer abwechselnd eine kleine (257 — 514 Byte) und eine große (118784 — 126976 Byte) Allokation durch. Von diesen Allokationen werden 500 Große sowie 70000 Kleine vorgehalten.
- **dj**: In diesem Test werden in 200 parallelen Fäden je 20000 achtzig Byte Objekte allokiert und direkt wieder freigegeben. Sehr vereinzelt werden Allokationen an andere Fäden weitergereicht und dort zurückgegeben.
- **dj2**: Für diesen letzten synthethischen Test spielt der Zufall eine große Rolle. Das erzeugte Verhalten ähnelt stark dem für *tcmalloc* verwendeten `t-test1`. Es werden zufällig bestimmte API Funktionen sowie weitverteilte zufällige Größen verwendet. Von diesen zufälligen Allokationen werden zufällig Bestimmte in einem anderen Faden zurückgegeben.
- **qemo-virtio, qemu-win7**: Diese Lastprofile stammen von der Virtualisierungssoftware *qemu* und weisen die in Abbildung 3.2 zu sehenden Anfragenprofile auf. Es gibt eine Initialisierungsphase gefolgt von einem Sägezahnprofil während der Durchführung der Virtualisierung. Der Großteil der Allokationen (91% bei `virtio`) ist unter 512 Byte groß und es gibt nur vereinzelt seitengroße und Allokationen über 60K.
- **389-ds-2, proprietary-1, proprietary-2**: Das Speicherprofil dieser Serveranwendungen weisen ein ganz ähnliches Muster auf allerdings ohne die markante anfängliche Initialisierungsphase. Ebenfalls ist der Großteil der Allokationen kleiner als 256 Byte. Nur `389-ds-2` fordert zusätzlich eine große Menge an seitengroßen Blöcken an.
- **oocalc**: Dieser Test, der das Anfrageverhalten eines Tabellenkalkulationsprogramms simuliert, hat eine Initialisierungsphase, ein Ladephase, eine Nutzungs- sowie eine Abbauphase. Die Nutzungsphase weist anders als die anderen Profile kein Sägezahnmuster auf sondern wächst stetig. Wieder liegen die meisten Allokationen unter 256 Byte mit einer markant höheren Zahl an 464-479 Byte großen Objekten.

Im folgenden Kapitel werden die Ergebnisse der soeben beschriebenen Tests und ihre von *allocbench* erzeugte graphische Representation präsentiert.

[3] http://www.delorie.com/malloc/

Abbildung 3.2 – Exemplarisches Speicherprofil des Virtualisierungstests `qemu-win7`. Der rote Graph zeigt die angeforderte Speichermenge.



4 EVALUATION UND DISKUSSION

Das folgende Kapitel soll einen Eindruck über die in *allocbench* enthaltenen Tests und eine Interpretation ihrer Ergebnisse verschaffen. Außerdem werden dafür die in Kapitel 2 vorgestellten Allokatoren in verschiedenen Dimensionen verglichen, beobachtete Effekte der Allokatoren soweit möglich erklärt und die Aussagekraft der Tests diskutiert.

4.1 Rahmenbedingungen

Für die tatsächliche Durchführung der Tests wurde ein *Ubuntu 16.04* System mit einem *Intel(R) Xeon(R) CPU E5-4640 v2* Prozessor verwendet. Das System besitzt vier NUMA Domänen à zehn Prozessoren, 32GB Hauptspeicher und 20 MB L3-Zwischenspeicher, also insgesamt 40 Prozessoren und 128GB Arbeitsspeicher. Jeder Prozessor unterstützt zwei Hardwarefäden und verfügt über 256KB L2- sowie je 32KB Daten- und Instruktions-L1-Zwischenspeicher. Die maximale Taktrate der Prozessoren liegt bei 2.2 GHz. Es wurden keine spezifischen Einstellungen, die den Planer des Linuxkerns oder die Prozessoraffinität einzelner Fäden verändern, vorgenommen.

Alle verwendeten Allokatoren wurden mit der in *Ubuntu 16.04* enthaltenen GNU Compiler Collection (gcc) in Version 5.4.0 und der Optimierungsstufe `-O2` erzeugt. Die Optimierungsstufe der Allokatoren *jemalloc* und *Hoard*, die standardmäßig `-O3` ist, musste, da diese für die *glibc* nicht verwendbar war, um gerechte Voraussetzungen zu schaffen, reduziert werden.

Die folgenden Versionen und Konfigurationsoptionen der Allokatoren wurden verwendet:

- *libtcmalloc* aus *gperftools* [4] in Version 2.7 vom 29. April 2018
- *glibc* [5] in Version 2.28 vom 14. August 2018
- *glibc-notc* dieselbe Version der *glibc* ohne fadenlokale Zwischenspeicher mit der Konfigurationsoption `--disable-experimental-malloc`
- *Hoard* [6] *master* Zweig des *git* Repositories vom 24. Mai 2018, letzter commit `baa62ab`, erzeugt mit gcc statt *clang* und der Optimierungsstufe `-O2`
- *jemalloc* [7] in Version 5.1.0 vom 8. Mai 2018 mit der Option `CFLAGS=-O2`

Um sicherzustellen, dass keine durch externe Einflüsse verfälschten Messwerte interpretiert werden, wurden alle Tests fünfmal durchgeführt und der Durchschnitt der Messergebnisse betrachtet.

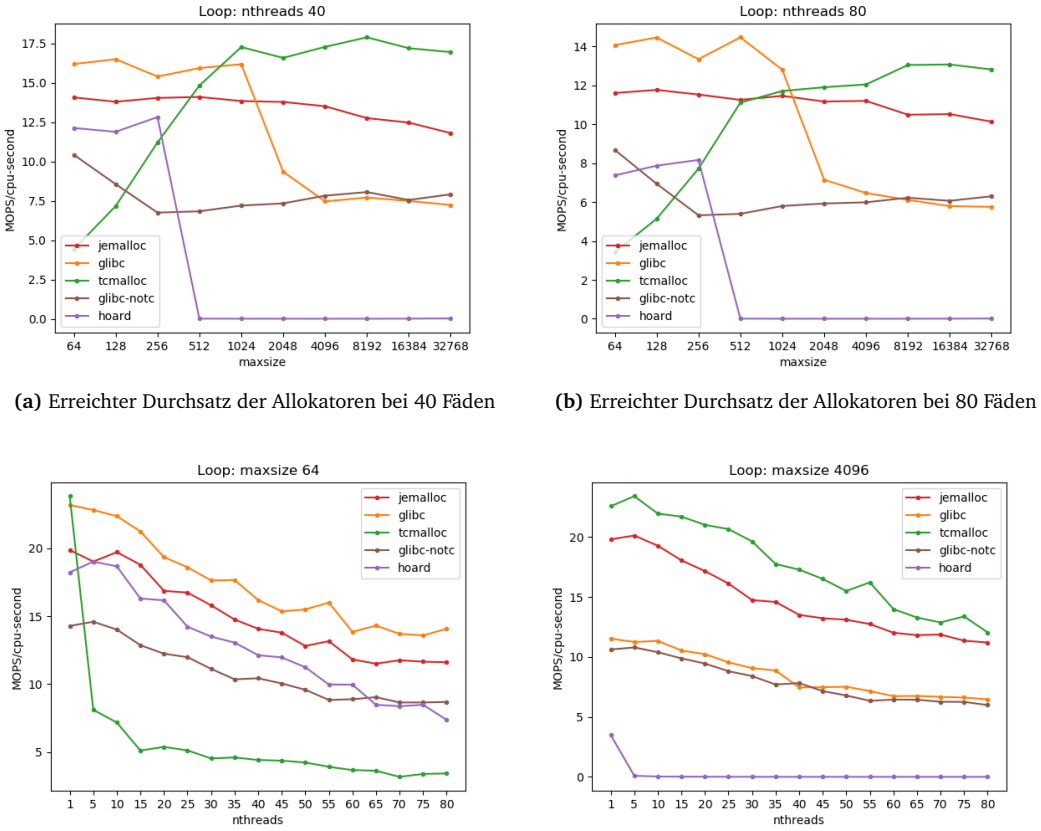
[4] `https://github.com/gperftools/gperftools/tree/gperftools-2.7`
[5] `git://sourceware.org/git/glibc.git`
[6] `https://github.com/emeryberger/Hoard/commit/baa62abb7eb873bd40068fbb96f44d7939164a20`
[7] `https://github.com/jemalloc/jemalloc/tree/5.1.0`

4.2 loop Test

Abschnitt 3.2.1.2

Alle im Folgenden dargestellten und interpretierten Werte weisen eine geringere Standardabweichung als 0.1% ihres Durchschnitts auf, das bedeutet, dass keine Ausreißer aufgrund externer Faktoren gemessen wurden und alle Werte verwendbar sind.

Abbildung 4.1 – Durchsatz des Tests `loop` (mehr ist besser)



(a) Erreichter Durchsatz der Allokatoren bei 40 Fäden

(b) Erreichter Durchsatz der Allokatoren bei 80 Fäden

(c) Erreichter Durchsatz der Allokatoren bei Maximalgröße 64 Byte

(d) Erreichter Durchsatz der Allokatoren bei Maximalgröße 4096 Byte

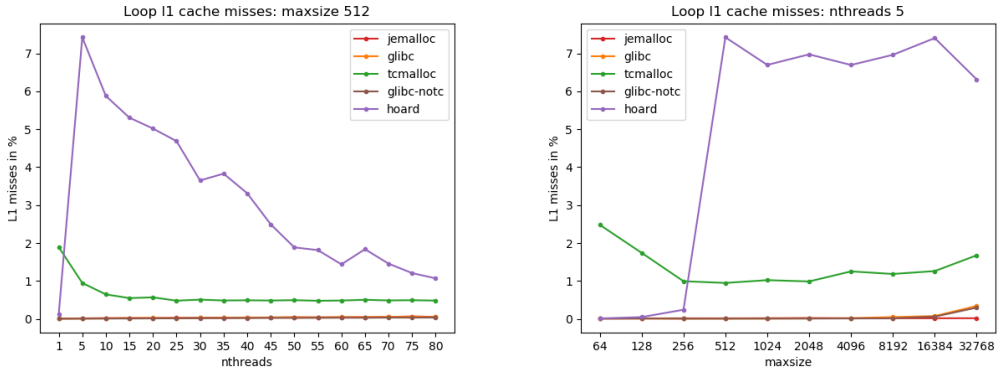
Abbildung 4.1 zeigt die Anzahl der durchgeführten Paare aus Allokation und Deallokation in Millionen pro benötigte CPU-Sekunde. Die ersten beiden Graphiken zeigen das Verhalten bei 40 und 80 Fäden und zunehmenden Allokationsgrößen. Zu sehen ist, dass fadenlokale Zwischenspeicher - gut zu beobachten bei der *glibc* - deutlich die Geschwindigkeit des kürzesten Codepfades und somit die Anzahl an Operationspaaren pro Sekunde erhöhen. Solange bei Größen unter 1032 Byte die Zwischenspeicher aktiv sind, wird fast der doppelte Durchsatz im Vergleich zu *glibc-notc* erreicht. Nachdem die Zwischenspeicher nicht mehr aktiv sind, ist zu beobachten, dass sich die zusätzliche Komplexität auch negativ auf die Leistung eines Allokators auswirken kann. Ein ähnlicher Einbruch wie der der *glibc* ist auch bei *Hoard* nach Größen über 256 Byte festzustellen. *Tcmalloc* verhält sich entgegengesetzt zu diesen Effekten und erreicht bei kleinen Allokationen nicht den ab 2048 Byte

vollen Durchsatz. Im Gegensatz zu allen anderen Allokatoren, auf die die Größe der Allokationen einen großen Einfluss zu haben scheint, verhält sich *jemalloc* sehr konsistent und vorhersagbar.

Die beiden unteren Graphiken zeigen den Durchsatz für feste Maximalgrößen und eine steigende Anzahl an parallelen Fäden. Dabei sind keine so starken Effekte durch die Anzahl an Fäden zu erkennen. Das bedeutet die Anzahl der parallelen Fäden ist für den Durchsatz der Allokatoren weniger entscheidend als die angeforderten Größen. Alle bereits erwähnten Effekte der Allokationsgröße sind wieder zu finden. Zum Beispiel die bei 4096 Byte kaum noch aktiven Zwischenspeicher der *glibc* erhöhen nur unmerklich den erzielten Durchsatz.

Festzuhalten ist, dass sich drei Sieger in unterschiedlichen Kategorien abzeichnen, *jemalloc* ist der Konsistenteste. Die *glibc* mit Zwischenspeichern ist für Anwendungen empfehlenswert, die durch viele kleine Allokationen diese gut ausnützen können und *tcmalloc* leidet unter wenigen kleinen Allokationen, aber erreicht für Allokationen größer als 1024 Byte den besten Durchsatz.

Abbildung 4.2 – L1-Daten-Zwischenspeicherfehler des Tests `loop` (weniger ist besser)

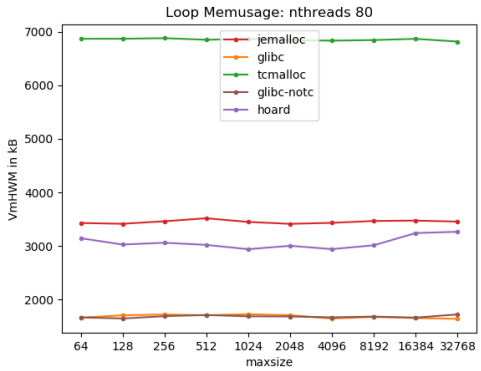


(a) L1-Daten-Zwischenspeicherfehler in Prozent bei der Maximalgröße 512

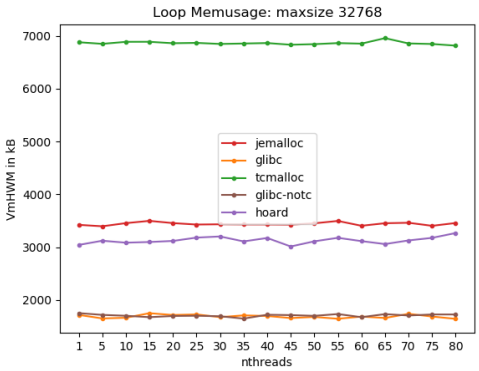
(b) L1-Daten-Zwischenspeicherfehler in Prozent bei 5 parallelen Fäden

Abbildung 4.2 zeigt, dass *Hoard* ab einer Allokationsgröße von 512 Byte deutlich mehr Zwischenspeicherfehler verursacht, was für eine höhere Verwendung eines geteilten inneren Zustands spricht. Eigentlich würde man diesen Effekt nach Aussagen des Papers erst ab der Hälfte einer Seite also 2048 Byte erwarten[Ber+00], da ab dieser Größe direkt Superblöcke aus dem zentralen Heap für die Allokation verwendet werden. Diese Strategie erklärt warum, *Hoard* für größere Allokationen schlechter skaliert als die Konkurrenten, die weiter weniger bis gar keine Synchronisationskosten haben.

Abbildung 4.3 – Speicherverbrauch des Tests `loop` (weniger ist besser)



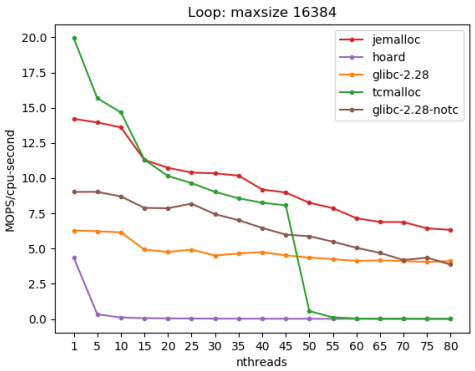
(a) Speicherverbrauch bei 80 parallelen Fäden



(b) Speicherverbrauch bei Maximalgröße 32768

Der gemessene Speicherverbrauch ist, wie Abbildung 4.3 zeigt, auf Grund der wenigen gleichzeitig aktiven Allokationen bei allen Testaufrufen nahezu konstant. Der beobachtete Speicherverbrauch ist demnach das Minimum, das die verschiedenen Speicherverwaltungen für ihre Funktionalität benötigen.

Abbildung 4.4 – Durchsatz eines leicht modifizierten `loop` Tests (mehr ist besser)

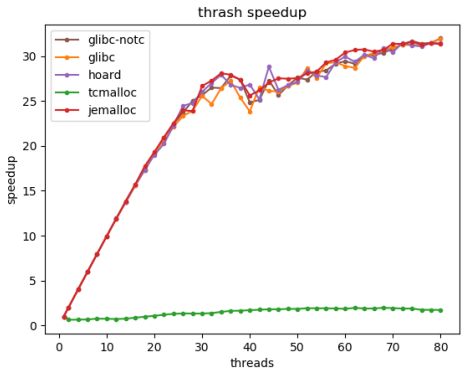


(a) Durchsatz bei Maximalgröße 16384 und 10 pro Faden vorgehaltenen Allokationen

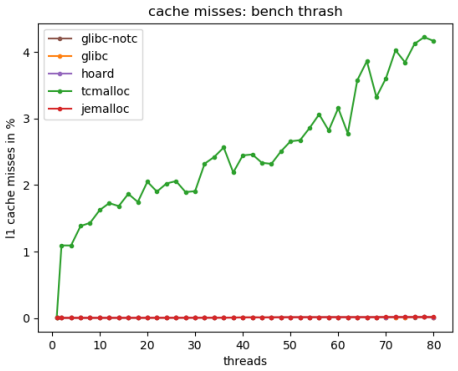
Zum Schluss ist in Abbildung 4.4 noch ein interessanter Effekt zu sehen, der in einer modifizierten Version des `loop` Tests, bei dem pro Faden 10 aktive Allokationen vorgehalten und am Stück freigegeben werden, auftritt. Es ist ein Einbruch von *tcmalloc* ab 50 Fäden zu beobachten. Dieser Effekt wird durch die häufige Auslösung der automatischen Speicherbereinigung verursacht, die Speicher aus einem zu vollen Zwischenspeicher in die globale Halde überführt, wobei immer synchronisiert werden muss. Das zeigt, dass nicht nur die unterschiedlichen Allokatoren sondern auch die verwendeten Parameter, um ihr Verhalten zu beeinflussen relevant sind. Verhindern könnte man den Einbruch von *tcmalloc* zum Beispiel dadurch, dass man die erlaubte Menge an Speicher in den fadenlokalen Zwischenspeichern per Umgebungsvariable erhöht.

4.3 falsesharing Tests

Abbildung 4.5 – Ergebnisse des Tests `cache-thrash`



(a) Linearer Speedup bedeutet keine aktive Verursachung irriger Mitbenutzung

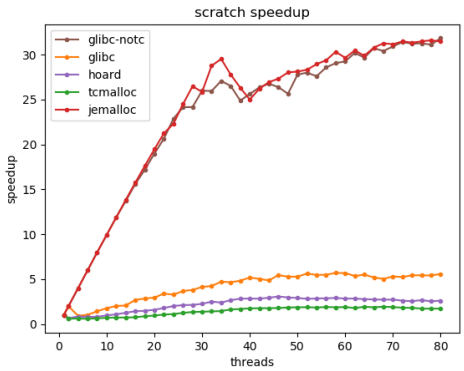


(b) L1-Daten-Zwischenspeicherfehler in Prozent

Abbildung 4.5 zeigt die Ergebnisse des Tests `cache-thrash`, der die aktive Einführung irriger Mitbenutzung durch den Allokator testet[Ber+00]. In der linken Graphik ist bei allen Allokatoren außer *tcmalloc* ein linearer Speedup bis zu zehn Fäden zu sehen. *Tcmalloc* erreicht gar keinen bis maximal zweifachen Speedup, was gut durch die in der rechten Graphik dargestellten L1-Daten-Zwischenspeicherfehler zu erklären ist. Warum kein unbegrenzter linearer Speedup erreicht werden kann, liegt am Test selbst. Da die Arbeit auf alle Fäden aufgeteilt wird, gibt es einen Punkt ab dem die Zeit, welche die wenigen Allokationen benötigen nicht mehr in der zu erledigenden Arbeit verschwindet. Ab diesem Zeitpunkt, wenn die Allokation einen nicht vernachlässigbaren Anteil der Gesamtzeit einnimmt kann, kein linerarer Speedup mehr erreicht werden.

`Cache-thrash` zeigt, dass nur *tcmalloc* aktiv irrige Mitbenutzung einführt. Der Grund dafür liegt in dem langsam startenden Algorithmus von *tcmalloc*. Es wird immer das Minimum von der wachsenden Maximallänge der aktuellen Größenklassenliste und einer Konstante an zu bewegenden Objekten tatsächlich bewegt. Da alle Größenklassenlisten mit der Maximallänge von eins initialisiert werden, wird bei der ersten Allokation einer Größenklasse nur ein Objekt in den fadenlokalen Zwischenspeicher bewegt. Deshalb erhalten zwei unterschiedliche Fäden zwei direkt hintereinanderliegende Speicherstücke, die sich mit hoher Wahrscheinlichkeit eine Zwischenspeicherzeile teilen. Diese aktive Einführung von irriger Mitbenutzung ist möglich, solange die Maximallängen der Größenklassen kleiner als eine Zwischenspeicherzeile, unter der die Konstante liegt und dadurch nicht ausschließlich komplette Zwischenspeicherzeilen in den Fadenspeicher bewegt werden. Eine einfache Lösung, um die aktive Einführung irriger Mitbenutzung zu verhindern, wäre es für Objekte, die kleiner als eine Zwischenspeicherzeile sind, immer mindestens eine ganze Zeile in einen neuen fadenlokalen Zwischenspeicher zu bewegen. Mit einer anderen Vorinitialisierung der Maximallängen wäre dies leicht zu bewerkstelligen.

Abbildung 4.6 – Ergebnisse des Tests `cache-scratch`



(a) Linearer Speedup bedeutet keine passive oder aktive Verursachung irriger Mitbenutzung

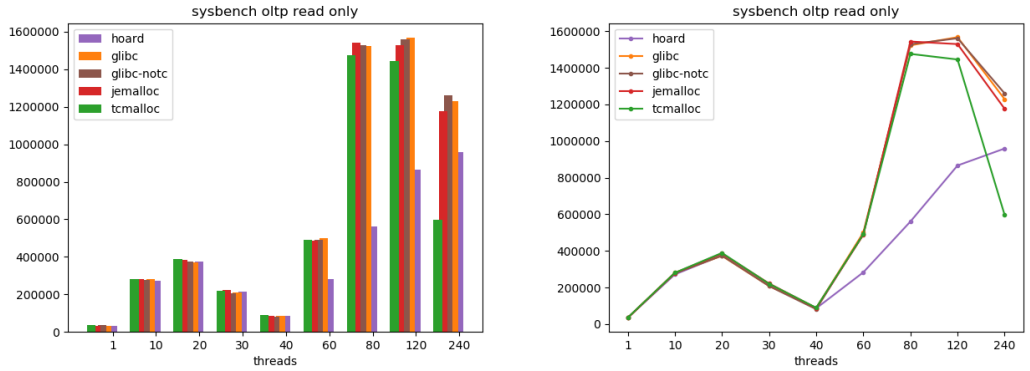
(b) L1-Daten-Zwischenspeicherfehler in Prozent

Der Test `cache-scratch` prüft zusätzlich, ob der Allokator passiv irrige Mitbenutzung ermöglicht, indem er Blöcke nach einem `free` an einen anderen Faden als an ihren ursprünglich Besitzer ausgibt. Abbildung 4.6 zeigt, dass nur *jemalloc* und die *glibc* ohne fadenlokale Zwischenspeicher durch den Allokator beeinflussbare irrige Mitbenutzung gänzlich verhindern. Verwunderlich ist dabei, dass *Hoard*, der nach Aussagen seiner Autoren entwickelt wurde, um irrige Mitbenutzung zu verhindern[Ber+00], dieses Versprechen bei den eigenen Vergleichstest nicht einhält. Bei der *glibc* ist dieses Verhalten hingegen zu erwarten, da durch die relativ neuen fadenlokalen Zwischenspeicher verhindert wird, dass ein Block immer in seine ursprüngliche Arena zurückgegeben wird. Falls Platz im Zwischenspeicher des freigebenden Fadens ist, wird der Block dort eingehängt, unabhängig davon wo dieser ursprünglich alloziert wurde. Dadurch wird die eindeutige Zuordnung von Fäden und ihren Allokationen zu Arenen und somit Speicherbereichen, welche die Einführung irriger Mitbenutzung sehr unwahrscheinlich macht, aufgehoben. Die naheliegende Lösung für dieses Problem, das alle fadenlokalen Zwischenspeicher haben, ist es zumindest Blöcke, die kleiner als eine Zwischenspeicherzeile sind, nur in den Zwischenspeicher eines Fadens einzuhängen, der aktuell die Arena benutzt, aus der der Block stammt. Da *jemalloc* nicht unter diesem Effekt leidet, ist davon auszugehen, dass dort diese oder eine ähnliche Vorgehensweise implementiert ist.

Da sich `cache-scratch` nur marginal von `cache-thrash` unterscheidet tritt hier derselbe Effekt, ab 30 Fäden ein.

4.4 mysql Test

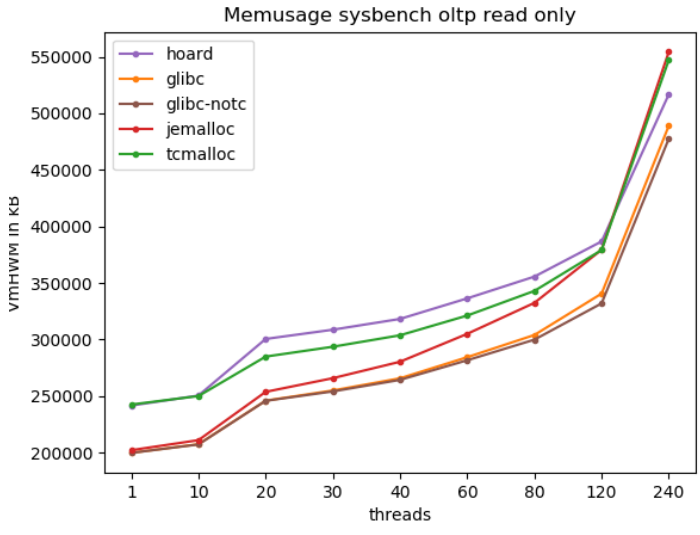
Abbildung 4.7 – Ergebnisse `mysql`



(a) Transaktionen in 60 Sekunden (mehr ist besser).

Allokator / Fäden	1	10	20	30	40
hoard	34205.000**	271142.800**	375325.600*	212962.600**	86124.200*
glibc	34042.200*	280008.200**	372569.000**	207976.400*	87676.800
glibc-notc	34588.600*	276729.600**	373811.800*	207671.600*	81328.000*
jemalloc	34121.200*	279328.400**	385411.600*	221710.200**	87666.000*
tcmalloc	35025.000*	280102.800*	388673.200*	220173.800*	89351.000*
	60	80	120	240	
hoard	282659.800	560103.200*	866168.400*	958981.000*	
glibc	499855.400*	1522341.800*	1567146.400**	1228011.400**	
glibc-notc	492538.400	1528567.200*	1561061.400*	1259714.000*	
jemalloc	487286.400*	1543318.600*	1529248.800**	1175650.200*	
tcmalloc	490404.000*	1475776.600*	1445111.400**	595721.000*	

(b) Anzahl der Transaktionen in 60 Sekunden. (Mehr ist besser. Grün = Bester. Rot = Schlechtester)
Standardabweichung in Prozent des Durchschnitts: $= 0.2-0.1\%$; $* = 0.1-0.01\%$; $** = 0.01-0.001\%$



(c) Speicherverbrauch des *Mysql* Servers (niedriger ist besser).

Das Erste, das bei der Betrachtung von Abbildung 4.7 auffällt, ist, dass zwischen 20 und 40 Fäden kein Anstieg der Transaktionen sondern ein Rückgang zu beobachten ist. Dies ist höchstwahrscheinlich ein Effekt, der durch die NUMA Architektur des verwendeten Testsystems entsteht. Die Kommunikationskosten für die zusätzlichen NUMA Domänen überwiegen den Zugewinn der hinzukommenden Fäden. Erst bei 80 Fäden (20 Fäden pro NUMA Domäne) kann die Hardware maximal ausgenutzt werden. Dieses Verhalten war auf einem konventionellen Laptop nicht zu beobachten.

Außerdem ist zu sehen, dass mehr logische Fäden als die maximale Anzahl an verfügbaren physikalischen Fäden keinen Zuwachs an Transaktionen verursachen. *Tcmalloc* erzielt bis 40 Fäden mit einer Ausnahme bei 30 Fäden den höchsten Durchsatz. Ab 60 Fäden erreicht der *MySQL* Server mit der *glibc* die größte Leistung für rein lesende Anfragen. Die Unterschiede zwischen den zwei vermessenen *glibc* Versionen sind oft sehr knapp, aber bei 5 aus 9 Fadenanzahlen sorgen die Zwischenspeicher für eine höhere Leistung.

Den geringsten Speicherbedarf hat durchgängig die *glibc*, wobei die Version ohne fadenlokale Zwischenspeicher bei mehr logischen Fäden als Hardwarefäden leicht speichereffizienter arbeiten kann, weil kein Speicher unbenutzt in Zwischenspeicher nicht aktiver logischer Fäden gehalten wird. *Jemalloc* verbraucht mehr zusätzlichen Speicher pro Faden als die anderen Allokatoren und überholt deshalb bei 120 Fäden *tcmalloc* und *Hoard* mit seinem Speicherverbrauch.

Der durchgängig am schlechtesten abschneidende Allokator ist *Hoard*. Dies ist allerdings nicht verwunderlich, da er ein Forschungsallokator ist und im Gegensatz zu seiner Konkurrenz nicht mehr weiterentwickelt wird.

Die Ergebnisse des `mysql` Tests zeigen außerdem, das nicht nur die Anwendung sondern auch deren Parameter entscheidend für die Auswahl des passenden Allokators sind.

4.5 dj_trace Tests

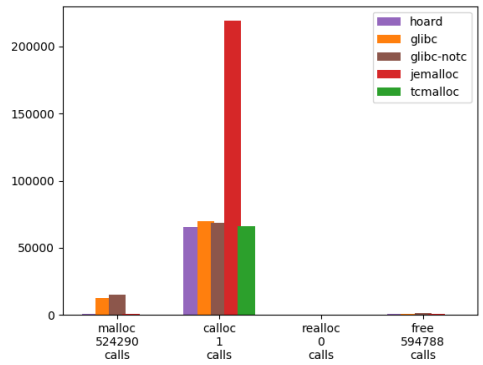
Im Folgenden werden die Ergebnisse der Simulationen der von DJ Delorie bereitgestellten Lastprofile gruppiert nach der Art der aufgezeichneten Anwendungen präsentiert.

Während der Simulation werden Allokationen nicht beschrieben oder gelesen, deswegen ist die gemessene Gesamtausführungszeit nicht direkt übertragbar auf eine echte Anwendung mit gleichem Profil, weil Datenlokalität und irrige Mitbenutzung keine Rolle in der Simulation spielen. Viel mehr ist die Gesamtlaufzeit als die Zeit, die die Anwendung in der Allokatorbibliothek verbringen wird, zu verstehen.

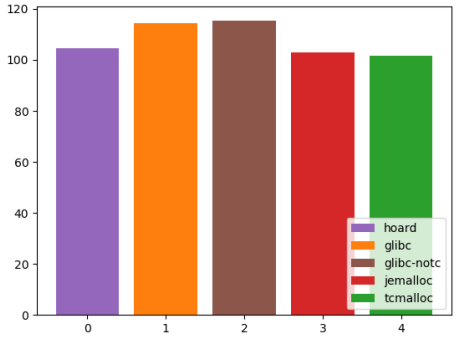
4.5.1 Synthetische Lastprofile

Die synthetischen Profile unterscheiden sich stark sowohl untereinander als auch von den restlichen realen Profilen.

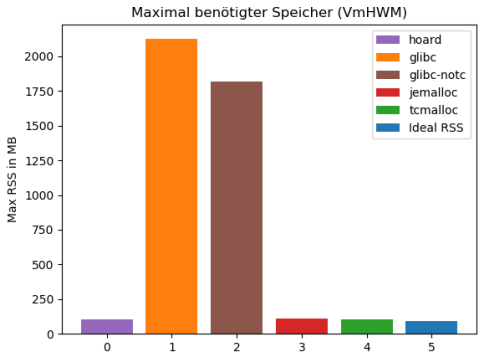
Abbildung 4.8 – Ergebnisse `mt_test_one_alloc` (2 Fäden - Niedriger ist besser)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



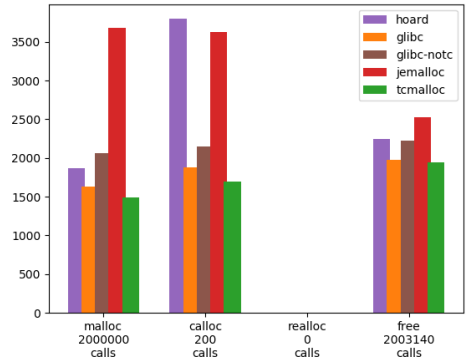
	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	104745.231 / 0.065	103863.200 / 0.000
glibc	114424.426 / 0.003	2125046.400 / 0.000
glibc-notc	115328.528 / 0.000	1820332.000 / 0.000
jemalloc	102932.107 / 0.001	110633.600 / 0.002
tcmalloc	101641.976 / 0.001	104143.200 / 0.000

Bei der Betrachtung der Laufzeiten der einzelnen Funktion ist zu beachten, dass die Zeiten der `calloc` Funktion keine Aussagekraft besitzen, da die Funktion nur ein einziges Mal aufgerufen wurde.

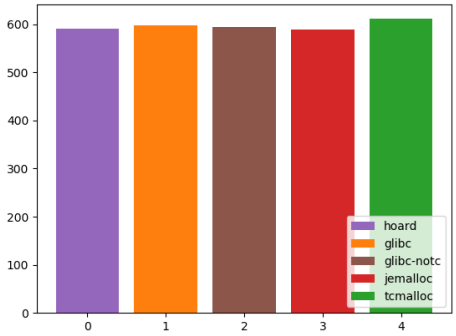
Zu sehen ist, dass das für die *glibc* erzeugte Worst-Case Verhalten, die anderen Allokatoren nicht betrifft. Die abwechselnden Allokationen von großen und kleinen Blöcken verhindert das erfolgreiche Verschmelzen von Blöcken und sorgt mit den weit schwankenden zufälligen Größen dafür, dass kaum Speicher wiederverwendet werden kann, das zu dem enormen Speicherverbrauch von mehr als 2GB führt. Außerdem sind die deutlich längeren Laufzeiten der `malloc` Funktion dadurch zu erklären, dass der in Algorithmus 2.2 in Zeile 10 beschriebene Fall eintritt und bei jeder Allokation versucht wird, alle Listen zu sortieren und zu verschmelzen. Aufgrund der vorangegangenen kleinen Allokation, die auch deutlich länger behalten wird als die nach ihr allozierte Große, ist das Verschmelzen sehr selten möglich ist.

Die anderen Allokatoren liegen maximal 3 CPU-Sekunden auseinander, was weniger als 3% der Gesamtlaufzeit entspricht. Am schnellsten schneidet dabei *tcmalloc* ab. Auch der Speicherverbrauch der restlichen Allokatoren unterscheidet sich nicht auffällig voneinander und liegt nahe am minimal nötigen Speicherverbrauch.

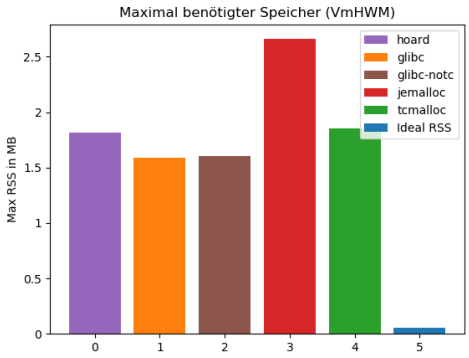
Abbildung 4.9 – Ergebnisse `dj` (201 Fäden - Niedriger ist besser)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden

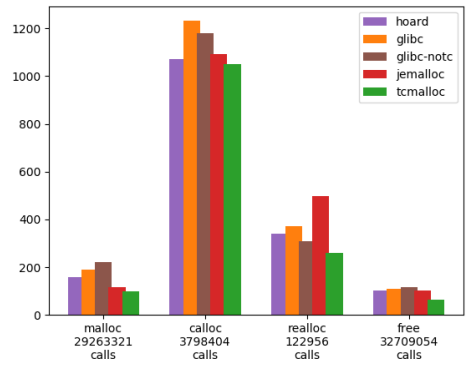


	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	591335.067 / 0.038	1816.000 / 0.010
glibc	598328.699 / 0.021	1588.000 / 0.002
glibc-notc	593585.267 / 0.038	1600.800 / 0.015
jemalloc	589166.555 / 0.037	2660.000 / 0.027
tcmalloc	611182.772 / 0.029	1852.800 / 0.001

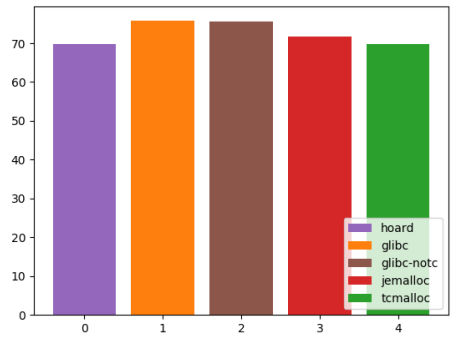
Abbildung 4.9 zeigt, dass beim Test `dj` alle Allokatoren außer *tcmalloc* nahezu gleich schnell sind. Verblüffend ist die nur bei diesem Test beobachtbare Diskrepanz zwischen gemessenen API Zeiten und Gesamtlaufzeit. *Jemalloc* hat trotz deutlich längeren Funktionszeiten die kürzeste Gesamtlaufzeit. Genau das Gegenteil trifft für *tcmalloc* zu. Dieser Effekt tritt zuverlässig auf und ist durch keinen der mit *perf -dd* gemessenen Werte erklärbar.

Trotz der Anomalie bei den gemessenen Zeiten ist das gemessene Speicherverhalten aussagekräftig. Zu sehen ist, dass alle Allokatoren deutlich mehr als den minimal erforderlichen Speicher benötigen. Außerdem zeigt sich, dass die Strategien der *glibc* bei gleichmäßigen Profilen zu guten Ergebnissen führen und beinahe halb soviel Speicher benötigen wie der schlechteste *jemalloc*.

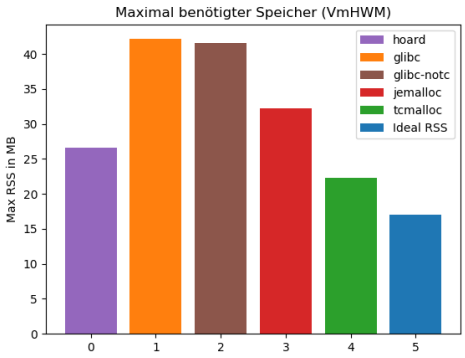
Abbildung 4.10 – Ergebnisse `dj2` (36 Fäden - Niedriger ist besser)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



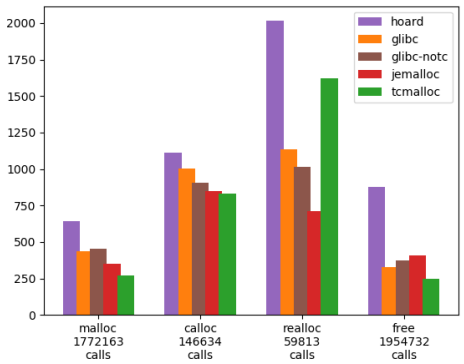
	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	69759.018 / 0.050	26605.600 / 0.005
glibc	75769.156 / 0.039	42160.800 / 0.002
glibc-notc	75584.885 / 0.031	41627.200 / 0.003
jemalloc	71801.043 / 0.049	32184.000 / 0.006
tcmalloc	69733.917 / 0.049	22359.200 / 0.001

Dieser stark zufällige Test sorgt wie bereits `mt_test_one_alloc` für schlechtes Verhalten bei der *glibc*. Der komplexe Algorithmus der *glibc*, der viel zusätzliche Arbeit wie das Teilen und Verschmelzen von Blöcken erledigt, mit der Annahme, dass diese Arbeit nicht oft erledigt werden muss und vorteilhaft für tatsächliche Anwendungen ist, verschwendet viel unnötige Zeit bei zufälligen Anfrageprofilen. Wohingegen der einfache, immer gleiche Algorithmus von *tcmalloc* diese Schwächen nicht hat, dadurch durchgängig gute Leistung zeigt und in diesem Test sowohl der schnellste als auch speichereffizienteste Allokator ist.

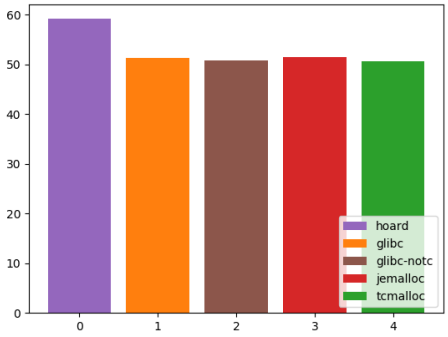
Die absoluten Ergebnisse dieser synthetischen Tests dürfen nicht überinterpretiert werden. Allerdings geht aus ihnen hervor, dass die *glibc* schlecht für stark variierende und zufällige Anfrageprofile ist.

4.5.2 Lastprofile der Virtualisierungssoftware *qemu*

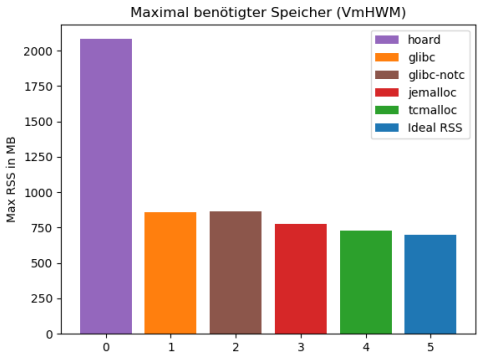
Abbildung 4.11 – Ergebnisse `qemu-virtio` (3 Fäden - Niedriger ist besser)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



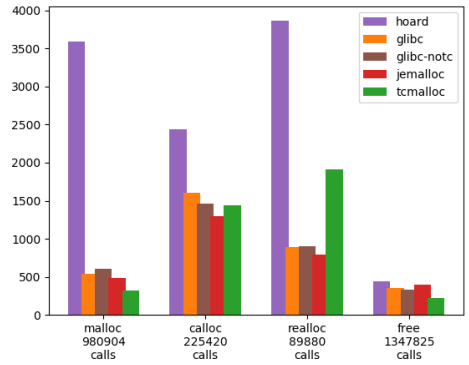
	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	59140.035 / 0.004	2082314.400 / 0.000
glibc	51270.294 / 0.015	857869.600 / 0.010
glibc-notc	50726.310 / 0.015	863619.200 / 0.006
jemalloc	51374.258 / 0.016	776719.200 / 0.002
tcmalloc	50608.111 / 0.013	729124.800 / 0.001

Wie in Abbildung 4.11 zusehen ist, liegen alle Allokatoren bis auf der Ausreißer *Hoard* innerhalb eines Intervalls von 600ms, was weniger als einem Prozent der Gesamtlaufzeit von ca. 50 CPU-Sekunden entspricht. Nur *Hoard* ist um ungefähr 8 CPU-Sekunden langsamer als die Konkurrenten.

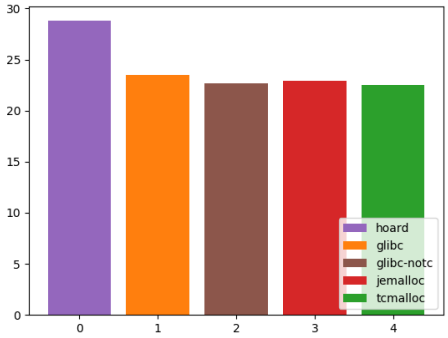
Der Speicherverbrauch weist größere Unterschiede auf als die Laufzeiten, dabei braucht *Hoard* mehr als zweieinhalb mal so viel Speicher als die anderen Speicherverwaltungen, die in einem deutlichen Intervall von 130 MB, 15 — 18 % des Gesamtverbrauchs, liegen.

Tcmalloc schneidet am besten ab und ist nicht nur der Schnellste sondern benötigt auch 50MB weniger Speicher als der Zweitbeste.

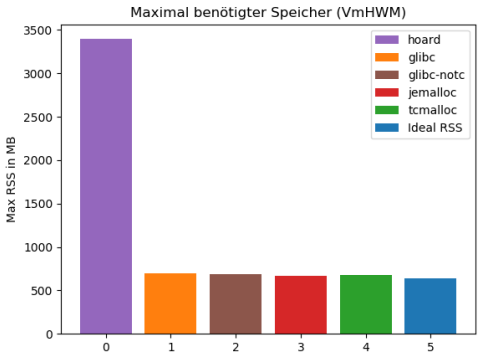
Abbildung 4.12 – Ergebnisse `qemu-win7` (6 Fäden. Niedriger ist besser.)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



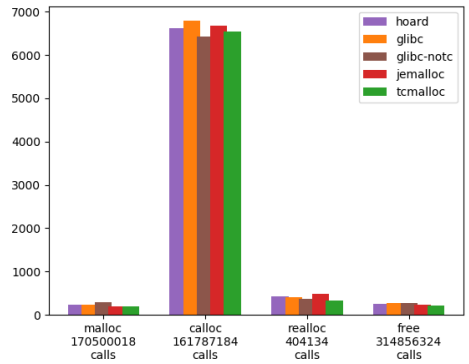
	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	28793.758 / 0.031	3396071.200 / 0.000
glibc	23463.092 / 0.027	692780.000 / 0.001
glibc-notc	22710.053 / 0.017	691672.000 / 0.001
jemalloc	22938.326 / 0.022	669662.400 / 0.003
tcmalloc	22515.075 / 0.016	673060.000 / 0.001

In Abbildung 4.12 zeigt sich ein ähnliches Bild wie beim vorangegangenen Test. Zeitlich unterscheiden sich die Allokatoren um ca. 0.9 CPU-Sekunden, was ungefähr 4% der Gesamtlaufzeit entspricht. Auch der Speicherverbrauch liegt eng beieinander in einem Intervall von ca. 2%. Nur *Hoard* ist wieder in beiden Metriken deutlich schlechter als die Anderen. Diesmal ist er 6 CPU-Sekunden langsamer und benötigt ungefähr viermal so viel Speicher.

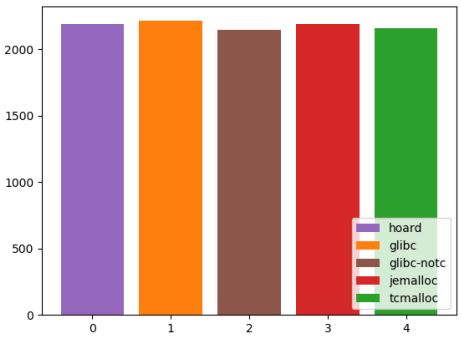
Warum *Hoard* für die Profile der Virtualisierungssoftware *qemu* deutlich schlechter abschneidet als die anderen Allokatoren, ist nicht über die Größen der Allokationen, die fast ausschließlich unter 512 Byte liegen, zu erklären. Da laut Paper bei diesen Größen die Größenklassen von *Hoard* aktiv sind und somit nicht viel interne Fragmentierung und Synchronisation entstehen dürfte [Ber+00]. Da *Hoard* aber auch im `loop` Test bei 512 Byte einbricht, ist davon auszugehen, dass es sich um denselben Effekt handelt.

4.5.3 Serverlastprofile

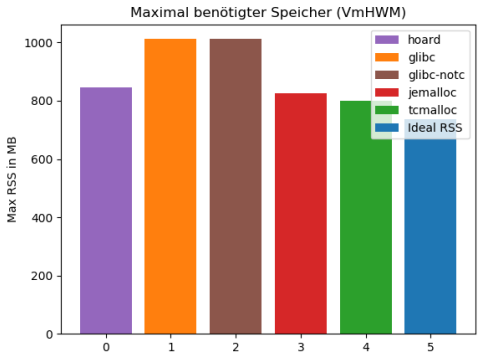
Abbildung 4.13 – Ergebnisse `389-ds-2` (41 Fäden. Niedriger ist besser.)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden

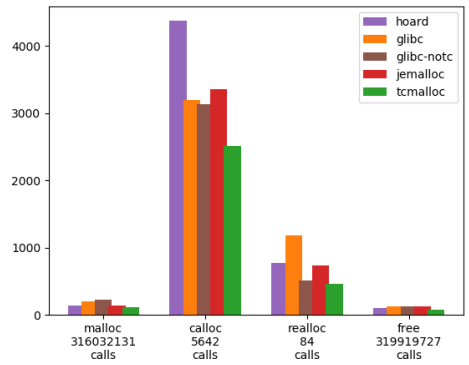


	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	2187051.905 / 0.047	845284.800 / 0.000
glibc	2213888.664 / 0.023	1012038.400 / 0.000
glibc-notc	2146537.107 / 0.006	1011712.800 / 0.000
jemalloc	2191590.016 / 0.021	826196.800 / 0.000
tcmalloc	2156022.870 / 0.016	801092.000 / 0.000

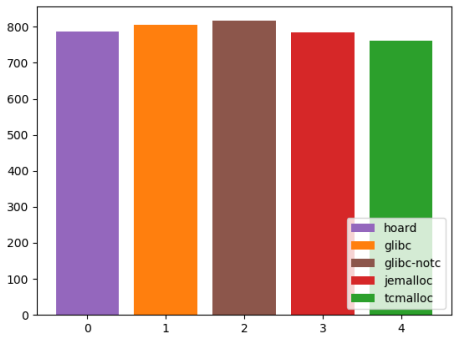
Interessant bei diesem Test ist, wie der Tabelle in Abbildung 4.13 zu entnehmen ist, dass die *glibc* sowohl schnellster als auch langsamster Allokator ist. Die fadenlokalen Zwischenspeicher der *glibc* führen zu einem Zeitverlust von 67 CPU-Sekunden, was ca. 3% der Gesamtlaufzeit entspricht. Beim Speicherverbrauch ist ebenfalls die Version ohne Zwischenspeicher besser, aber es ändert nichts daran, dass beide mit 200MB (ca. 20% Abstand) schlechter sind als die Konkurenzimplementierungen.

Einen eindeutig besten Allokator gibt es nicht, allerdings ist *tcmalloc* wieder der Speichereffizienteste und nur knapp langsamer als die *glibc* ohne Zwischenspeicher.

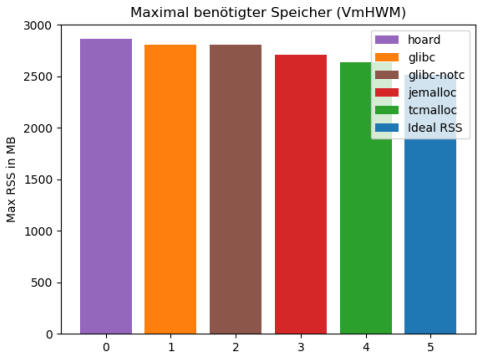
Abbildung 4.14 – Ergebnisse `proprietary-1` (20 Fäden. Niedriger ist besser.)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden

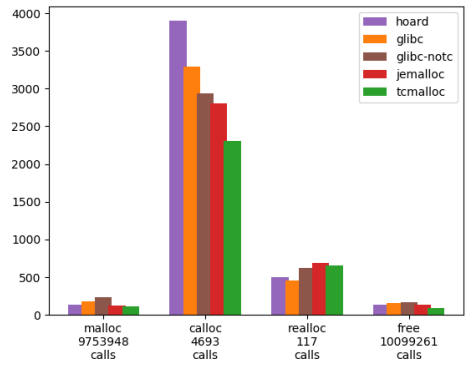


	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	786878.766 / 0.010	2862251.200 / 0.024
glibc	805461.972 / 0.011	2810572.800 / 0.001
glibc-notc	816682.810 / 0.011	2807924.800 / 0.000
jemalloc	783605.183 / 0.007	2707145.600 / 0.000
tcmalloc	761164.698 / 0.007	2638352.000 / 0.002

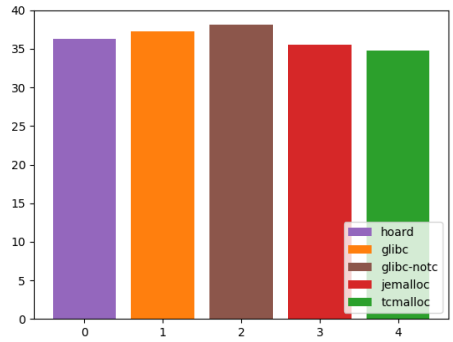
Im Test *proprietary-1* schneidet abermals *tcmalloc* in beiden Kategorien am besten ab. Der Zeitunterschied beträgt 50 CPU-Sekunden zum Schlechtesten (*glibc* ohne Zwischenspeicher), ungefähr 6% der Gesamtlaufzeit. Ebenfalls können 230MB (ca. 8%) durch die Verwendung von *tcmalloc* im Vergleich zu *Hoard* eingespart werden.

Zu beachten ist auch, dass die Version der *glibc* mit fadenlokalen Zwischenspeichern schneller ist aber dafür auch mehr Speicher benötigt.

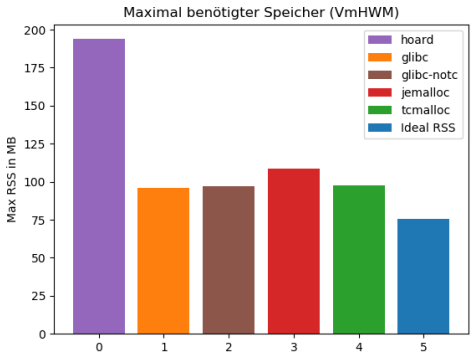
Abbildung 4.15 – Ergebnisse `proprietary-2` (19 Fäden. Niedriger ist besser.)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



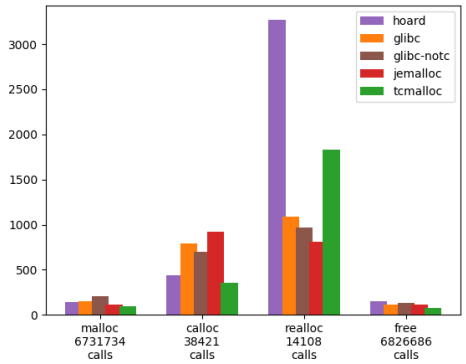
	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	36328.467 / 0.014	193839.200 / 0.001
glibc	37310.561 / 0.020	96082.400 / 0.008
glibc-notc	38117.419 / 0.014	96836.000 / 0.012
jemalloc	35500.163 / 0.037	108676.800 / 0.007
tcmalloc	34810.599 / 0.034	97573.600 / 0.001

Bei `proprietary-2` liegen die Gesamtlaufzeiten der Allokatoren in einem Intervall von 3.4 CPU-Sekunden (8-10%) mit *tcmalloc* an der Spitze. Im Speicherverbrauch fällt nur *Hoard* mit fast dem doppeltem Verbrauch, aus dem 40MB (4%) Intervall der Anderen.

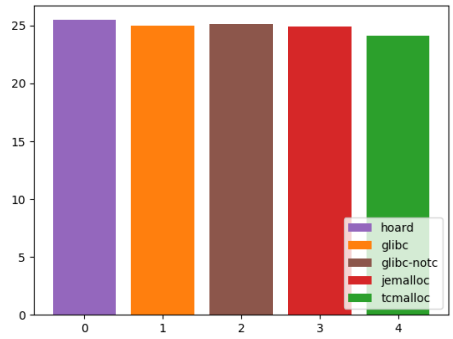
Im Gegensatz zu den anderen beiden Servertests, bei denen die *glibc* nicht speichereffizient arbeiten konnte, ist sie für dieses Profil in beiden Versionen der sparsamste Allokator.

4.5.4 Desktoplastprofil

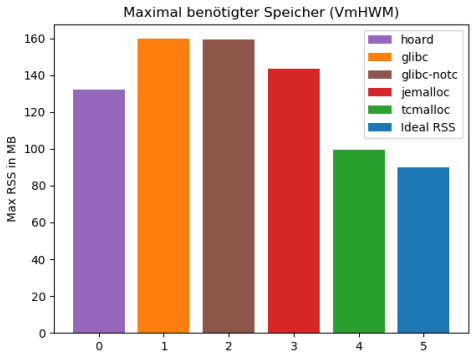
Abbildung 4.16 – Ergebnisse `oocalc` (81 Fäden. Niedriger ist besser.)



(a) Durchschnittliche Zeit der API Funktionen in cycles



(b) Gesamt benötigte Laufzeit in CPU-Sekunden



	Zeit (ms) / σ (%)	VmHWM (KB) / σ (%)
hoard	25485.900 / 0.009	132328.800 / 0.004
glibc	25014.041 / 0.013	159705.600 / 0.006
glibc-notc	25155.863 / 0.022	159175.200 / 0.002
jemalloc	24936.215 / 0.008	143236.800 / 0.001
tcmalloc	24152.145 / 0.023	99508.800 / 0.001

Auch bei der vermessenen Desktopanwendung zeigen sich die Vorteile von *tcmalloc*. Die Speicherverwaltung von *Google* schneidet abermals in beiden Metriken am besten ab. Sie ist sogar, was den Speicherverbrauch angeht, deutlich (37 MB ca. 33%) besser als der Zweitbeste.

4.5.5 Zusammenfassung und Diskussion der Ergebnisse

Wie in Abbildungen 4.5 und 4.6 zu sehen ist, verursacht *tcmalloc* sowohl aktiv als auch passiv irrige Mitbenutzung. Das heißt diese Tests, bei denen dieser Aspekt komplett wegfällt, begünstigen besonders *tcmalloc*, der - wie beim `loop` Test in Abschnitt 3.2.1.2 - zu sehen einen dauerhaft hohen Durchsatz erreicht. Außerdem ist die Strategie von *jemalloc*, auf hohe Datenlokalität zu achten, in diesem Test nicht repräsentiert wodurch, *jemalloc* schlechter erscheint als er für reale Anwendungen wäre.

Was die Tests aber zeigen ist, dass *tcmalloc* sieben von acht mal am wenigsten Zeit benötigt, um die Anfragen zu erledigen und dabei bei fünf von sechs realen Profilen am speichereffizientesten vorgeht. Das bedeutet, dass für die untersuchten Profile der mehr Aufwand für Buchhaltung der Zwischenspeichergrößen zumindest im Vergleich zu *jemalloc*, der darauf mit einer periodischen automatische Speicherbereinigung verzichtet, sich auszahlt.

4.6 Resümee

Die durchgeführten Tests lassen deutliche Unterschiede zwischen den verschiedenen Allokatortechniken erkennen. Das traditionelle Vorgehen, Blöcke zu zerteilen und wieder zuverschmelzen, ist anfällig für ungeeignete Anfrageprofile mit sich stark in Größe unterscheidenden Allokationen sowie stark unterschiedliche Lebenszeiten. Die anderen Allokatoren, die Größenklassen physisch trennen, leiden nicht unter diesen Mustern.

Die gemessenen Unterschiede sind oft marginal und keinesfalls so, dass sich eine gänzlich überlegene Speicherverwaltung abzeichnet. Vorallem Abschnitt 4.5 erweckt den Eindruck, dass der Durchsatz weitgehend gleich ist und die Metrik, nach der gut optimiert werden kann, der benötigte Speicher ist. Eingesparte 200MB können anderweitig zum Beispiel für einen Festplattenzwischenspeicher eingesetzt werden, um so die Anwendung wiederum zu beschleunigen.

Es zeigt sich jedoch, dass Hoard oft deutlich schlechter ist als die Anderen. Das schlechte Speicherverhalten liegt daran, dass ein Forschungsgegenstand von Hoard Skalierbarkeit für kleine Allokationen ist und für Allokationen ab halber Seitengröße immer ganze Superblöcke ausgegeben werden, was zu sehr hoher interner Fragmentierung führen kann.

5 FAZIT

Der mit der entwickelten Umgebung *allocbench* durchgeführte Vergleich, der in Kapitel 2 vorgestellten Allokatoren und ihrer verwendeten Techniken zeigt, dass das Thema dynamische Speicherverwaltung keines Falls gelöst ist. Trotz oft marginaler Unterschiede in den Geschwindigkeiten der Allokatoren, was großteils daran liegt, dass fadenlokale Zwischenspeicher quasi Standard geworden sind, zeigt sich ein Unterschied in der Speichereffizienz für bestimmte Anfrageprofile. Während Teile und Verschmelze Verfahren für Anfragen mit stark variierenden Größen und unterschiedlichen Lebenszeiten schlechte Speichernutzung verursachen weißen sie einen geringen Grundverbrauch auf und lohnen sich bei Anfrageprofilen mit vielen ähnlichen Allokationen.

Außerdem ging aus der Interpretation des Vergleichs hervor, dass die in *allocbench* enthaltenen Vergleichstests ausreichen um befriedigend zu behaupten ein Allokator wäre grundsätzlich besser als der andere. Ergänzende Test könnten das innere Verhalten der Allokatoren auf irrige Mitbenutzung und Lokalität überprüfen. Darüber hinaus wären Tests wünschenswert die extrem Verhalten für unterschiedliche Hardware Topologien erzeugen. Sowohl rein synthetische Tests, wie sie *ACDC* generiert [AK13] als auch zwar reale aber sehr spezifische, wie der enthaltene Test `mysql`, können nur einzelne nicht funktionale Eigenschaften evaluieren uns ermöglichen deshalb kaum Aussagen über konkretes Anwendungsverhalten zu oder testen ein eventuell ein grundlegend anderes Lastprofil. Deshalb ist es von großer Bedeutung, dass die Anwendung direkt integriert und vermessen werden kann. Anhand der realen Anwendungsergebnisse kann abgewägt welche Eigenschaften für das Einsatzszenario am wichtigsten sind und gezielt nach diesen optimiert werden.

Aber nicht nur für die Vermessung einer bestimmten Anwendung kann *allocbench* eingesetzt werden, der Vergleich hat auch interessante Unterschiede für die zwei unterschiedlichen Versionen der *glibc* aufgezeigt.

Darüber hinaus ist aufgefallen, dass historisch bedingt der Fokus sowohl der Forschung als auch der Technik auf den Wechselwirkungen mit dem physikalischen Zwischenspeichern als Hauptkriterium für unterschiedliche Zugriffskosten untersucht und behandelt werden. Keiner der untersuchten Allokatoren betrachtet ein größeres Bild der Hardware, wie NUMA Architekturen. Die vereinzelten Ansätze der Forschung wie *palloc*, einem Allokator, der versucht einzelne DRAM Zellen exklusiv Prozessorkernen zuzuweisen um somit die Zugriffkosten auf Mehr-Kern Systemen zu minimieren [Yun+14], oder *Xalloc*, einem Allokator, der entwickelt wurde um nicht nur für eine zunehmende Anzahl an Kernen sondern auch für eine zunehmenden Anzahl an Single Instruction, Multiple Data (SIMD) Einheiten zu skalieren [Hua+10], haben ihren Weg noch nicht in die gebräuchlichen Allokatoren gefunden.

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface

CPU Central Processing Unit

LIFO Last In First Out

GNU GNU's Not Unix

TLS Thread-local Storage

LWP Light-weight Process

LDAP Lightweight Directory Access Protocol

NUMA Non-Uniform Memory Access

gcc GNU Compiler Collection

SIMD Single Instruction, Multiple Data

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

QUELLCODEVERZEICHNIS

ALGORITHMENVERZEICHNIS

LITERATUR

[AK13] Martin Aigner und Christoph M Kirsch. “ACDC: towards a universal mutator for benchmarking heap management systems”. In: *ACM SIGPLAN Notices*. Bd. 48. 11. ACM. 2013, S. 75–84.

[Ber+00] Emery D. Berger u. a. “Hoard: a scalable memory allocator for multithreaded applications”. In: *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ASPLOS-IX. Cambridge, Massachusetts, United States: ACM, 2000, S. 117–128. ISBN: 1-58113-317-0. DOI: http://doi.acm.org/10.1145/378993.379232. URL: http://doi.acm.org/10.1145/378993.379232.

[Del17] DJ Delorie. *malloc per-thread cache: benchmarks*. 2017. URL: https://sourceware.org/ml/libc-alpha/2017-01/msg00452.html.

[DO] DJ Delorie und Charles ODonell. *Malloc Internals*. URL: https://sourceware.org/glibc/wiki/MallocInternals.

[Eva06] Jason Evans. “A scalable concurrent malloc (3) implementation for FreeBSD”. In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[Eva11] Jason Evans. “Scalable memory allocation using jemalloc”. In: *Notes Facebook Eng* (2011). URL: https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919.

[GM07] Sanjay Ghemawat und Paul Menage. *TCMalloc: Thread-caching malloc, 2007*. 2007. URL: https://gperftools.github.io/gperftools/tcmalloc.html.

[Gro13] IEEE/The Open Group. *malloc(3) POSIX Programmer’s Manual*. 2013.

[Hua+10] Xiaohuang Huang u. a. “Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines”. In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE. 2010, S. 1134–1139.

[Lab79] Bell Labs. *malloc(3) Unix V7 User’s Manual*. http://man.cat-v.org/unix_7th/3/malloc. 1979.

[LG96] Doug Lea und Wolfram Gloger. *A memory allocator*. http://g.oswego.edu/dl/html/malloc.html. 1996.

[LJR14] Sangho Lee, Teresa Johnson und Easwaran Raman. “Feedback directed optimization of TCMalloc”. In: *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM. 2014, S. 3.

[LK98] Per-Åke Larson und Murali Krishnan. “Memory allocation for long-running server applications”. In: *ACM SIGPLAN Notices*. Bd. 34. 3. ACM. 1998, S. 176–185.

[Str12] Alexey Stroganov. 2012. URL: https://www.percona.com/blog/2012/07/05/impact-of-memory-allocators-on-mysql-performance/.

[Wil+95] Paul R. Wilson u. a. “Dynamic storage allocation: A survey and critical review”. In: *Memory Management*. Hrsg. von Henry G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, S. 1–116. ISBN: 978-3-540-45511-0.

[Yun+14] Heechul Yun u. a. “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE. 2014, S. 155–166.