Preprint from Proceedings of Echtzeit 2008, Aktuelle Anwendungen in Technik und Wirtschaft

Dynamische Verwaltung Virtuellen Speichers für Echtzeitsysteme

Martin Böhnert, Thorsten Zitterell und Christoph Scholl

Lehrstuhl für Betriebssysteme, Albert-Ludwigs-Universität Freiburg
{boehnert,tzittere,scholl}@informatik.uni-freiburg.de

Zusammenfassung. In der vorliegenden Arbeit beschreiben wir ein neues Konzept, das im Hinblick auf Echtzeitsysteme sowohl virtuelle Speicherverwaltung als auch dynamische Belegungen bzw. Freigaben von Speicher erlaubt. Das Verfahren zielt auf ein effizientes, von Anwendungsdaten unabhängiges Laufzeitverhalten ab und weist bei einer virtuellen Adressraumgröße n für die Operationen Allokation, Freigabe und Zugriff eine Laufzeitkomplexität von $O(\log \log n)$ auf (die Laufzeit ist insbesondere unabhängig von der Anzahl der virtuellen Seiten m der allokierten Region). Da die Größe des virtuellen Adressraums bei einer Architektur fest vorgeben ist, kann die Laufzeit somit als (quasi) konstant betrachtet werden.

1 Einleitung

Virtueller Speicher wird in modernen Betriebssystemen dazu benutzt, individuelle Adressräume für verschiedene Tasks bereitzustellen, Speicherschutzmechanismen zu realisieren und einer Fragmentierung des physikalischen Speichers bei dynamischen Speicherbelegungen und -freigaben entgegenzuwirken. Sowohl der virtuelle Adressraum als auch der physikalische Adressraum wird hierfür in einzelne Seiten vorgegebener Größe eingeteilt. Die Adressen werden jeweils aufgeteilt in eine Seitennummer und einen Offset innerhalb der zugehörigen Seite. Jede der virtuellen Seiten kann auf einen physikalischen Speicherbereich abgebildet werden, indem die jeweilige Zuordnung der virtuellen Seitennummer zur physikalischen Seitennummer in einer Seitentabelle eingetragen wird. Die Übersetzung virtueller Adressen in physikalische Adressen zur Laufzeit, also das Nachschlagen eines Eintrags in der Seitentabelle, wird meist mit hardwareseitiger Unterstützung durch eine *Memory Management Unit (MMU)* realisiert. Gängige Strukturen zur Organisation einer Seitentabelle sind beispielsweise direkte, hierarchische oder invertierte Tabellen [1].

Während sich die Seitentabelle nur um die Abbildung des virtuellen Speichers kümmert, sind für Verwaltung dynamischer Belegungen und Freigaben von Speicherseiten weitere Datenstrukturen und Algorithmen notwendig. Das Betriebssystem muss den Speicher so organisieren, dass freie Seiten bei Bedarf gefunden, sowie freigegebene Seiten wieder benutzt werden können. Gut vorhersagbare und schnelle Laufzeiten für beliebige Speicherbelegungen und -freigaben zu erzielen,

die nicht von den aktuellen Eingaben abhängen, stellt jedoch für die bekannten Implementierungen von General-Purpose-Systemen eine große Schwierigkeit dar. So würde eine explizite Freigabe einer Speicherregion über m virtuelle Seiten auch eine Freigabe der zugehörigen physikalischen Seiten notwendig machen. Dies würde ein Durchlaufen der Seitentabelle in $O(m)$ mit jeweiliger Freigabe der entsprechenden physikalischen Seiten bedeuten. Auch wenn auf eine explizite Freigabe verzichtet wird, um bei Neubelegungen von Speicher durch den aktuellen Prozess diesen Speicher einfach wiederverwenden zu können, müssen die entsprechenden Speicherseiten zumindest in $O(m)$ markiert werden.[1] Ähnliche Überlegungen gelten auch für Belegung eine Region von m Speicherseiten.

Im Allgemeinen lässt sich feststellen, dass die impliziten Mechanismen moderner General-Purpose-Betriebssysteme, wie z.B. bei Linux, auf eine niedrige *durchschnittliche* Laufzeit optimiert sind. Wiedergewinnung von freier Speicherseiten erfolgt meist erst, wenn der Speicher knapp wird. Obwohl mit dieser Vorgehensweise in der Regel eine hohe mittlere Effizienz erreicht wird, so ist ein wesentlicher Nachteil, dass sich die Laufzeiten bei hoher Speicherauslastung erhöhen. In sicherheits- und zeitkritischen Systemen wird deshalb üblicherweise auf die dynamische Verwaltung von virtuellem Speicher gänzlich verzichtet, da sowohl bei expliziten als auch impliziten Mechanismen das Worst-Case-Verhalten schwer abschätzbar ist.

Für Echtzeitsysteme wurde in [2,3] ein Allokator (TLSF) vorgestellt, der Speicherbelegungen und -freigaben unabhängig von den Anwendungsdaten in konstanter Zeit durchführt (bei fester virtueller Adressraumgröße). Dieser arbeitet jedoch nur direkt auf der Ebene des physikalischen Speichers, eine Lösung in Kombination mit virtueller Speicherverwaltung wurde nicht vorgestellt. Ebenfalls in Hinblick auf Echtzeitsysteme wurde in [4] ein Ansatz vorgestellt, der virtuelle Speicherregionen in Intervallen organisiert, die linear auf einen physikalischen Bereich abgebildet werden. Die hinzugefügten Einträge werden nach virtuellen Basisadressen sortiert abgelegt und es wird zur Laufzeit mit binärer Suche wieder darauf zugegriffen. Durch die lineare Abbildung virtueller Speicherregionen auf physikalische Speicherseiten ergibt sich hierbei aber das Problem einer mit der Laufzeit zunehmenden Fragmentierung des physikalischen Speichers trotz virtueller Speicherverwaltung. Weiterhin ist die Laufzeitkomplexität für Manipulationen nicht unabhängig von der Anzahl der Einträge in der Seitentabelle.

Mit unserer Arbeit wollen wir ein neuartiges Konzept zur dynamischen Verwaltung virtuellen Speichers vorstellen, welches bei einer virtuellen Adressraumgröße n für die Operationen Allokation, Freigabe und Zugriff eine Laufzeitkomplexität von $O(\log \log n)$ aufweist. Die Besonderheit des Verfahrens ist, dass die Laufzeit unabhängig ist von der Anzahl der virtuellen Seiten m der allokierten Region. Da die Größe des virtuellen Adressraums bei einer Architektur fest

[1] Ansonsten lässt sich der entsprechende physikalische Speicher entweder nicht wiedergewinnen oder man läuft bei Systemen mit Hintergrundspeicher und Seitenverdrängungsstrategien Gefahr, dass bei Speicherknappheit Speicherregionen auf die Festplatte geschrieben werden, die eigentlich schon freigegeben sind.

vorgeben ist, kann die Laufzeit somit als konstant betrachtet werden. Die Laufzeitkomplexität wird mit der Organisation der Seitentabelle auf Basis geschichteter Bäume [5,6] erreicht. Weiterhin ist garantiert, dass der explizit freigegebene Speicher direkt wieder im System zur Verfügung steht.

Mit der virtuellen Speicherverwaltung allein lassen sich nur Allokierungen durchführen, deren Bereich sich über ein Vielfaches der Seitengröße ersteckt. Sie bildet damit die untere Schicht der Speicherverwaltung. Um auch für Echtzeitsysteme Belegungen beliebiger Größe zu ermöglichen, wird in dieser Arbeit gezeigt, wie der TLSF-Allokator so angepasst werden kann, dass er auf unserer virtuellen Speicherverwaltung aufbaut und gleichzeitig die Laufzeitkomplexität von $O(\log \log n)$ aufrechterhalten wird. Dieser Allokator hat nur die Sicht auf den virtuellen Adressraum und bildet die obere Ebene der Speicherverwaltung.

In unserem Ansatz verzichten wir bewusst auf eine Speicherhierarchie mit Hintergrundspeicher, da durch Verdrängung von physikalischen Speicherseiten auf Festplatte in Echtzeitanwendungen schwer abschätzbare Verzögerungen entstehen würden. Stattdessen gehen wir davon aus, dass im System genügend Hauptspeicher zur Verfügung gestellt wird, um Obergrenzen hinsichtlich des Speicherverbrauchs einzuhalten. Obergrenzen für den Speicherbedarf lassen sich ähnlich wie bei der Laufzeitanalyse statisch ermitteln. Die Analysen werden aufgrund unserer expliziten Speicherfreigabemechanismen wesentlich vereinfacht.

Die Arbeit ist wie folgt gegliedert: In Kapitel 2 wird unsere Verwaltung virtuellen Speichers im Zusammenspiel mit dynamischen Speicherbelegungen und -freigaben detailliert beschrieben. Hier machen wir zunächst die Annahme, dass die allokierten Speicherbereiche ganzzahlige Vielfache von Seitengrößen sind. In Kapitel 3 werden dann die Erweiterungen für Speicherbereiche beliebiger Größe beschrieben. Erste experimentelle Ergebnisse, die die Echtzeitfähigkeit des Ansatzes untermauern, sind in Kapitel 4 zu finden. Kapitel 5 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick auf künftige Arbeiten ab.

2 Dynamische Verwaltung virtuellen Speichers

Die beteiligten Komponenten bei der virtuellen Speicherverwaltung sind in der Regel die Seitentabelle und das Betriebssystem. Für die unterschiedliche Programme auf dem System mit jeweils individuellen Adressräumen A_i der Größe n erfolgt die Vergabe des virtuellen Speichers in der Regel völlig transparent. Soll beispielsweise ein neues Programm gestartet werden, so stellt das Betriebssystem einen neuen virtuellen Adressraum bereit, kopiert das Programm an eine definierte Stelle und startet es dort. Benötigt das Programm zur Laufzeit weiteren Speicher, so bildet das Betriebssystem zusätzliche virtuelle Seiten auf physikalische Adressen ab.

Im Folgenden gehen wir ebenfalls davon aus, dass das Betriebssystem intern Operationen zur Allokierung und Freigabe von virtuellem Speicher bereitstellt. Der Aufruf `vmalloc`(A_i, a, x) reserviert hierbei eine beliebige Anzahl x aufeinanderfolgender, virtueller Seiten ab der Seitennummer a innerhalb des Adressraumes A_i. Die Operation `vfree`(A_i, a) gibt diesen Speicherbereich komplett wieder

frei. Des Weiteren gibt es eine Routine `vmap`(A_i, b), welche bei einem Seitenfehler aufgerufen wird und die Seite b innerhalb eines virtuellen Speicherbereichs einer physikalischen Seite zuordnet. Bevor wir auf die Arbeitsweise dieser Operationen in Abschnitt 2.2 genauer eingehen, wollen wir zunächst einen Überblick über wichtige Komponenten und die verwendeten Datenstrukturen geben, die eine Laufzeitkomplexität von $O(\log \log n)$ ermöglichen.

2.1 Verwendete Datenstrukturen

Die Aufgabe einer herkömmlichen Seitentabelle ist es, die Zuordnung virtueller Seiten zu physikalischen Seiten bereitzustellen. Neben der Abbildung selbst enthält ein Eintrag meist noch Zugriffs- und Gültigkeitsbits. Unsere virtuelle Speicherverwaltung basiert auf einer einstufigen Tabelle mit direkter Abbildung. Um in konstanter Zeit sämtliche physikalische Seiten freigeben zu können, die für eine virtuelle, mittels `vmalloc` allokierte Speicherregion vergeben wurden, müssen diese an einer für die Region zentralen Stelle, dem *Virtual Region Header (VRH)*, gesammelt werden. Um weiterhin sämtliche Zuordnungen von physikalischen Seiten zu virtuellen Seiten auf einen Schlag ungültig machen zu können (und um bei Allokation auch nicht in Linearzeit initialisieren zu müssen), arbeiten wir mit einem auf initialisierungsfreien Arrays [6, S. 289–290] basierenden *Pages Validity Array (PVA)*, das über den Virtual Region Header (VRH) zugreifbar ist. Um bei Zugriff auf eine virtuelle Adresse die Gültigkeit des Seitentabelleneintrags überprüfen zu können und um bei Seitenfehlern neu belegte physikalische Seiten sammeln zu können, muss somit der VRH der zugehörigen Speicherregion bestimmt werden. Dies wird durch den *Virtual Regions Tree (VRT)* ermöglicht, der es erlaubt, beim Zugriff auf eine beliebige virtuelle Adresse den zugehörigen VRH in $O(\log \log n)$ zu finden. Freie oder einem VRH zugehörige, physikalische Seiten werden jeweils durch verkettete Listen zusammengefasst. Im Folgenden werden wir diese Datenstrukturen (Abbildung 1) genauer erläutern:

- *Seitentabelle (ST):* Die Seitentabelle wird genutzt, um virtuelle Adressen in physikalische zu übersetzen. Dies geschieht immer für Speicherseiten fester Größe.
- *Pages Validity Array (PVA):* Das Pages Validity Array ist eine Datenstruktur, die auf einem initialisierungsfreien Array beruht. Ein initialisierungsfreies Array der Größe k lässt sich realisieren, indem man zusätzlich zu einem Feld der Größe k einen Stapel mit Maximalgröße k und ein Hilfsfeld der Größe k in geeigneter Weise verwendet [6, S. 289–290]. Mit dem PVA lässt sich dann (ohne vorherige Initialisierung) in konstanter Zeit überprüfen, ob Einträge in der Seitentabelle gültig sind oder nicht. Ein wesentlicher Vorteil des PVA in Bezug auf Echtzeiteigenschaften ist die Möglichkeit, eine komplette Speicherregion (mit einer beliebigen Anzahl von Einträgen in der Seitentabelle) in konstanter Zeit als ungültig zu erklären, indem nur der Stapelzeiger neu gesetzt wird. Die entsprechenden Datenstrukturen können kompakt abgespeichert werden, indem der Zeiger auf den Stapel im VRH abgelegt wird und die Elemente des Hilfsfeldes in die Repräsentation der physikalischen Speicherseiten verlagert werden.

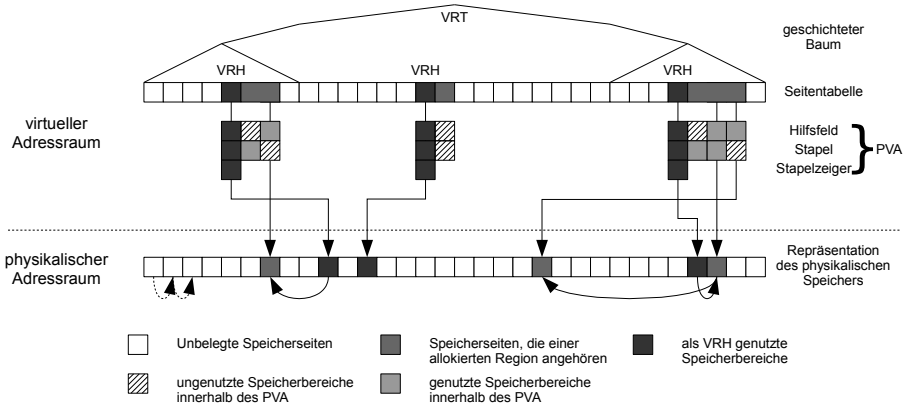

Abb. 1. Verwendete Datenstrukturen der virtuellen Speicherverwaltung

- *Virtual Region Header (VRH):* Jeder Virtual Region Header repräsentiert eine allokierte Speicherregion und zeigt auf den Kopf der Liste aller physikalischen Seiten, die mittels der `vmap()`-Routine dieser Region zugeordnet wurden. Bei der Freigabe einer virtuellen Speicherregion können diese physikalischen Seiten in konstanter Zeit dem Pool freier Seiten wieder zugeführt werden. Der VRH ist immer der ersten Seite einer virtuellen Speicherregion zugeordnet. Da der VRH immer eine Speicherregion repräsentiert, kann er benutzt werden, um Zugriffsrechte anzuzeigen.
- *Virtual Regions Tree (VRT):* Der Virtual Region Tree dient dem Zweck, bei einem Zugriff auf eine beliebige virtuelle Speicherseite den Anfang der jeweiligen Region und damit auch den zugehörigen VRH zu ermitteln. Die Organisation der Regionen und die nötigen Funktionen, um die Anfänge der Regionen zu markieren, zu finden und wieder zu löschen, sind über die `union()`-, `split()`- und `find()`-Operationen eines geschichteten Baumes implementiert. Die Funktionen haben eine Komplexität von $O(\log \log n)$ über die Anzahl n der Seiten des virtuellen Adressraumes, wie in [5] und [6] zu sehen ist.
- *Physikalische Seiten:* Die Seiten des physikalischen Adressraumes werden durch ein lineares Feld repräsentiert, wobei jedes Feldelement einen Zeiger besitzt, um physikalische Seiten in einer einfachen Liste zu verketten. Zusätzlich enthält jedes Element einen Zeiger auf das letzte Element der Liste. Bei der Systeminitialisierung werden alle physikalischen Seiten zu einer Liste freier Seiten verkettet. Falls zur Laufzeit eine freie physikalische Seite bei `vmap()` benötigt wird, wird diese aus der Liste der freien Seiten entfernt und an die benutzten Seiten einer Speicherregion, welche über den VRH ermittelt wird, angefügt. Bei Freigabe einer Speicherregion mittels `vfree()` werden die gesammelten physikalischen Seiten wieder mit der Liste freier Seiten verknüpft.

2.2 Virtuelle Speicherverwaltung

Der Schwerpunkt unserer Arbeit liegt auf einem Konzept der virtuellen Speicherverwaltung, das es erlaubt, die Operationen `vmalloc()`, `vmap()` und `vfree()` unabhängig von der Größe des angeforderten bzw. freigegebenen Speichers in der Zeit $O(\log \log n)$ über die virtuelle Adressraumgröße n durchzuführen. Die verwendeten Datenstrukturen wurden in Abschnitt 2.1 erläutert. Im Folgenden werden wir auf die Arbeitsweise der drei Operationen näher eingehen.

Allokation von Seiten: Eine virtuelle Speicherregion mit der Basisadresse a und der Größe bzw. Anzahl der Seiten x wird innerhalb eines Adressraumes A_i mittels `vmalloc`(A_i, a, x) allokiert. Die Allokation besteht aus drei Schritten. Zuerst wird eine einzelne physikalische aus der Liste der verfügbaren freien Seiten genommen, welche den VRH für die Speicherregion repräsentieren soll. Die erste virtuelle Seite a wird durch einen Eintrag in der Seitentabelle auf diese physikalischen Seite abgebildet. Als nächstes wird dieser Seiteneintrag im PVA als gültig markiert. Als letzte Operation muss noch der Anfang der Speicherregion im VRT markiert werden. Der letzte Schritt sorgt dafür, dass Zugriffe auf andere virtuelle Seiten innerhalb des gleichen Bereichs dem zugehörigen Virtual Region Header zugeordnet werden können. Weitere Speicherseiten innerhalb des virtuellen Bereichs werden erst dann auf eine physikalische Seite abgebildet, wenn der erste Zugriff darauf stattfindet. In diesem Fall zeigt das PVA an, dass der Seiteneintrag ungültig ist und es wird ein Seitenfehler ausgelöst. Dieser veranlasst die Zuordnung einer weiteren Speicherseite für die virtuelle Speicherregion.

Speicherzugriffe und Seitenfehlerbehandlung: Wir gehen davon aus, dass die Übersetzung von virtuellen in physikalische Adressen transparent mit Hilfe einer *Extended Memory Management Unit (ExtMMU)* in Hardware durchgeführt wird. Die ExtMMU prüft beim Zugriff auf eine Seite zuerst die Gültigkeit des Seitentabelleneintrags mit dem Pages Validity Array, auf welches über den VRH zugegriffen wird. Falls eine virtuelle Speicherseite ungültig ist, wird ein Seitenfehler ausgelöst. In allen anderen Fällen wird die virtuelle Adresse direkt in die zugehörige physikalische Speicheradresse übersetzt.

In Falle eines Seitenfehlers muss nun die ungültige Seite b mittels `vmap`(A_i, b) auf eine physikalische Seite abgebildet werden. Dafür wird wieder eine physikalische Seite aus der Liste freier Seiten entnommen und mit den bereits benutzten im VRH verkettet. Am Schluss wird ein entsprechender Eintrag in die Seitentabelle geschrieben und die entsprechende Seite im PVA als gültig markiert.

Freigabe von virtuellen Seiten: Die Freigabe einer virtuellen Speicherregion mit Basisadresse a erfolgt mit der Operation `vfree`(A_i, a). Diese gibt den zur Laufzeit allokierten Speicher einer Region wieder in drei Schritten frei: Als erstes wird der Bereich im PVA in konstanter Zeit als ungültig markiert. Danach werden die benutzten und im Virtual Region Header verketteten physikalischen Speicherseiten wieder an die Liste der freien Seiten angehängt. Im letzten Schritt wird die Anfangsmarkierung der Speicherregion aus dem Virtual Regions Tree gelöscht.

Durch das explizite Allokations- und Freigabeschema wird somit kein Speicherplatz verschwendet, da nur tatsächlich benötige Seiten auf physikalische Seiten abgebildet werden und die Seiten nach einer Deallokation sofort von anderen Programmen wieder verwendet werden können.

2.3 Kompakte Repräsentation der Datenstrukturen

In diesem Abschnitt wollen wir zeigen, dass — obwohl unsere Seitenverwaltung auf zusätzlichen Datenstrukturen basiert — die Daten von VRT und PVA auf eine kompakte Art und Weise gespeichert werden können. Die dem Virtual Regions Tree zu Grunde liegende Datenstruktur ist ein geschichteter Baum. Die gegebene Implementierung nach [6] geht von einer Liste mit dynamischer Größe aus, die mit `union()`-, `split()`- und `find()`-Operationen in Intervalle unterteilt werden kann. Die einzelnen Komponenten der Liste und des geschichteten Baumes werden hier über mehrere explizite Zeiger (z.B. Vorgänger- und Nachfolgerbeziehungen, Verweise auf übergeordnete Baumstrukturen, usw.) miteinander verknüpft. Da bei unserer Speicherverwaltung die virtuelle Adressraumgröße fest vorgegeben ist, sind die Beziehungen zwischen den einzelnen Komponenten implizit gegeben und es sind weitreichende Optimierungen des Speicherbedarfs möglich. So müssen beispielsweise die einzelnen Zeiger nicht mehr abgespeichert werden und einzelne virtuelle Seiten werden nicht mehr durch Listenelemente sondern kompakt durch Bitfelder repräsentiert.

3 Speicherverwaltung für beliebige Speichergrößen

Die virtuelle Speicherverwaltung bildet die untere Schicht der Speicherverwaltungshierarchie innerhalb eines Betriebssystems und erlaubt es, Speicherbereiche mit Größen vom ganzzahligen Vielfachen einer virtuellen Speicherseite zu allokieren. Auf höherer Ebene bzw. innerhalb der virtuellen Adressräume stellen Betriebssysteme meist noch weitere Allokatoren bereit, um auch die Belegung und Freigabe beliebiger Speichergrößen (z.B. wenige Bytes) mittels der Operationen `malloc()` und `free()` zu ermöglichen. Die virtuelle Speicherverwaltung erfolgt aus Sicht der Anwendung transparent – benötigte virtuelle Seiten werden vom Betriebssystem mittels `vmalloc()` allokiert und umgekehrt mit `vfree()` wieder freigegeben. Im Folgenden wollen wir zeigen, wie wir den TLSF-Allokator [2,3] angepasst haben, so dass die Gesamtkomplexität von $O(\log \log n)$ erhalten bleibt.

TLSF selbst besitzt eine Laufzeitkomplexität von $O(1)$, um Speicherblöcke zu allokieren und freizugeben, erwartet aber einen existierenden und kontinuierlichen Adressbereich, um seine *Bounding Tags* zu speichern, mittels derer die einzelnen Speicherstücke verwaltet werden. Auf Grund dieser Eigenschaften eignet sich TLSF in der ursprünglichen Form für den direkten Einsatz auf physikalischem Speicher bei gleichzeitigem Verzicht auf individuelle, virtuelle Adressräume für die Programme. Prinzipiell kann TLSF auch auf virtuellem

Speicher arbeiten, doch dann muss dafür gesorgt werden, dass vor der Laufzeit genügend virtueller Speicher auf physikalische Seiten abgebildet wurde, was gegebenenfalls zu einer Verschwendung führt. Unser Ziel war es deshalb TLSF soweit anzupassen, dass virtuelle Seiten *nur bei Bedarf* auf physikalische Seiten abgebildet werden. Dies betrifft jene Seiten, die für allokierte Bereiche und für die Bounding Tags tatsächlich benötigt werden.

Die Frage, ob virtuelle Seiten auf physikalische Seiten abgebildet werden müssen, stellt sich immer, wenn Bounding Tags durch die TLSF-Funktionen `split()` und `merge()` erstellt oder gelöscht werden. Hier kann aufgrund der virtuellen Adresse und der Länge eines Speicherbereichs, sowie seiner direkten Umgebung (also virtuell vorangehende und nachfolgende Speicherbereiche) entschieden werden, ob an der neuen Startposition bereits eine Speicherseite zugeordnet ist oder eine neue Zuordnung erstellt werden muss. Umgekehrt ist es auf gleichem Wege möglich, herauszufinden, ob eine bestehende Zuordnung gelöscht werden kann. Das explizite Freigabeschema garantiert weiterhin, dass nicht mehr belegte Speicherseiten sofort an das System wieder zurückgegeben werden.

4 Experimente

Zur Evaluierung unseres Ansatzes wurde eine Bibliothek in C implementiert, welche die virtuelle Speicherverwaltung simuliert und darauf aufbauend die Operationen des angepassten TLSF-Allokators nach außen zur Verfügung stellt. Ein Beschreibung der Simulationsumgebung erfolgt in Abschnitt 4.1. Erste Ergebnisse, die das Laufzeitverhalten unseres Verfahrens zeigen, werden in Abschnitt 4.2 vorgestellt.

4.1 Simulationsumgebung

Die untere Schicht der Simulationsumgebung implementiert die virtuelle Speicherverwaltung. Da bisher noch keine reale Hardware implementiert wurde, welche die gewünschte Funktionsweise der ExtMMU besitzt, haben wir hierfür das Nachschlagen der Einträge in der Seitentabelle und deren Gültigkeitsprüfung über den Virtual Region Tree und dem Pages Validity Array in Software simuliert. Der notwendige Speicher zur Repräsentation des physikalischen Adressraums wird vom Hostsystem bereitgestellt. Die darauf aufbauende, obere Schicht bildet der von uns angepasste TLSF-Allokator mit den Operationen `malloc()` und `free()`.

Zur exakten Messung des Zeitverhaltens haben wir zu dem Werkzeug *callgrind* [7] (Teil der *valgrind*-Suite [8]) gegriffen. Das Tool misst hierbei die Anzahl der Instruktionen, die jeweils für die Operationen `malloc()` und `free()` und damit auch für `vmalloc()` und `vfree()` notwendig sind. Ein vergleichbarer Zeitaufwand würde auch bei einer Integration unseres Verfahrens in einem Betriebssystem (mit Hardwareunterstützung durch die ExtMMU) anfallen.

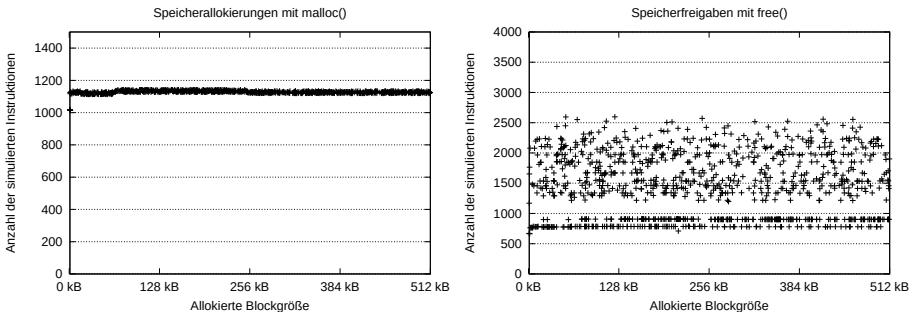

Abb. 2. Die Abbildungen zeigen die Kosten für Aufrufe von `malloc()` (links) und `free()` (rechts). Insgesamt wurden 1000 verschachtelte Allokationen und Freigaben mit Speicherblockgrößen zwischen 16 Bytes und 512 kBytes simuliert.

4.2 Ergebnisse

Die ersten Experimente für diese Arbeit wurden mit 1000 verschachtelten Allokationen und Freigaben von Speicherstücken zwischen 16 Bytes und 512 kBytes durchgeführt. In beiden Diagrammen der Abbildung 2 stellt die x-Achse jeweils die gegebene Blockgröße dar, während die y-Achse den dazugehörigen Zeitaufwand in Anzahl der Instruktionen widerspiegelt. Die Ergebnisse bestätigen unsere Aussage, dass der Zeitaufwand sowohl der Aufrufe von `malloc()` (linkes Diagramm) als auch der Aufrufe von `free()` (rechtes Diagramm) unabhängig von der Größe der gewählten Speicherstücke ist.

Das linke Diagramm für `malloc()` zeigt eine obere Grenze der Laufzeitkosten über den gesamten Bereich der Allokationsgrößen auf. Für Allokationen mit einer Speichergröße unterhalb der Seitengröße muss nicht immer eine zusätzliche physikalische Seite einer virtuellen Speicherseite zugeordnet werden. Dies ist in den Messungen mit geringeren Laufzeitkosten zu sehen. Im rechten Diagramm für `free()` erkennt man ebenfalls die Unabhängigkeit des Zeitaufwandes von der allokierten Blockgröße. Da die Freigabe durch ihren expliziten Mechanismus komplexer ist, bildet sich ein Band mit gestreuten Werten heraus. Aber auch hier sind die Laufzeitkosten durch eine obere und untere Schranke eindeutig begrenzt.

Aufgrund des frühen Stadiums der Bibliothek und der Messungen lassen sich zu diesem Zeitpunkt nur qualitative Aussagen über unser Verfahren treffen. Um quantitative Aussagen treffen zu können, wäre eine reale Umgebung mit ihren architekturspezifischen Anpassungen und Optimierungen nötig.

5 Fazit und weiterführende Arbeiten

In unserer Arbeit haben wir ein neuartiges Konzept aufgezeigt, um virtuellen Arbeitsspeicher dynamisch zu verwalten. Unser Verfahren erreicht eine Laufzeitkomplexität von $O(\log \log n)$ für Allokationen, Zugriffe und Deallokationen

über einen virtuellen Adressraum mit n Seiten. Weiterhin haben wir gezeigt, wie der TLSF-Allokator erweitert wurde, um Speicherstücke beliebiger Größe zu verwalten, ohne die asymptotische Komplexität der Operationen zu verändern.

Da der Zeitaufwand unabhängig von der Größe des allokierten Speichers ist und die Größe des virtuellen Adressraums bei einer spezifischen Architektur fest vorliegt, lässt sich der Zeitaufwand in der Anwendung für alle Operationen der Speicherverwaltung als konstant abschätzen. Durch die Verwendung eines expliziten Freigabemechanismus wird kein Speicherplatz verschwendet, schlecht vorhersagbarer zeitlicher Overhead durch implizite Freigabemechanismen wurde vermieden. Somit wurde für Echtzeitsysteme ein großer Schritt in Richtung besserer Vorhersagbarkeit der Laufzeit und der maximalen Speicherauslastung erzielt.

In einem nächsten Schritt planen wir, das Konzept anhand einer (prototypischen) Hardwarerealisierung der ExtMMU und einer Integration in ein Echtzeitbetriebssystem zu evaluieren. Hierbei ist eine Aufteilung der Tasks vorgesehen einerseits in harte Echtzeittasks mit Speicherverwaltung nach den hier vorgestellten Prinzipien und andererseits in Tasks ohne harte Echtzeitbedingungen, die mit klassischen Speicherverwaltungsverfahren behandelt werden sollen.

Literaturverzeichnis

1. B. Jacob, T. Mudge. Virtual memory: Issues of implementation. *Computer*, 31(6):33–43, 1998.
2. M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society
3. M. Masmano, I. Ripoll, J. Real et al. Implementation of a constant-time dynamic storage allocator. *Software: Practice and Experience*, 38:995–1026, August 2008
4. Xiangrong Zhou and Peter Petrov. The interval page table: virtual memory support in real-time and memory-constrained embedded systems. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 294–299, New York, USA, 2007. ACM.
5. P. van Emde, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, December 1976.
6. Kurt Mehlhorn. *Data Structures and Algorithms 1*: Sorting and Searching, volume 1 of *Monographs in Theoretical Computer Science*. An *EATCS Series*. Springer, 1984.
7. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.
8. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, USA, 2007. ACM.