Energieeffiziente Hauptspeicherkompression

Studienarbeit im Fach Informatik

vorgelegt von

Holger Wunderlich

geboren am 24. April 1980 in Kemnath

Institut für Informatik
Lehrstuhl für verteilte Systeme und Betriebssysteme
Friedrich Alexander Universität Erlangen-Nürnberg

Betreuer:	Dipl.-Inf. Andreas Weißel
	Dr. Ing. Frank Bellosa
	Prof. Dr. Wolfgang Schröder-Preikschat
Beginn:	21. Januar 2004
Abgabe:	20. Oktober 2004

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.

Erlangen, den 20. Oktober 2004

Holger Wunderlich

Kurzfassung

Trotz der immer besseren Technologien ist neben der Leistung auch der Energieverbrauch mobiler Geräte (z. B. Laptop, PDA) gestiegen. Dies resultiert unter anderem daraus, dass das verfügbare Energiesparpotential nicht optimal genutzt wird.

Bislang wurden nur die Leistung der CPU dynamisch an die Anforderung angepasst und andere Geräte, wie z. B. Festplatten und Netzwerkadapter möglichst sparsam betrieben (Abschaltung, wenn nicht benötigt).

Völlig außer Acht gelassen wurde jedoch der Hauptspeicher. Dessen Größe und der damit verbundene Energiebedarf hat sich in den letzten Jahren aufgrund niedriger Preise und erhöhten Anforderungen seitens der Software vervielfacht.

In dieser Arbeit wird untersucht, ob der Inhalt des Hauptspeichers komprimierbar ist und ob die durch Abschaltung der somit ungenutzten Teile des Hauptspeichers eingesparte Energie den Energieaufwand für Kompression und Dekompression übersteigt.

Hierbei richtet sich das Augenmerk nicht auf den Hauptspeicher als Ganzes sondern auf die Speicherbereiche der einzelnen Prozesse. Diese Bereiche werden wiederum in Daten- und Codesegmente aufgeteilt. Die benutzten Prozesse stammen aus den Anwendungsbereichen Internet, Textverarbeitung, Bildbearbeitung und Audio. Die Segmente werden mit den Algorithmen LZRW1, LZO, WK4x4, WKdm, WHdm, WHsd komprimiert und dekomprimiert.

Dabei werden die Kompressionsrate und der Energie- und Zeitbedarf bei Kompression und Dekompression bestimmt.

Diese Arbeit zeigt, dass durch die Kompression des Hauptspeichers bei einer Kompressionsgewschwindigkeit von mindestens 2 MB / s (bei 400 MHz) bis zu 75 % der ursprünglichen Größe eingespart werden kann

Des Weiteren wird gezeigt, dass die Unterscheidung in Code- und Datensegment keine Auswirkung auf die Auswahl des besten Algorithmus für die Anwendung hat.

Die Auswertung ergibt, dass ohne konkretes Anwendungsszenario zwei Algorithmen optimal erscheinen, von denen der eine bei der Kompressionsrate und der andere bei der Kompressionsgeschwindigkeit besser abschneidet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Leistungsfähigkeit mobiler Rechner (z. B. Laptops und PDAs) stieg in den letzten Jahren stark an. Dies ist vor allem auf immer schnellere CPUs und größere und schnellere Hauptspeicher zurückzuführen. Der Wirkungsgrad der Batterien konnte jedoch nur wenig gesteigert werden. Die durchschnittliche Laufzeit der Geräte konnte somit nur durch größere Akkus (mehr Gewicht) konstant gehalten werden.

Aufgrund dieser Problematik wurden unter anderem neue Prozessorgenerationen (Pentium M, Transmeta Crusoe und Efficeon, AMD (Duron) Mobile) entwickelt. Für einige weitere Komponenten (z. B. Festplatten und Netzwerkadapter) wurden Energiesparmodi implementiert, die durch intelligente Nutzung (z. B. Interaktion mit dem Betriebssystem statt statischer Regeln in der Firmware) den Stromverbrauch drastisch senken können, aber trotzdem nur minimale (u. U. auch keine) Auswirkungen auf die Performance des Rechners haben.

Was jedoch nur wenig beachtet wurde, ist der Einfluss des Hauptspeichers auf den Energieverbrauch. Dieser macht aufgrund der sehr sparsamen CPUs (wenige Watt) einen immer größeren Teil des Gesamtverbrauchs aus. Dieser konstante Verbrauch ist vor allem darauf zurückzuführen, dass der Hauptspeicher fortwährend Energie benötigt, um die gespeicherten Daten zu erhalten. Ebenso ist eine Abschaltung von Teilen des Hauptspeichers nicht möglich, da auch dann die enthaltenen Daten verloren gingen.

Motivation

Diese Arbeit beschäftigt sich mit der Kompression von Hauptspeicherdaten. Die in Folge dessen frei werdenden Teile des RAMs könnten somit abgeschaltet werden. Alternativ könnte auch weniger Hauptspeicher in den Rechner eingebaut werden.

Es werden verschiedene Kompressionsalgorithmen hinsichtlich ihres Energie- und Zeitbedarfs bei Kompression und Dekompression untersucht. Außerdem wird die Kompressionsrate betrachtet.

Die zur Kompression verwendeten Daten stammen aus dem Hauptspeicher. Dabei wird der Hauptspeicher aber nicht als große Einheit betrachtet sondern in verschiedene Bereiche eingeteilt. Die erste Einteilung erfolgt in die Speicherbereiche der Prozesse, die dann wiederum in Datensegmente und Codesegmente unterteilt werden.

Sinn dieser Aufteilung ist es, zu untersuchen, ob die Struktur der Daten- und Codesegmente so unterschiedlich ist, dass es für jeden Segmenttyp einen speziellen Kompressions-

algorithmus mit optimalem Ergebnis gibt.

Zur Anwendung kommen dabei die Algorithmen LZRW1, LZO, WKdm, WK4x4, WHdm, WHsd16 und WHsd256.

Die Evaluation der gesammelten Daten zeigt, ob es möglich ist, durch Hauptspeicherkompression Energie zu sparen und welcher Aufwand (Zeit und Energie) dabei zusätzlich anfällt.

Die Ergebnisse dieser Arbeit könnten dann z. B. dazu verwendet werden, um zukünftig direkt im Kernel bei Prozessumschaltung den Speicher des deaktivierten Prozesses zu komprimieren und den Speicher des aktivierten Prozesses zu dekomprimieren.

So könnten, um Energie zu sparen, die nicht benötigten Teile des Hauptspeichers deaktiviert werden oder von vorneherein weniger RAM in die Geräte eingebaut werden.

Struktur

Nach einer Übersicht ähnlicher Arbeiten zum Thema folgt in Kapitel 3 eine Beschreibung der Speicherverwaltung und die Erklärung der Datensegmente und der Codesegmente.

In Kapitel 4 werden die verwendeten Kompressionsalgorithmen vorgestellt, sowie die Testdaten und die Testumgebung beschrieben.

Die Testresultate werden in Kapitel 5 präsentiert und ausgewertet.

Als letztes folgt ein kurzer Ausblick auf weitere Möglichkeiten, mit Hilfe der Hauptspeicherkompression Energie zu sparen.

Kapitel 2

Verwandte Arbeiten und Vorarbeiten

Die meisten Arbeiten, die sich bisher mit Hauptspeicherkompression beschäftigt haben, verwenden diesen als Puffer, um die Festplattenzugriffe zu beschleunigen. Dadurch kann dann aber auch die Festplatte länger im Ruhezustand bleiben, da die Zugriffe gruppiert werden und der Gesamtenergieverbrauch sinkt.

Ein anderer Ansatz ist die Kompression des Hauptspeichers über dedizierte Hardware wie er in [TFR+01] vorgestellt wird.

2.1 Kompression zur Verringerung der Festplattenzugriffe

Dies wird durch die Einführung einer neuen Ebene in der Speicherhierarchie erreicht.

Als einer der Ersten beschrieb Douglis in [Dou93] wie der Performance-Unterschied zwischen Hauptspeicher und Festplatte vermindert werden kann. Dabei setzte er bei der Ein- und Auslagerung von Seiten in bzw. aus dem Hauptspeicher an. Soll eine Seite aus dem Speicher ausgelagert werden, wird diese nicht direkt auf Festplatte geschrieben sondern in einem Teil des Hauptspeichers, dem *compression cache*, in komprimierter Form gespeichert. Erst wenn dieser Bereich voll ist werden die ältesten Seiten auf Festplatte ausgelagert. Soll nun eine „ausgelagerte" Seite wieder eingelagert werden, so kann diese aus dem *compression cache* entnommen und dekomprimiert werden.

Douglis kam jedoch zu dem Resultat, dass sich zwar die Anzahl der I/O-Operationen verringert, sich bei normalen Anwendungen jedoch das Laufzeitverhalten verschlechtert. Jedoch erkannte er, dass dies an der CPU-Geschwindigkeit liegt und dass Kompression durchaus sinnvoll sein kann, wenn die CPU-Geschwindigkeiten weiter steigen.

In [CCB99] wird ein ähnlicher Ansatz untersucht. Wieder wird eine Art *compression cache* verwendet. Im Unterschied zu Douglis werden jedoch die auf Festplatte ausgelagerten Seiten auch komprimiert gespeichert, was zu einer Vergrößerung des Swap-Speichers führt, ohne dass mehr Festplattenplatz belegt wird. Zusätzlich werden die Ein- und Auslagerungen gruppiert, also immer mehrere Seiten ein-/ausgelagert. Dies hat den Vorteil, dass die Fixkosten des Festplattenzugriffs (Such-/Positionierungs-Latenzen) nur einmal und nicht pro Seite anfallen.

Kaplan (und Wilson) verfolgen in [Kap99, WKS99] den Ansatz von Douglis weiter und verwenden unter anderem spezielle, auf das Problem zugeschnittene Kompressionsalgorith-

men. Zusätzlich diskutieren sie den Einsatz anderer Verdrängungsstrategien, da die normalerweise verwendete (LRU) bei einigen Programmabläufen sehr schlecht abschneidet.

Basierend auf dieser Arbeit wurde in [dCdLS03] eine Implementierung für Linux entwickelt. Dabei wurde der Ansatz mit dem *compression cache* weiter verbessert, indem der komprimierte Speicherbereich dynamisch an die Gegebenheiten angepasst wird und nicht mehr wie zuvor eine feste Größe besaß.

2.2 Kompression des CPU-Caches

Ein anderer Ansatz zur Reduktion des Energiebedarfs wird in „Frequent Value Compression in Data Caches" [YZG00] vorgestellt. Allerdings wird hier nicht der Hauptspeicher sondern der CPU-Cache betrachtet. Dabei wird der so genannte *compression cache* eingeführt, in diesem Fall ein First-Level-Cache, der pro Cache-Line entweder einen unkomprimierten oder aber zwei komprimierte Werte speichert.

Durch ein neu eingeführtes Kompressionsschema wird dabei auch die Möglichkeit des wahlfreien Zugriffs (random access) auf die Daten erhalten.

Durch die Kompression können im Cache mehr Daten gespeichert werden, was zu einer Reduktion der Cache-Miss-Rate führt, also die Datentransfers zwischen First-Level-Cache und dem langsameren Second-Level-Cache, bzw. RAM reduziert.

Dieser Ansatz wurde in [KAM02] wieder aufgegriffen und weiter verbessert. Die Neuerung bei dieser Arbeit ist, dass zum Lesen eines komprimierten Wertes nur die obere bzw. untere Hälfte einer Cache-Line aktiviert werden muss.

Dies wird erreicht durch ein Kompressionsverfahren namens *sign compression*. Dabei wird ausgenutzt, dass viele Werte sehr klein sind, also nur die unteren 16 bzw. 8 Bit verwendet werden. Die oberen 16 respektive 24 Bit sind dann entweder nur Nullen oder nur Einsen. Diese Werte werden bei *sign compression* durch ein einziges Bit dargestellt (0 wenn alles Nullen und 1 wenn alles Einsen).

Einen direkten Nutzen, z. B. Abschalten des leeren Hauptspeichers, gab es jedoch lange Zeit nicht, da unter anderem die Hardware keine Abschaltung unterstützte.

2.3 Abschalten des Hauptspeichers

Dies wurde erst mit der Einführung des RDRAM von RAMBUS möglich.

2.3.1 Neue Hardware

Bei SD RAM und DDR RAM sind die Bänke parallel angeordnet, um eine hohe Geschwindigkeit zu erreichen. Bei einem Speicherzugriff werden alle Bänke angesprochen. Die Daten werden in eine zufällig ausgewählte Bank geschrieben, was zu einem stark fragmentierten Speicher führt. Das bedeutet jedoch auch eine niedrige Zugriffszeit schon bei geringen Frequenzen bedeutet.

Bei RDRAM sind die Bänke in einer Reihe angeordnet (siehe Abb. 2.1). Die Geschwindigkeit wird hierbei durch eine hohe Frequenz erreicht. Bei einem Speicherzugriff wird nur

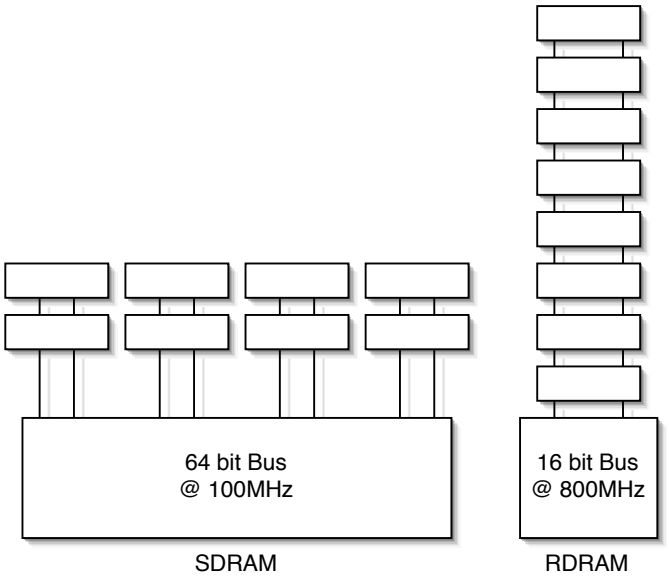

Abbildung 2.1: Blockdiagramm von SDRAM und RDRAM

die benötigte Bank angesprochen. Da die Hardware jede Bank unabhängig von den anderen in vier unterschiedliche Energiesparmodi versetzen kann, ist es möglich, nur die benötigte Bank zu aktivieren und die anderen im Energiesparmodus zu belassen.

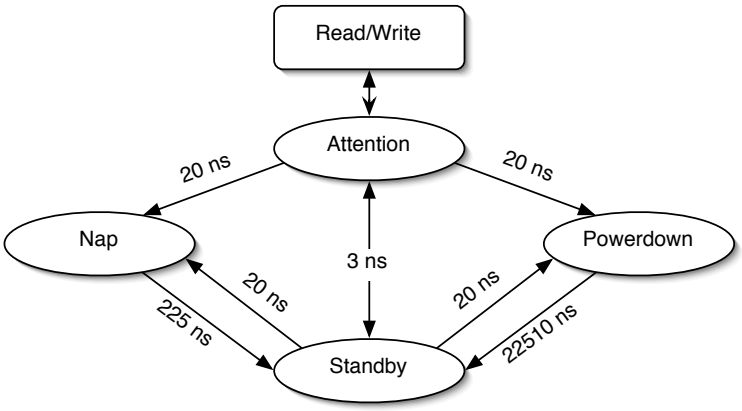

Abbildung 2.2: Energiezustände von RDRAM

Abbildung 2.2 zeigt die möglichen Zustände von RDRAM, die möglichen Übergänge zwischen den Zuständen und die Zeit, die für den Zustandswechsel benötigt wird.

In Tabelle 2.1 sind der Strombedarf und die noch aktiven Komponenten der Zustände dargestellt [Ram99, HPS03].

Da jedoch im Betriebssystem davon ausgegangen wird, dass der Speicher mit parallel geschalteten Bänken verwendet wird, wird benötigter Speicher nicht sequenziell sondern

Zustand	Verbrauch	Aktive Komponenten
Attention	323 mW	Refresh, Clock, Row decoder, Column decoder
Standby	225 mW	Refresh, Clock, Row decoder
Nap	11 mW	Refresh, Clock
Powerdown	7 mW	Refresh

Tabelle 2.1: Energieverbrauch und aktive Komponenten eines RDRAM-Bausteins in den verschiedenen Energiesparzuständen (nach [HPS03])

zufällig reserviert.

Dies führt dazu, dass der verwendete Speicher relativ gleichmäßig auf alle Bänke verteilt wird. Bei SDRAM bringt dies Geschwindigkeitsvorteile mit sich. Bei RDRAM hingegen wird dadurch verhindert, dass die Bänke in den Energiesparmodus schalten, da ja viele (im schlimmsten Fall alle) Bänke bei einem Speicherzugriff angesprochen werden müssen.

2.3.2 Neue Seitenreservierungsstrategien

In [LFZE00] wird eine neue Seitenreservierungsstrategie vorgestellt, die den Speicher sequentiell reserviert und somit den für einen Prozess benötigten Speicher auf wenige (im Idealfall eine) Bänke beschränkt.

Da dadurch der insgesamt benötigte Speicher an einem Stück im Hauptspeicher liegt, gibt es Bänke, die keine Daten enthalten und somit nie aus dem Energiesparmodus aufwachen müssen. Wird dieses neue Reservierungsschema mit RDRAM verwendet, wird bis zu 50 % der Energie, die mit dem „normalen" Reservierungsschema und RDRAM benötigt würde, gespart.

[HPS03] beschäftigt sich mit „Power Aware Virtual Memory". Hier wird gezeigt, dass mit Hilfe von RDRAM und angepasster Speicherverwaltung ohne spürbare Performance-Einbußen 54 - 94 % Energie eingespart werden können.

Erreicht wird dies durch eine Änderung der Speicherverwaltung von UMA auf NUMA[1]. Bei NUMA gibt es verschiedene Speicherbereiche (Nodes), die sich getrennt ansprechen lassen. Diese Nodes werden von den Autoren so gelegt, dass sie mit den Bänken des RAMs übereinstimmen. Dadurch, dass versucht wird, den Speicher eines Prozesses in einem Node zu reservieren, wird erreicht, dass die Prozesse immer möglichst wenige Nodes verwenden. Das heißt, dass sie auch in einer oder wenigen Bänken des RDRAM liegen, und somit bei Ausführung eines Prozesses nur die betroffenen Bänke aktiv sein müssen und der restliche Speicher in den Ruhezustand versetzt werden kann.

Die durch die Aktivierung und Deaktivierung der Bänke entstehenden Latenzen, die beim Prozesswechsel auftreten, können durch eine geschickte Scheduling-Strategie verdeckt werden. Dabei wird bei jedem Prozesswechsel zum einen der Speicher des verdrängten Prozesses deaktiviert und zum anderen nicht nur der Speicher des besten sondern auch der des zweitbesten Kandidaten aktiviert. Dieser ist bis zum nächsten Kontextwechsel aktiv und

[1] UMA: Für Ein-Prozessor-Rechner. Der verfügbare Speicher steht der CPU als Einheit zur Verfügung.
NUMA: Für Mehr-Prozessor-Rechner. Jede CPU hat einen eigenen Hauptspeicher, den sie direkt adressieren kann. Auf den Hauptspeicher der anderen CPUs kann sie nur über einen (langsameren) Bus zugreifen.

kann sofort verwendet werden.

Eine elegantere Möglichkeit ist, die Latenzen durch die Zeit, die für den Kontextwechsel benötigt wird, verdecken zu lassen. Dies hat den Vorteil, dass nur der tatsächlich benötigte Speicher aktiv ist. Das Problem dabei ist aber, dass die Zeit für den Kontextwechsel mit immer schnelleren CPUs leicht abnimmt und es so doch zu Latenzen kommen kann.

Zusätzlich werden noch folgende Optimierungen eingeführt:

dll aggregation Alle geladenen Bibliotheken werden in Node 0 beginnend am Stück im Speicher abgelegt. Dies hat den Vorteil, dass die Anzahl der aktiven Nodes weiter reduziert wird, da die Bibliotheken nicht in dem Node liegen, in dem der Prozess liegt, den die Bibliothek als erstes benötigt hat.

page migration Von nur einem Prozess verwendete Bibliotheken werden in den Node verschoben, in dem der Prozess liegt. Dies führt zu einer Verkleinerung des Speichers, der für die global verwendeten Bibliotheken verwendet wird, was dazu führt, dass weniger Nodes für die Bibliotheken benötigt werden und somit weniger Nodes immer aktiv sein müssen.

reducing migration overhead Dabei werden die Prozesse klassifiziert in *private-page dominated* und *public-page dominated*. Bei als *private-page dominated* klassifizierten Prozessen wird keine *dll-aggregation* vorgenommen, da der Prozess viele spezielle, nur von diesem Prozess verwendete Bibliotheken verwendet. Diese würden bei der *page migration* wieder zurück verschoben. Somit wird weitere Energie eingespart, da Umlagerungen entfallen. Anwendungen, die zur Klasse *public-page dominated* gehören, werden normal behandelt.

2.4 Kompression durch dedizierte Hardware

In [TFR+01] wird vorgestellt, wie mit Hilfe spezieller Hardware der Hauptspeicher komprimiert werden kann.

Dabei wird der Hauptspeicher nicht mehr direkt angesprochen sondern alle Zugriffe auf den L3-Cache umgeleitet. Der L3-Cache ist ein großer und schneller Cache, in dem häufig verwendete Daten unkomprimiert gespeichert werden. Werden Daten angefordert, so werden diese zuerst im L3-Cache gesucht. Sind sie nicht enthalten, so werden sie aus dem Hauptspeicher gelesen. Gleichzeitig werden Daten aus dem Cache in den Hauptspeicher ausgelagert.

Dabei kommt das eigentliche Herzstück dieser Technik zu Einsatz: Ein spezieller Chip, der die zu lesenden Daten dekomprimiert und die zu schreibenden Daten komprimiert. Diese Vorgänge laufen parallel. Dabei wird nach der Kompression zuerst noch einmal geprüft, ob die Kompression auch Platz spart. Ist dies nicht der Fall, so werden die Daten unkomprimiert geschrieben (und später natürlich auch nicht mehr dekomprimiert).

Dieses Verfahren führt im schlechtesten Fall zu einer Verdopplung des Hauptspeichers. Im besten Fall kann sogar das Sechsfache des physikalisch vorhandenen Hauptspeichers verwendet werden.

Kapitel 3

Speicherverwaltung unter Linux

Dieses Kapitel beschreibt zunächst die Speicherverwaltung eines Betriebssystems am Beispiel von Linux. Anschließend wird erklärt, wie der Hauptspeicher in eine Datei kopiert werden kann. Des Weiteren werden die Unterschiede und die Klassifikation von Daten- und Codesegment aufgezeigt.

Um mehrere Prozesse auf einem Computer laufen zu lassen, ist die direkte Adressierung von Hauptspeicher ungeeignet, da hier für jeden Prozess bekannt sein müsste, an welche Adressen er geladen wird. Deshalb wurde die virtuelle Adressierung entwickelt.

Bei dieser wird jedem Prozess ein eigener virtueller Speicher (=logischer Adressraum) zugewiesen, der dann in den realen Speicher abgebildet wird. Dabei „sieht" der Prozess nur den logischen Adressraum, der bei 0 beginnt und meist wesentlich größer ist als der verfügbare Hauptspeicher (bei Linux normalerweise 4 GB). Sämtliche Befehle des Programms beziehen sich auf den logischen Adressraum und werden dann von der Speicherverwaltung zu realen Adressen transformiert.

Da innerhalb eines Zeitraums meist nur ein sehr kleiner Teil des kompletten Addressraums benötigt wird, kann der nicht benötigte Speicher ausgelagert werden. Es reicht, wenn nur die benötigten Programm- und Datenbereiche (=„working-set") im Speicher gehalten werden. Um die nicht benötigten Bereiche auslagern zu können, ist der logische Adressraum in Seiten aufgeteilt (Größe typischerweise 256 Byte bis 64 KB). Diese Seiten werden dann bei Bedarf nachgeladen, bzw. wenn sie nicht mehr benötigt werden, wieder ausgelagert. Dies alles geschieht für den Prozess transparent, das heißt er merkt davon nichts.

Um ein solches dynamisches Laden (*Demand Loading* bzw. *Demand Paging*) zu ermöglichen, wird für jeden Prozess eine Liste verwaltet, die sämtliche Regionen des virtuellen Adressraumes enthält. Unter Linux funktioniert dies durch einen Eintrag in der `task_struct`, der auf eine Instanz von `mm_struct` zeigt. Dort wiederum mit in einem Eintrag auf eine verkettete Liste von `vm_area_struct`s gezeigt (siehe Abb. 3.1).

Jedes `vm_area_struct` enthält eine Region des virtuellen Adressraums, einen Pointer auf das nächste `vm_area_struct` und weitere Daten über die Region (Zugriffsrechte, Dateinamen, etc.) [Gor04, QW04]. Die Anzahl der Regionen variiert von Mal zu Mal, je nachdem wie fragmentiert der eigentliche Hauptspeicher ist.

Um jetzt den Speicher eines Prozesses auszulesen, genügt es somit, durch die Liste der `vm_area_struct`s zu gehen und die Regionen und benötigte Zusatzdaten auszugeben.

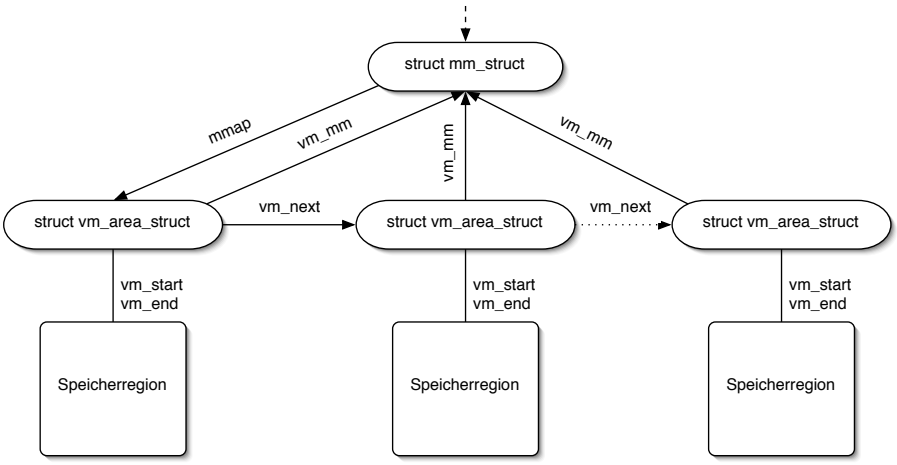

Abbildung 3.1: Aufbau des Prozessspeichers

Dies wurde in dieser Arbeit mit Hilfe des Programms *dump*[1] erledigt.

3.1 Auslesen des Speichers

dump wurde ursprünglich von David Duffey entwickelt, um dem Prozess sämtlichen Speicher zu entziehen, damit anderen Prozessen mehr Speicher zur Verfügung stünde, und diesen später wieder zurückzuladen, damit der Prozess selber weiter ausgeführt werden könnte (siehe [Duf02]).

dump ist ein Kernelmodul, das nach dem Laden im */proc*-Verzeichnis eine Schnittstelle bereitstellt, mit deren Hilfe der Prozessspeicher in eine Datei kopiert werden kann.
Für diese Arbeit wurde *dump* so verändert, dass der Speicher nur ausgegeben (=dupliziert) wird, jedoch nicht aus dem Hauptspeicher entfernt wird.

Nach dem Erzeugen des dump-Files (Struktur siehe Abbildung 3.2) muss dieses in die einzelnen Speicherregionen zerlegt werden und als Daten- bzw. Codesegment klassifiziert werden.

Für die Klassifikation wurden mehrere Verfahren getestet:

1. Als Codesegment wird eine Region dann gewertet, wenn die Region eine gemappte Datei ist.
   Dies war der allererste Versuch, der aber verworfen wurde, da auch Datensegmente gemappte Dateien sein können (z. B. ein Musikstück bei einem Audioplayer)

2. Der erste Ansatz wurde verfeinert, indem Einschränkungen für die Dateien getroffen wurden. Jetzt musste der Dateiname entweder der Programmname sein oder aber die Endung „.so" enthalten.
   Der Nachteil hierbei war, dass auch Libraries ein Datensegment haben, das mit dieser Methode aber als Codesegment klassifiziert wird.

[1] Obwohl angenommen werden könnte, dass es für Linux mehrere Programme gibt, die den Speicher eines Prozesses in eine Datei schreiben, ist im Verlauf dieser Arbeit nur ein einziges Tool bekannt geworden, das dies in der hier benötigten Form tut.

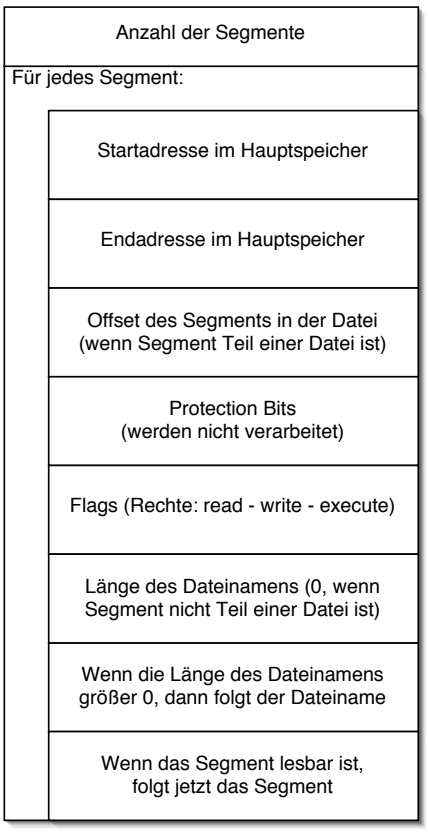

Abbildung 3.2: Aufbau der von *dump* erzeugten Dateien

3. Im nächsten Ansatz wurden die Rechte betrachtet. Als Codesegmente sollten diejenigen gelten, bei denen das x-Flag (ausführbar) gesetzt ist.
   Das Problem hierbei ist, dass auch Datensegmente in einigen Fällen als ausführbar markiert werden.

4. Ein anderer Ansatz wertete alle nicht schreibbaren Regionen (kein w-Flag) als Codesegmente.
   Aber auch hier besteht das Problem, dass Datensegmente nicht unbedingt schreibbar sein müssen (Musikstück beim Audioplayer).

5. Das verwendete Verfahren klassifiziert alle Regionen, die lesbar und ausführbar (Rechte: r - x) sind, als Codesegmente. Alle anderen Regionen (in Frage kommen hier noch folgende Rechte: r - -, r w -, r w x) sind Datensegmente.

Zum Zerlegen und Klassifizieren wurde ein kleines Tool geschrieben, das einen Speicherdump analysiert, entsprechende Verzeichnisse anlegt und darin die einzelnen Regionen als passend benannte Dateien ablegt. Anhand des Dateinamens ist unter anderem erkennbar, ob es sich um ein Code- oder Datensegment handelt, zu welchem Speicherdump die Region gehört und die wievielte Region es ist. Zusätzlich wird für jeden Speicherdump noch eine Datei erstellt, in der sämtliche Daten der Header der Regionen gespeichert sind.

Diese Dateien werden dann direkt zum Testen der Algorithmen verwendet.

3.2 Code- und Datensegmente

Der Aufwand zur Trennung und Klassifikation der Speichersegmente ist sinnvoll, da sich die Struktur der Daten in den Segmenten unterscheidet. Diese Strukturen werden jetzt aufgezeigt.

3.2.1 Codesegment

Die Codesegmente haben zumeist eine sehr regelmäßige Struktur. Normalerweise besteht das Segment aus einer Reihe von Instruktionen und deren Parametern. Dabei machen 256 verschiedene Instruktionen 50 % - 99 % der Gesamtinstruktionen aus (siehe [BMMP99]). Dies ermöglicht, gerade bei RISC-Architekturen[2], eine gute und einfache Kompression des Segments (die Instruktionen können durch 1-Byte-Tags ersetzt werden).

Im Codesegment verwendete Zahlen (Konstanten) sind sehr häufig Integerwerte, die im Bereich von -1000 bis 1000 liegen. Somit bestehen die High-Bits aus Einsen (bei Werten <0) bzw. Nullen (Werte >0).

Auch die Sprunganweisungen zeigen häufig auf Speicheradressen im gleichen Segment. Damit ergeben sich gleiche Muster, die komprimiert werden können (siehe [KI02, BW03]).

Für diesen Segmenttyp eignen sich voraussichtlich speziell zugeschnittene Kompressionsalgorithmen, die die Offsets von Adressen entfernen und die Instruktionen im Ganzen komprimieren.

3.2.2 Datensegment

Zu den Datensegmenten zählen in dieser Arbeit auch der Stack und der Heap.

Die im Datensegment gespeicherten Daten sind entweder Pointer auf andere Stellen im Segment oder Nutzdaten.

Pointer zeigen häufig auf Adressen im gleichen Segment, somit sind auch hier gleiche Muster zu finden. Oft kommt es auch vor, dass mehrere Pointer eines Bereichs auf dieselbe Adresse zeigen (siehe [WKS99, KI02]) und daher identisch sind.

Die Nutzdaten sind entweder Zeichen oder Zahlen.
Während Zahlen immer mehrere Byte lang sind, sind Zeichen nur ein Byte lang.
Dementsprechend sind etwaige Muster bei Zeichen kürzer als bei Zahlen.

Hier gibt es also wesentlich mehr unterschiedliche Optionen als im Codesegment, was eine wesentlich höhere Flexibilität des Kompressionsalgorithmus erfordert.

[2] Alle Befehle sind in Zeit und Raum gleich lang. Somit kann eine konstante Wortgröße bei den Vergleichen zur Kompression eingesetzt werden.

Kapitel 4

Evaluation der Kompressionsalgorithmen

In diesem Kapitel werden die in dieser Arbeit verwendeten Kompressionsalgorithmen und der Ablauf der Messungen beschrieben.

4.1 Kompressionsalgorithmen

4.1.1 LZ77

Der Algorithmus wurde von Abraham Lempel und Jacob Ziv in den späten 70er Jahren entwickelt und 1977 in dem Artikel „A Universal Algorithm for Sequential Data Compression" [ZL77] vorgestellt.

Die Einführung dieses Algorithmus hatte starken Einfluss auf die Informationstheorie. Noch heute wird er in Abwandlungen in vielen bekannten Kompressionsalgorithmen eingesetzt (z. B. ZIP und ARJ).

Bei LZ77 kommt ein zweiteiliges Fenster zur Anwendung, das vom Anfang der zu komprimierenden Daten (=links) bis zu deren Ende (=rechts) geschoben wird. Als Trennung im Fenster dient die aktuelle Position. Der links der aktuellen Position liegende Teil des Fensters ist der so genannte *Lempel*, der die zuletzt kodierten Symbole enthält. Er ist mehrere Tausend Byte groß (z. B. 4096 Byte). Der rechte Bereich des Fenster wird *Ziv* genannt. Er ist wenige Bytes groß (z. B. 16 Byte) und enthält die als nächstes zu kodierenden Zeichen.

Beim Kodieren wird der Lempel von rechts nach links nach einer Übereinstimmung mit dem ersten Symbol im Ziv durchsucht. Danach wird überprüft, wie viele aufeinander folgende Symbole übereinstimmen. Dann wird die Suche nach links fortgesetzt und nach weiteren Übereinstimmungen gesucht. Schließlich wird die längste, bzw. die zuletzt gefundene Übereinstimmung ausgewählt und ein Tripel `(O, L, C)` ausgegeben. Dabei ist `O` der Offset von der aktuellen Position zum Anfang der Übereinstimmung, `L` die Länge der Übereinstimmung und `C` das erste Symbol. Wird keine Übereinstimmung gefunden, so wird nur ein Zeichen durch das Tripel kodiert, da ja das erste Symbol im Tripel enthalten ist. Nach der Ausgabe wird das Fenster um die Anzahl der übereinstimmenden Symbole nach rechts geschoben. Dieser Vorgang wird wiederholt, bis die kompletten Daten verarbeitet sind.

Bei der Dekompression wird das Tripel gelesen und entsprechend der darin enthaltenen Angaben werden die Symbolketten ausgegeben [Hra, Jäg, ZL77].

Die beiden Algorithmen LZRW1 und LZO, die in dieser Arbeit eingesetzt werden, basieren auf der eben beschriebenen Implementierung von Lempel und Ziv.

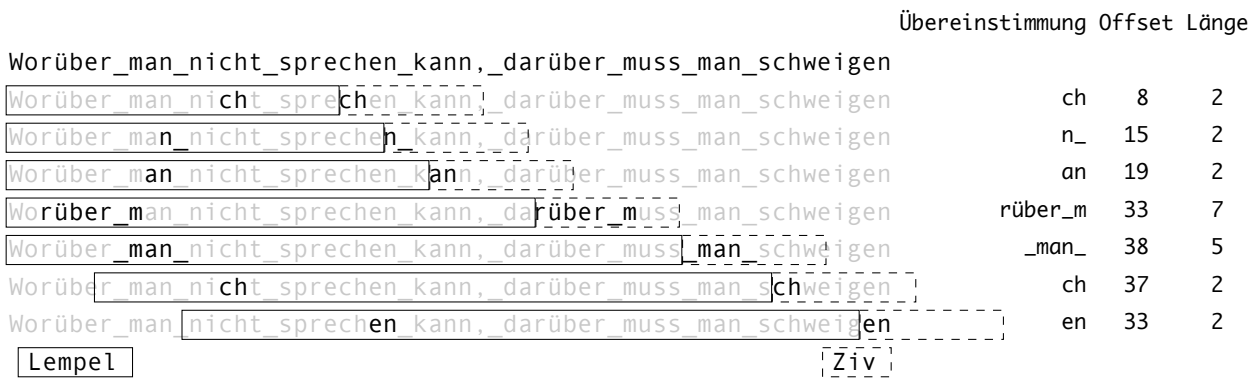

Abbildung 4.1: Darstellung der Suchoperationen von LZ77 (Übereinstimmungen von einzelnen Zeichen wurden weggelassen)

4.1.1.1 LZRW1

LZRW1 ist eine Abwandlung des LZ77 und auf Geschwindigkeit optimiert. Dies wird durch eine Hash-Tabelle erreicht, der das Suchen von Übereinstimmungen der nächsten Werte des Ziv mit Werten im Lempel beschleunigt.

Dabei existieren folgende Unterschiede:

Es wird nicht nur ein Byte sondern immer gleich die nächsten drei Bytes des Ziv verwendet. Dann wird über eine Hashfunktion ein Eintrag in der 4096-elementigen Hash-Tabelle selektiert und auf Übereinstimmung geprüft. Wenn mindestens drei Bytes übereinstimmen und der Pointer in die letzten 4096 Bytes (Lempel) zeigt, werden wie bei LZ77 die Bytes aus dem Ziv entfernt und durch einen Pointer ersetzt. Zusätzlich wird in jedem Fall der Hash-Eintrag durch einen Eintrag auf die aktuelle Position ersetzt, um die Hash-Tabelle aktuell zu halten [Wil91].

4.1.1.2 miniLZO

In dieser LZ77-Implementierung wird besonderer Wert auf Geschwindigkeit und geringen Platzbedarf bei (De-)Kompression gelegt.

Der Algorithmus eignet sich besonders für eingebettete Systeme, bei denen Geschwindigkeit wichtiger ist als die Kompressionsrate [Obe].

4.1.2 WKdm und WK4x4

Im Gegensatz zu dem in LZ77 verwendeten 4096-elementigen Suchbereich kommt bei WKdm bzw. WK4x4 nur ein 16-elementiges Wörterbuch zum Einsatz. Auch dieses ändert sich während der (De-)Kompression, und zwar immer dann, wenn ein Wort nicht im Wörterbuch enthalten ist. Dann wird ein Eintrag aus dem Wörterbuch mit dem aktuellen Wort überschrieben.

Bei der Kompression werden, wie auch bei den LZ77-Arten, die zu komprimierenden Daten einmal durchlaufen und dabei komprimiert. Dabei wird jedes Wort mit einem Wörterbucheintrag verglichen, der anhand einer Hashfunktion ausgewählt wird, und nicht mit

dem kompletten Wörterbuch, wie vielleicht zu erwarten wäre.

Im Gegensatz zu den LZ77-Arten wird bei WKdm/WK4x4 der Pointer auf das frühere Vorkommen und die Länge der Übereinstimmung nicht direkt im Text eingetragen sondern in eigenen Bereichen.

WKdm/WK4x4 unterscheiden dabei vier verschiedene Bereiche:

Tag-Bereich speichert für jedes Wort einen 2-Bit-Eintrag, der die Verarbeitung des Wortes angibt.
Es gibt vier verschiedene Tags (Beschreibung für die Kompression, bei der Dekompression wird entsprechend gelesen):

Zero gibt an, dass das Wort aus Nullen besteht. Es wird sonst nichts mehr geschrieben.

Match zeigt eine Übereinstimmung mit einem Wörterbucheintrag an.
Im Pointer-Bereich wird der Index des passenden Wörterbucheintrags hinterlegt.

Partial Match zeigt an, dass die oberen 22 Bit des Wortes mit einem Wörterbucheintrag übereinstimmen.
Im Pointer-Bereich wird der Index des passenden Wörterbucheintrags gespeichert. Zusätzlich werden im Partial-Match-Bereich die unteren 10 Bit gesichert. Der Wörterbucheintrag wird durch das aktuelle Wort ersetzt.

Mismatch gibt an, dass es keine Übereinstimmung mit dem Wörterbuch gibt.
Das Wort wird in den Mismatch-Bereich geschrieben und es ersetzt den bisherigen Eintrag im Wörterbuch.

Pointer-Bereich Hier wird bei Übereinstimmung und teilweiser Übereinstimmung der Index (4 Bit) des passenden Wörterbucheintrages hinterlegt.

Mismatch-Bereich speichert die Wörter (32 Bit), die nicht im Wörterbuch enthalten waren.

Partial-Match-Bereich speichert die unteren 10 Bit bei einer teilweisen Übereinstimmung.

Erst nach der Kompression werden die Bereiche, deren Einträge während der Kompression in Arrays geschrieben wurden, mit einer schnellen Routine „gepackt", so dass der „Verschnitt" minimiert wird (Tag-Bereich: 16 Elemente je Wort (32 Bit), Pointer-Bereich: 8 Elemente je Wort, Partial-Match-Bereich: 3 Elemente je Wort). Der einzige Unterschied zwischen WKdm und WK4x4 besteht in der Verwaltung des Wörterbuchs.

WKdm verwendet einen direkt gemappten Cache, bei dem das gesuchte Wort per Hashfunktion genau einem Eintrag des Wörterbuchs zugeordnet wird.

Bei WK4x4 kommt dagegen ein „4-way set associative cache" [Kap99] mit LRU als Ersetzungsschema zur Anwendung. Bei diesem wird das gesuchte Wort einem Set zugeordnet. Jedes Set besteht aus vier Wörtern, die nach dem Zeitpunkt der letzten Benutzung sortiert sind (das zuletzt verwendete Wort zuerst). Das gesuchte Wort wird dann mit jedem Wort des Sets verglichen und ersetzt bei einem Mismatch das älteste Wort des Sets [WKS99, Kap99].

4.1.3 WHdm

Da der WKdm-Algorithmus ursprünglich nur für 4 KB-Segmente funktionierte, in dieser Arbeit aber auch andere Segmentgrößen getestet werden sollten, wurde der Algorithmus komplett neu implementiert.

Dabei wurden alle statischen Speicherreservierungen durch dynamische ersetzt, was den Algorithmus zwar etwas verlangsamt, den Speicherbedarf aber reduziert. Zusätzlich werden die Werte der verschiedenen Bereiche direkt bei der Kompression zusammengefasst, was den Speicherbedarf noch weiter senkt.

4.1.4 WHsd

Mit WHsd wurde ein komplett anderer Ansatz implementiert. Hier kommt ein statisches Wörterbuch zum Einsatz, das zu Beginn der Kompressionsphase aus den am häufigsten vorkommenden Wörtern im Segment gebildet wird.

Hierzu wird das komplette Segment durchgegangen und die Auftrittshäufigkeit jedes Wortes gespeichert. Aus den 15 (255)[1] häufigsten Wörtern wird dann das Wörterbuch gebildet.
Danach wird das Segment noch einmal durchlaufen und dabei komprimiert.

Der Ablauf der Kompression ist dabei folgendermaßen:

Zuerst wird das Wörterbuch in den Speicher (für das komprimierte Segment) geschrieben, anschließend die beiden Bereiche Tag-Area und Miss-Area.

Das *Tag-Area* enthält für jedes Wort aus dem Originalsegment einen Eintrag von 4 (8) Bit. Dieser ist entweder 0-14 (254) (= Wörterbuchelement, das mit dem Originalwort übereinstimmt), wenn das zu komprimierende Wort im Wörterbuch enthalten ist, oder 15 (255), wenn das Wort nicht im Wörterbuch enthalten ist.

Ist das Wort nicht enthalten, so wird das Wort in den zweiten Bereich geschrieben, dem *Miss-Area*. Die Größe dieses Bereichs ist am Anfang unbekannt, da nicht berechnet wird, wie viele Elemente des Segments nicht im Wörterbuch enthalten sind.
Das Tag-Area hingegen hat eine feste Größe: Anzahl der Wörter * 4 (8) Bit.

Dieser Algorithmus ist bei der Kompression extrem langsam, da zum einen der zu komprimierende Bereich zweimal komplett gelesen wird und zum anderen das Erzeugen des Wörterbuchs sehr viele Vergleichsoperationen beinhaltet (Erkennung, ob das Wort schon in der Liste enthalten ist).

Dies könnte beschleunigt werden, indem mehrdimensionale Arrays verwendet werden und die Bytes des Wortes als Indizes des Arrays verwendet werden. Der Nachteil dabei ist jedoch der sehr hohe Platzbedarf von Beginn an. So werden bei einem zweidimensionalen Array schon 65.536 Einträge reserviert, von denen niemals alle verwendet werden[2].

Der große Vorteil dieses Kompressionsalgorithmus ist die schnelle Dekompression. Durch das statische Wörterbuch sind pro Wort nur ein Vergleich und drei Speicherzugriffe nötig (1. Zugriff: Tag holen, Vergleich: Tag-Vergleich, 2. Zugriff: Element aus Wörterbuch/Miss-Area holen, 3. Zugriff: Element schreiben). Alle anderen Algorithmen benötigen wesentlich mehr Operationen.

Dieser Algorithmus stellt bei den Dekompressionszeiten die untere Grenze dar.

[1] Dieser Algorithmus wurde mit zwei verschiedenen Wörterbuchgrößen implementiert: einmal 16 Elemente und einmal 256 Elemente. Der Wert in Klammern stellt immer den Wert für den Algorithmus mit dem 256-elementigen Wörterbuch dar.

[2] Das größte verwendete Segment hat 64 kB, d. h. 16.000 Wörter. Im Array sind aber 65.536 mögliche Kombinationen reserviert. Zudem sollte das Segment ja komprimierbar sein, also Wiederholungen desselben Wortes beinhalten.

4.2 Testverfahren

4.2.1 Verwendete Hardware

Die (De-)Kompressionen wurden auf einem Triton Starterkit II mit einem Intel XScale PXA255 mit 400 MHz und 16 MB SD RAM durchgeführt [KARb, KARa, Int].
Die verwendeten Taktraten für Prozessor und der Speicher werden in Tabelle 4.1 aufgezeigt. Dabei wird bei Änderung der Taktrate der CPU auch die Versorgungsspannung der CPU angepasst. Der Energiebedarf des Gerätes im Leerlauf bei den verschiedenen Frequenzen wird in Tabelle 4.2 aufgeführt.

Bezeichnung	CPU	Speicher
400	398	99
300	298	99
200	199	99
100	99	49

Tabelle 4.1: Getestete Taktraten von CPU und Speicher (in MHz).

CPU	Energieverbrauch
100 MHz	0.31 W
200 MHz	0.34 W
300 MHz	0.36 W
400 MHz	0.43 W

Tabelle 4.2: Leerlauf-Energieverbrauch der verschiedenen CPU-Geschwindigkeiten.

Auf dem Rechner wird Linux mit einem Kernel der Version 2.6.3 eingesetzt.
Die Energiemessung wird an der 3,3 Volt-Leitung mit einem zwischengeschaltetem Messwiderstand vorgenommen. Die Werte werden mit einem AD-Wandler mit 8-Bit-Auflösung bei einer Auflösung von 20.000 Samples pro Sekunde auf einem zweiten Rechner gespeichert.

4.2.2 Testdaten

Als Testdaten wurden typische Anwendungen für mobile Geräte aus den folgenden Bereichen verwendet:

- Netzwerkanwendungen - insbesondere Internet
- Textverarbeitung
- Bildbearbeitung
- Audio-Anwendungen

Die Speicherabbilder wurden auf einem iPAQ mit OPIE-Linux gemacht.
Es wurden Dumps von folgenden Programmen angefertigt:

konqueror - Webbrowser (Netzwerkanwendung)
Der Speicher des Programms wurde einmal ohne geladene Webseite, einmal mit einer einfachen Seite (Nur HTML mit CSS, ein kleines Bild) und einmal mit einer komplexen Seite (Frames, Grafiken, HTML, CSS) ausgegeben.
Die Besonderheit bei diesem Programm ist, dass es mit mehreren Threads arbeitet, die alle ausgegeben wurden.

madplay - einfacher Kommandozeilen-Audioplayer (Audio-Anwendung)
Der Speicher dieses Programms wurde nur einmal ausgegeben, nämlich mit geöffnetem Musikstück, da es eher unwahrscheinlich ist, dass man auf Kommandozeile einen Audioplayer ohne Musikstück startet.

showimg - Bildbetrachter (Bildbearbeitung)
Auch hier wurden zwei Speicherabbilder erstellt, einmal nur das Programm ohne geöffnete Dateien und einmal während ein Bild angezeigt wurde.

textedit - Texteditor (Textverarbeitung)
Hier wurden drei Speicherabbilder erstellt. Einmal kurz nach dem Start ohne geöffnete Dateien, einmal mit einem kurzen Text und einmal mit einem mehr als tausend Zeilen langen Text.

4.2.3 Messablauf

Zur Laufzeit können die kompletten Segmente nicht auf einmal komprimiert werden, da dabei viel zusätzlicher Speicher auf einmal benötigt wird. Zudem ist der virtuelle Hauptspeicher bereits in Seiten (je nach System mit anderer Größe) unterteilt.

Infolge dessen werden beim Test verschiedene Blockgrößen (4 KB, 8 KB, 16 KB, 32 KB, 64 KB) verwendet. Das heißt, dass die Segmente in einzelne Blöcke unterteilt werden, die dann komprimiert und dekomprimiert werden. Dabei werden zuerst alle Blöcke komprimiert und danach alle Blöcke wieder dekomprimiert.

Für den Test wurde ein Programm geschrieben, das die übergebene Datei mit allen Algorithmen bei der übergebenen Blockgröße komprimiert und dekomprimiert. Dabei wird für jeden Algorithmus die Anzahl der Durchläufe berechnet, damit die Gesamt(de)kompressionszeit ca. 350 ms beträgt (Sollte ein Durchlauf länger benötigen, wird dieser natürlich komplett verarbeitet). Dies ist nötig, da bei bestimmten Segment-Algorithmus-Konstellationen die Kompression nur einige ms benötigt und dadurch mögliche Messfehler (z. B. durch parallel ausgeführte Prozesse) das Ergebnis stark verfälschen können.

Um nicht am Ende des Segments einen kleineren Block verarbeiten zu müssen, wird zu Beginn die Größe des Segments überprüft und bei Bedarf am Ende des Segments die fehlenden Bytes angehängt (mit Nullen initialisiert).

Zusätzlich prüft das Programm die Größe der übergebenen Datei und führt nur Berechnungen durch, wenn die Datei mindestens halb so groß ist wie Blockgröße (also mindestens 2048 Byte bei einer Blockgröße von 4096 Byte). Der Grund hierfür ist, dass sich Null-Werte sehr gut komprimieren lassen und, wenn das Segment zum größten Teil aus Nullen besteht, keinerlei Aussage mehr über die Komprimierbarkeit der Daten getroffen werden kann.

Der Ablauf der Messung wird über ein Skript gesteuert, das nach Einstellung der CPU-Geschwindigkeit alle zu testenden Segmente durchgeht. Für jedes Segment werden alle

Blockgrößen durchgegangen und damit das Testprogramm aufgerufen. Nach Abschluss sämtlicher Messungen verändert das Skript wieder die CPU-Geschwindigkeit und startet den nächsten Messlauf.

Das Ergebnis ist für jede CPU-Geschwindigkeit, für jedes Segment, für jede Blockgröße und jeden Algorithmus ein Datensatz, der die folgenden Daten enthält:

- Kompressionsrate
- Dauer der Kompression
- Dauer der Dekompression
- Energieverbrauch der Kompression
- Energieverbrauch der Dekompression
- Anzahl der Wiederholungen der Kompression
- Anzahl der Wiederholungen der Dekompression

Insgesamt wurde der komplette Messlauf viermal wiederholt, um die Messungenauigkeiten weiter zu reduzieren und die gemessenen Daten zu validieren.

4.2.4 Verwendete Kennzahlen

Um vergleichbare Werte zu erhalten werden folgende Kennzahlen berechnet und ausgewertet:

Kompressionsrate, bzw. Kompressionsverhältnis
: Diese Größe gibt an, wie erfolgreich die Kompression war.
Eine hohe Kompressionsrate bedeutet, dass die Daten nur wenig komprimiert werden konnten. Eine niedrige Kompressionsrate gibt an, dass die Kompression erfolgreich war, also die Größe der komprimierten Daten im Vergleich zu der Größe der Originaldaten stark abgenommen hat.

Kompressionsgeschwindigkeit in MB pro Sekunde
: Darunter wird der Quotient aus der Größe der unkomprimierten Daten und der zur Kompression benötigten Zeit. Der Algorithmus ist um so besser, je höher dieser Wert ist.

Leistungsbedarf in Joule pro MB unkomprimierter Daten
: Benötigte Leistung um ein MB der ursprünglichen Daten zu verarbeiten. Je niedriger dieser Wert ist, desto besser ist der Algorithmus.

Kapitel 5

Analyse

In diesem Kapitel werden die Ergebnisse der Messungen präsentiert und diskutiert.

5.1 WHsd16 und WHsd256

Wie schon in 4.1.4 beschrieben, müssen bei den Algorithmen WHsd16 und WHsd256 die Daten zweimal durchlaufen werden (einmal, um das Wörterbuch zu generieren und ein zweites Mal, um die Daten zu komprimieren). Dadurch sind diese Algorithmen bei der Kompression schon langsamer als alle anderen. Zusätzlich kommt noch hinzu, dass beim Erzeugen des Wörterbuchs sehr viele Suchoperationen ausgeführt werden müssen, um zu prüfen, ob ein Element schon einmal vorgekommen ist, oder ob es ein neues Element ist.

Dies führt dazu, dass WHsd16 für die Kompression mindestens fünfmal[1] länger braucht, als der langsamste der anderen Algorithmen.

WHsd256 benötigt sogar 17mal länger. Daraus resultiert auch der mindestens dreimal bzw. elfmal höhere Energieverbrauch (siehe Abb. 5.1).

Da auch die Kompressionsrate weit hinter den anderen Algorithmen zurückbleibt (siehe Abb. 5.2 und Tabelle 5.1) werden diese beiden Algorithmen bei den folgenden Auswertungen nur noch in besonderen Fällen beachtet.

	Segmenttyp	
Algorithmus	Daten	Code
WHsd16	68,9 %	96,7 %
WHsd256	76,4 %	92,5 %
WHdm	56,6 %	89,2 %

Tabelle 5.1: Durchschnittliche Kompressionsrate bei Kompression des kompletten Speichers

[1] Diese Werte wurden ermittelt, durch eine Summierung aller zu komprimierenden Daten ungeachtet der später betrachteten Unterscheidungen wie BlockSize und CPU-Geschwindigkeit. Stichproben unter Beachtung dieser Werte haben jedoch dasselbe Ergebnis ergeben.

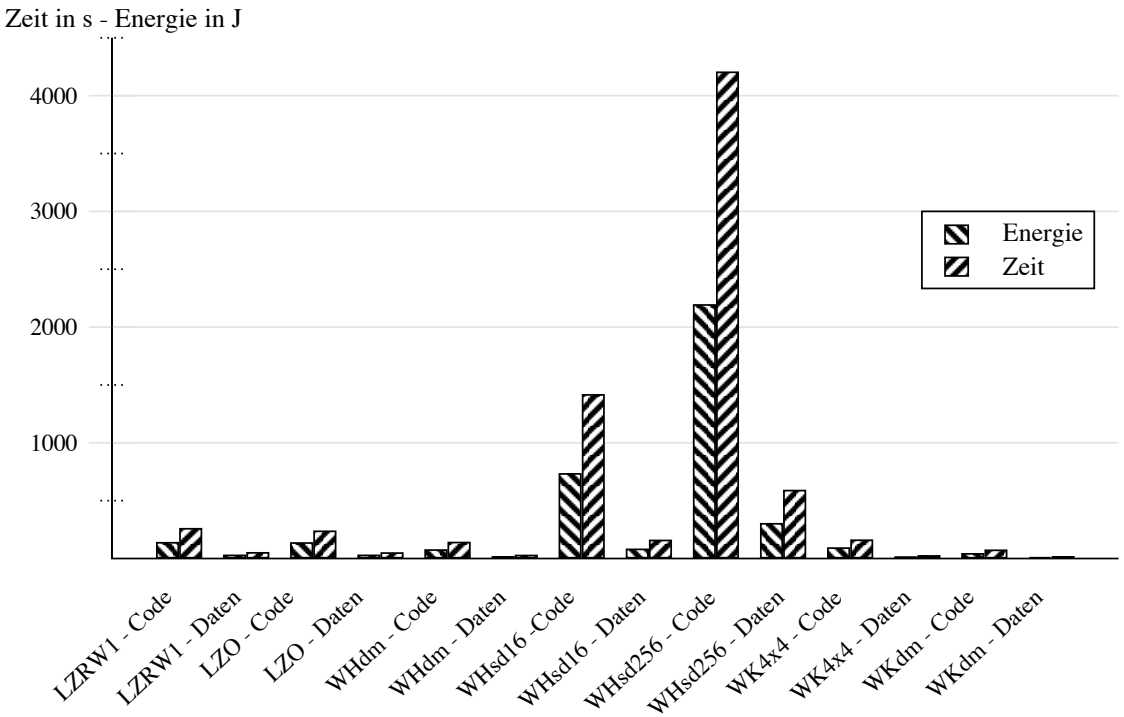

Abbildung 5.1: Zeit- und Energiebedarf der Kompression des gesamten Speichers

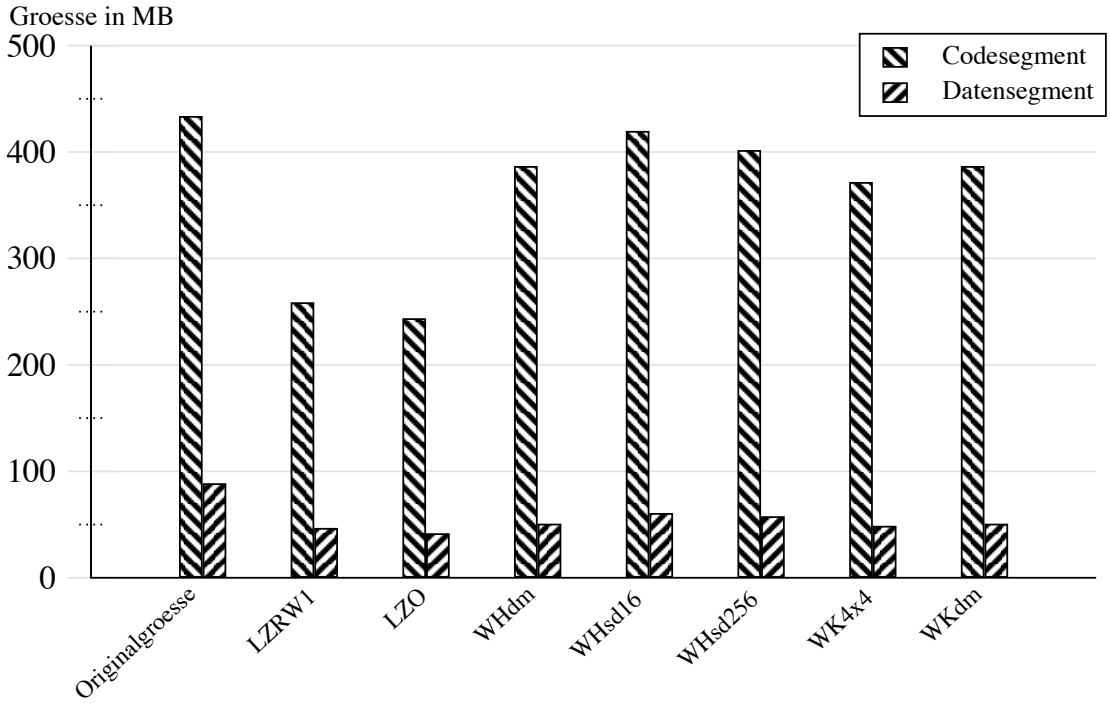

Abbildung 5.2: Größe der komprimierten Daten (Blockgröße 16 KB)

5.2 Auswertung der Anwendungsklassen

In diesem Abschnitt werden die Ergebnisse der Anwendungsklassen präsentiert und diskutiert. Dabei werden die optimalen Algorithmen für die Klassen und deren Daten- und Codesegmente ausgewählt.

Wenn im Folgenden von *Kompression* die Rede ist, ist immer der Wert Kompression plus Dekompression gemeint.

5.2.1 Bildbearbeitung - showimg

Von *showimg* wurden zwei Speicherdumps erstellt. Einmal das leere Programm und einmal das Programm mit einer geöffneten Grafik. Da diese Dumps kurz hintereinander vom selben Prozess erstellt wurden, sind die Codesegmente der beiden Dumps identisch [2]. Daher wird hier nur das Codesegment eines Dumps betrachtet.

	Segment		
	Code	Daten	
		leer	mit Daten
showimg	7,62 MB	1,17 MB	2,42 MB

Tabelle 5.2: Originalgröße der Segmente

Tabelle 5.2 zeigt die unkomprimierte Größe der Segmente. Aus ihr geht hervor, dass das Codesegment wesentlich größer ist als das Datensegment (mindestens Faktor drei). Eine niedrige Kompressionsrate im Datensegment spart somit nicht so viel Platz wie eine mittelmäßige Rate im Codesegment.

5.2.1.1 Codesegement

Die Kompressionsraten des Codesegments werden in Abbildung A.1 dargestellt. Auffällig ist der große Unterschied zwischen den LZ77-Algorithmen und den WK-Algorithmen (WHdm zählt auch zu dieser Klasse, da er ja nur eine Neuimplementation des WKdm ist).

Dieser Unterschied zeigt, dass die im Codesegment gespeicherten Daten entweder nicht in 4-Byte-Strukturen abgelegt sind, was die hohe (schlechte) Kompressionsrate der WK-Algorithmen erklärt, oder sich identische Daten in großen Abständen wiederholen, dafür aber dann gleich längere Teile identisch sind, was die niedrige Kompressionsrate der LZ77-Algorithmen erklärt.

Die unterschiedlichen Blockgrößen beeinflussen die Kompressionsrate nicht sehr stark. Die aus Abbildung A.1 ersichtliche Abnahme von ca. 3 % (4 KB zu 64 KB) bei den WK-Algorithmen ist darauf zurückzuführen, dass die Wahrscheinlichkeit, dass die zu komprimierenden Daten kleiner als die Blockgröße sind und daher mit Nullen aufgefüllt werden müssen, mit der Blockgröße steigt. Längere Ketten von Nullwerten sind jedoch sowohl mit den LZ-Algorithmen als auch mit den WK-Algorithmen effizient zu komprimieren.

Die ca. 7 % Verbesserung der LZ-Algorithmen resultiert daraus, dass mit steigender Blockgröße die volle Größe des Lempel länger verwendet werden kann (Bei 4 KB Blockgröße würde ein 4 KB großer Lempel erst im letzten Kompressionsschritt erreicht werden. Bei 64 KB

[2] Die Identität der Segmente wurde mit md5sum geprüft.

Blockgröße jedoch schon nach 4 KB und die restlichen 60 KB könnten auf die volle Größe des Lempel zugreifen).

Durch die Optimierung (Hash-Tabellen zur Suche im Lempel) der LZ-Algorithmen wirkt sich jedoch die Größe des Lempel nicht auf die Geschwindigkeit der Algorithmen aus. Der in Abbildung A.4 ersichtlichen Erhöhung der Geschwindigkeit mit steigender Blockgröße liegt wieder die erhöhte Anzahl der Nullwerte am Ende der Daten zugrunde.

Interessant ist auch, dass die Geschwindigkeit der Algorithmen linear zur CPU-Geschwindigkeit ansteigt (siehe Abb. A.4). Doppelte CPU-Geschwindigkeit bedeutet also auch doppelte Kompressionsgeschwindigkeit. Dies gilt auch für den Vergleich von 100 MHz, 200 MHz und 300 MHz. Bei der Steigerung von 100 MHz auf 200 MHz wird auch der Speichertakt verdoppelt. Von 200 MHz auf 300 MHz bleibt der Speichertakt unverändert. Trotzdem ist eine Steigerung der Kompressionsgeschwindigkeit um 50 % zu beobachten. Daraus kann gefolgert werden, dass die Taktrate des Speichers keinerlei Auswirkung auf die Geschwindigkeit der Kompression hat.

Dies ist jedoch beim Leistungsbedarf der Kompression nicht der Fall. Diese ist relativ unabhängig von der CPU-Geschwindigkeit, dafür aber von der Taktrate des Speichers abhängig.

Abbildung A.7 zeigt eine Abnahme der benötigten Leistung pro MB um ca. 10 % wenn die CPU-Geschwindigkeit verdoppelt wird (von 200 MHz auf 400 MHz).

Von 100 MHz auf 200 MHz sinkt jedoch der Leistungsbedarf pro MB um ca. 30 %.

In dieser Anwendungsklasse gilt also für das Codesegment (aus den Abbildungen A.4 und A.7), dass der Leistungsbedarf bei jeder CPU-Geschwindigkeit gleich ist, während die Kompressionsgeschwindigkeit mit der CPU-Geschwindigkeit zunimmt.

Somit ist die Kompression des Hauptspeichers bei maximaler CPU-Geschwindigkeit am effizientesten, da dort die Dauer der Kompression am niedrigsten ist.

Da der Kompressionsaufwand und -erfolg von der Blockgröße relativ unabhängig ist, kann die im System verwendete Größe für Seiten verwendet werden.

Die Auswahl des besten Algorithmus fällt bei getrennter Betrachtung der Geschwindigkeit und der Kompressionsrate nicht eindeutig aus.

Betrachtet man nur die Kompressionsgeschwindigkeit, so ist **WKdm** der Algorithmus der Wahl. Dieser benötigt für die Kompression die kürzeste Zeit, dafür ist die Kompressionsrate aber nur mittelmäßig.

Betrachtet man nur die Kompressionsrate, so schneidet **LZO** am besten ab. Dieser komprimiert mit der höchsten Rate, braucht dafür aber auch unverhältnismäßig viel mehr Leistung.

Es ist nicht möglich eine eindeutige Auswahl zu treffen, da es verschiedene Szenarien gibt, bei der unterschiedliche Kompressionsstrategien eingesetzt werden müssten.

Wenn zum Beispiel davon ausgegangen werden kann, dass die Anwendung für längere Zeit nicht mehr aktiviert wird, so ist der Einsatz des Algorithmus mit der niedrigsten Kompressionsrate ungeachtet des Leistungsbedarfs am besten.

Die zur Kompression zusätzlich benötigte Leistung ist im Verhältnis zu der eingesparten Leistung durch Abschalten eines größeren Speicherbereichs sehr gering.

Wenn jedoch zu Erwarten ist, dass die Anwendung nach kurzer Zeit wieder aktiviert wird, dann ist es besser, den sparsamsten Algorithmus zu verwenden.

Die für die höhere Kompressionsrate aufgewendete Leistung kann hier nicht über die Zeit wieder hereingeholt werden.

5.2.1.2 Datensegmente

Die Betrachtung der Kompressionsrate der beiden Datensegmente (*showimg ohne geöffnetes Bild* (Abb. A.2) und *showimg mit geöffnetem Bild* (Abb. A.3)) zeigt, dass in diesem Fall die Datensegmente wesentlich besser komprimiert werden können, als das Codesegment (maximal 23,5 % statt 52,4 %).

Die Abbildungen A.2 und A.3 zeigen, dass die Blockgröße die Kompressionsrate des Datensegments beeinflusst. Bei diesen Datensegmenten ist es also so, dass je größer die Blockgröße, desto besser das Kompressionsverhältnis. Für die LZ-Algorithmen trifft dasselbe zu, wie im Codesegment (effizientere Nutzung des Lempel bei größeren Blockgrößen).

Offenbar gibt es regelmäßige Muster in den Datensegmenten, die von den WK-Algorithmen mit den 16-elementigen Wörterbüchern gut erfasst werden. Die Erhöhung der Blockgröße führt gleichzeitig zu einer Verringerung der Größe des Miss-Bereichs, da das Wörterbuch weniger häufig komplett neu aufgebaut werden muss.

Das Datensegment von *showimg mit geladenem Bild* ist mehr als doppelt so groß wie das von *showimg ohne Bild*. Das bedeutet, dass der Größenunterschied in etwa der Größe des Bildes entsprechen muss. Interessant ist hier, dass die Kompressionsrate höher ist, dass also weniger komprimiert werden kann. Dies ist vor allem bei großen Blockgrößen der Fall.

Hierbei kommt zum Tragen, dass die geladene Grafik ein JPEG-Bild ist, dass schon stark komprimiert ist und daher die Kompressionsalgorithmen auch nichts mehr ausrichten können. Durch die Größe des Bildes wird auch der Effekt der fallenden Kompressionsrate bei steigender Blockgröße, der im anderen Fall deutlich sichtbar ist, verringert.

Durch die höhere Kompressionsrate bedingt, sinkt im Vergleich zum Codesegment auch der Leistungsbedarf pro MB, da weniger Änderungen am Wörterbuch vorgenommen werden müssen, bzw. die Übereinstimmungen länger sind und somit weniger Suchoperationen im Lempel vorgenommen werden müssen.

Wieder lässt sich gut beobachten, dass der Leistungsbedarf pro MB durch die Erhöhung der Speicher-Taktrate stark sinkt (im Schnitt ca. 30 %). Auch die Unabhängigkeit des Leistungsbedarfs von der CPU-Geschwindigkeit ist wieder ersichtlich.

Die schlechtere Kompression zeigt sich am leicht erhöhten Leistungsbedarf in Abbildung A.9 (*showimg mit geöffnetem Bild*). Hier werden im Schnitt 0,03 J / MB benötigt als in Abbildung A.8 (*showimg ohne Bild*).

Dasselbe gilt auch für die Kompressionsgeschwindigkeit (sichtbar in Abb. A.5 (*showimg ohne Bild*) und Abb. A.6 (*showimg mit Bild*)). Die schon beim Leistungsbedarf beobachtete Verbesserung der Werte lässt sich auch hier in einer Steigerung der MB pro Sekunde ablesen.

Auch hier besteht wieder das Problem, dass die Auswahl des Algorithmus vom Szenario abhängt. Soll eine niedrige Kompressionsrate erreicht werden, so ist **LZO** der Algorithmus der Wahl. Soll jedoch für die Kompression möglichst wenig Leistung aufgewendet werden, so ist **WKdm** zu verwenden.

5.2.1.3 Fazit

Für die Anwendungsklasse Bildbearbeitung gibt es keinen eindeutigen Favoriten unter den Algorithmen.

Legt man mehr Wert auf die Kompressionsgeschwindigkeit, so ist **WKdm** der bessere Algorithmus. Ist die Kompressionsrate wichtiger als die Geschwindigkeit, so ist **LZO** der effizientere Algorithmus.

Wenn man das Code- und die Datensegmente mit **WKdm** bzw. **LZO** bei 400 MHz und einer Blcokgröße von 64 KB komprimiert, wird die in den Tabellen 5.3 und 5.4 gezeigte Einsparung erreicht.

Die dafür benötigten Kosten in Form von Leistung und Zeit werden in den Tabellen 5.5 und 5.6 aufgelistet.

WKdm	Segment		
	Code	Daten	
		leer	mit Daten
Unkomprimiert	7,62 MB	1,17 MB	2,42 MB
Komprimiert	6,43 MB	0,38 MB	1,51 MB
Einsparung	1,19 MB	0,79 MB	0,91 MB

Tabelle 5.3: Originalgröße und mit WKdm komprimierte Größe der Segmente

LZO	Segment		
	Code	Daten	
		leer	mit Daten
Unkomprimiert	7,62 MB	1,17 MB	2,42 MB
Komprimiert	3,99 MB	0,27 MB	0,65 MB
Einsparung	3,63 MB	0,9 MB	1,77 MB

Tabelle 5.4: Originalgröße und mit LZO komprimierte Größe der Segmente

Segment	Leistung	Zeit
Codesegment	1,07 J	1,17 s
Datensegment leer	0,13 J	0,14 s
Datensegment mit Grafik	0,29 J	0,31 s

Tabelle 5.5: Kosten der Kompression mit WKdm

Segment	Leistung	Zeit
Codesegment	3,12 J	3,31 s
Datensegment leer	0,28 J	0,32 s
Datensegment mit Grafik	0,63 J	0,70 s

Tabelle 5.6: Kosten der Kompression mit LZO

5.2.2 Textverarbeitung - textedit

Von *textedit* wurden drei Dumps erstellt. Einmal das leere Programm, einmal mit einem kurzen Text (unter 100 Zeilen) und einmal mit einem langen Text (über 1000 Zeilen). Dabei wurden die in Tabelle 5.7 gezeigten Größen für die einzelnen unkomprimierten Segmente ermittelt. Dass das Codesegment von *textedit* genauso groß ist, wie das von *showimg* kommt daher, dass beide über ein Multicall-Binary[3] aufgerufen werden.

	Segment			
	Code	Daten		
		leer	einfach	komplex
textedit	7,62 MB	1,12 MB	1,20 MB	1,21 MB

Tabelle 5.7: Originalgröße der Segmente

Dadurch unterscheidet sich das Codesegment der beiden Anwendungen nur in einer einzigen Bibliothek (libshowimg.so.1.0.0 bzw. libtextedit.so.1.0.0).

5.2.2.1 Codesegment

Sämtliche gemessenen Daten stimmen mit den Daten des Codesegments von *showimg* überein. Daher wird das Codesegment hier nicht weiter betrachtet.

5.2.2.2 Datensegmente

Die Kompressionsgeschwindigkeit und der Energieverbrauch bei der Kompression der drei Datensegmente sind identisch zu denen bei der Kompression des Datensegments von *showimg ohne Datei*. Die Kompressionsrate des Datensegments von *textedit ohne Datei* ist äquivalent zu der des Datensegments von *showimg ohne Datei*.

Die Kompressionsrate der beiden Datensegmente mit offenen Dateien ist um 1-4 % (je nach Blockgröße) schlechter als die des Datensegments ohne Datei (siehe Abb. A.10 bzw. A.11).

Eigentlich wäre eine Verbesserung der Kompressionsrate mit längeren geladenem Text zu erwarten gewesen, da ein normaler Text aus vielen Wiederholungen von gleichen Buchstaben (Silben, Wörtern) besteht.

Diese Verschlechterung kann dadurch auftreten, dass die Textdaten nicht als einzelne Zeichen gespeichert werden, sondern in einer Binär-Form mit zusätzlichen Daten pro Zeichen (z. B. Größe, Schriftart, etc.). Alternativ könnte der Text auch in einer komprimierten Form im Speicher abgelegt sein, was aber eher unwahrscheinlich ist, da er dann laufend komprimiert und dekomprimiert werden müsste.

[3] Eine Multi-Call Binary ist ein Programm, das mehrere Programme in sich vereint. Beim Start wird argv[0] (der Name mit dem das Programm gestartet wird) geprüft und die entsprechende Funktionalität aufgerufen. Durch dieses Vorgehen kann man sehr viel Platz sparen, da Bibliotheken nur einmal gegen das Multi-Call Binary gelinkt werden müssen und nicht wie üblicherweise gegen die Binaries der einzelnen Programme.
Ein bekanntes Beispiel ist die BusyBox.

5.2.2.3 Fazit

Wie auch schon bei *showimg* sind **WKdm** und **LZO** die am besten geeigneten Algorithmen.

Die Tabellen 5.8, 5.9, 5.10 und 5.11 zeigen die Einsparung in MB und die Kosten in Joule. Die Werte stammen aus den Daten der Messung mit einer CPU-Geschwindigkeit von 400 MHz und einer Blockgröße von 64 KB.

WKdm	Code	Segment Daten		
		leer	mit kurzem Text	mit langem Text
Unkomprimiert	7,62 MB	1,12 MB	1,20 MB	1,21 MB
Komprimiert	6,43 MB	0,36 MB	0,46 MB	0,46 MB
Einsparung	1,19 MB	0,76 MB	0,74 MB	0,75 MB

Tabelle 5.8: Originalgröße und mit WKdm komprimierte Größe der Segmente

LZO	Code	Segment Daten		
		leer	mit kurzem Text	mit langem Text
Unkomprimiert	7,62 MB	1,12 MB	1,20 MB	1,21 MB
Komprimiert	3,99 MB	0,30 MB	0,33 MB	0,33 MB
Einsparung	3,63 MB	0,82 MB	0,87 MB	0,88 MB

Tabelle 5.9: Originalgröße und mit LZO komprimierte Größe der Segmente

Segment	Leistung	Zeit
Codesegment	1,07 J	1,17 s
Datensegment leer	0,12 J	0,14 s
Datensegment mit kurzem Text	0,13 J	0,15 s
Datensegment mit langem Text	0,13 J	0,15 s

Tabelle 5.10: Kosten der Kompression mit WKdm

Segment	Leistung	Zeit
Codesegment	3,12 J	3,31 s
Datensegment leer	0,27 J	0,31 s
Datensegment mit kurzem Text	0,29 J	0,33 s
Datensegment mit langem Text	0,29 J	0,33 s

Tabelle 5.11: Kosten der Kompression mit LZO

5.2.3 Audioanwendung - madplay

Von der Anwendung madplay wurden nur ein Speicherdump erstellt, da der Start eines Audioplayers von Kommandozeile aus normalerweise nicht ohne Musikstück erfolgt.

	Segment	
	Code	Daten
madplay	1,87 MB	4,38 MB

Tabelle 5.12: Originalgröße der Segmente

5.2.3.1 Codesegment

Die Kompressionsrate des Codesegments ist, wie in Abbildung A.12 dargestellt, stark von der Blockgröße abhängig. Der Unterschied zwischen den LZ-Algorithmen und den WK-Algorithmen ist nicht so stark, wie es bei *showimg* der Fall war.

Die Kompressionsrate der LZ-Algorithmen ist trotz Verbesserung mit steigender Blockgröße immer schlechter als die vergleichbare Kompressionsrate die bei *showimg*.

Die Kompressionsrate der WK-Algorithmen ist zuerst 3 % schlechter als die bei *showimg*, wird aber mit steigender Blockgröße besser und ist bei 64 KB Blockgröße sogar besser als die vergleichbare Kompressionsrate bei *showimg*.

Bei der Betrachtung der Kompressionsgeschwindigkeit präsentiert sich ein sehr ähnliches Bild wie beim Codesegment von *showimg*. Der einzige Unterschied ist, dass die LZ-Algorithmen im Schnitt 0,2 MB / s langsamer sind (siehe Abb. A.14).

Dasselbe zeigt sich auch beim Leistungsbedarf der Algorithmen. Die zur Kompression mit den LZ-Algorithmen benötigte Leistung ist um ca. 0,5 J / MB höher als sie es bei *showimg* ist. Der Leistungsbedarf der WK-Algorithmen hingegen ist in etwa gleich (siehe Abb. A.16).

In Anbetracht der geringen Größe des Codesegments und der hohen Kompressionsrate bzw. dem hohen Energieverbrauch wäre es für diese Anwendung am besten, das Codesegment **nicht** zu komprimieren.

5.2.3.2 Datensegment

Der erste Blick auf das Kompressionsratendiagramm A.13 bestätigt die Vermutung, dass das Datensegment des Audioplayers nicht komprimierbar ist.

Da die Audiodatei über 90 % des Datensegments ausmacht, bestimmt auch deren Kompressionsrate die Kompressionsrate des gesamten Segments. Da mp3-Dateien schon stark komprimiert sind, war zu erwarten, dass sich das komplette Datensegment nur sehr schlecht komprimieren lässt.

Das Kompressionsraten von über 100 % erreicht werden, liegt daran, dass bei der mp3-Datei der Overhead durch die Kompressionsalgorithmen größer ist, als die Einsparung durch die Kompression. Dadurch wird bei den WK-Algorithmen die Datei vergrößert statt verkleinert. Bei den LZ-Algorithmen ist der Overhead kleiner und dadurch wird eine sehr geringe

Kompression von maximal 6 % erreichbar.

Durch die schlechte Kompression ist auch der Leistungsbedarf pro MB höher als bei *showimg*, da wesentlich mehr Daten geschrieben werden müssen, als bei einer guten Kompression.

Aus Abbildung A.17 ist ersichtlich, dass die LZ-Algorithmen, um überhaupt eine Kompression zu erreichen, sehr viel Energie aufwenden.

Auch der Leistungsbedarf von WK4x4 nimmt zu. WHdm und WKdm dagegen benötigen in etwa genauso viel Leistung pro MB wie bei der Kompression von showimg.

Entsprechend der Zunahme des Leistungsbedarfs verringert sich die Kompressionsgeschwindigkeit pro MB (siehe Abb. A.15).

Während bei den LZ-Algorithmen die Geschwindigkeit verglichen mit showimg (Abb. A.5) um ca. 50 % abnimmt, verdoppelt sich der Leistungsbedarf bei LZRW1. Bei LZO ist der Bedarf sogar bis zu viermal so hoch.

Der Leistungsbedarf von WHdm und WKdm bleibt in etwa konstant. Der von WK4x4 nimmt um ca. 50 % zu, während die Geschwindigkeit um ca. 40 % abnimmt.

5.2.3.3 Fazit

Bei *madplay* ist es am besten, nur das Codesegment zu komprimieren.

Für das Datensegment ist **keine** Kompression am besten. Hier würde nur Energie und Zeit aufgewendet werden, um im Endeffekt das Segment in Originalgröße zu erhalten.

Die Tabellen 5.13, 5.14, 5.15 und 5.16 beinhalten dementsprechend nur die Werte für das Codesegment.

	Segment	
WKdm	Code	Daten
Unkomprimiert	1,87 MB	4,38 MB
Komprimiert	1,53 MB	—
Einsparung	0,34 MB	—

Tabelle 5.13: Originalgröße und mit WKdm komprimierte Größe der Segmente

	Segment	
LZO	Code	Daten
Unkomprimiert	1,87 MB	4,38 MB
Komprimiert	1,13 MB	—
Einsparung	0,74 MB	—

Tabelle 5.14: Originalgröße und mit LZO komprimierte Größe der Segmente

Segment	Leistung	Zeit
Codesegment	0,26 J	0,28 s

Tabelle 5.15: Kosten der Kompression mit WKdm

Segment	Leistung	Zeit
Codesegment	0,92 J	0,99 s

Tabelle 5.16: Kosten der Kompression mit LZO

5.2.4 Netzwerkanwendung - konqueror

Konqueror wurde dreimal gedumpt, einmal nur das *leere Programm*, einmal mit einer *einfachen Webseite*, mit einer kleinen Grafik, ohne Flash oder Javascript und einmal mit einer *komplexen Seite*, die sowohl Grafiken als auch Flash und Javascript enthält. Da Konqueror mit zwei Threads läuft, wurden immer beide ausgegeben. In Tabelle 5.17 werden die Größen der Segmente aufgelistet.

	Segment			
	Code	Daten		
		leer	einfach	komplex
konqueror Thread 1	10,95 MB	1,31 MB	1,96 MB	3,23 MB
konqueror Thread 2	10,95 MB	0,93 MB	0,93 MB	0,93 MB

Tabelle 5.17: Originalgröße der Segmente

5.2.4.1 Codesegment

Ein Vergleich der Codesegmente hat ergeben, dass alle Codesegmente identisch sind. Daher wird hier wieder nur ein Codesegment untersucht.

Die Kompressionsrate des Codesegments von *konqueror* entspricht der Kompressionsrate des Codesegments von *showimg*. Da auch der Leistungsbedarf und die Kompressionsgeschwindigkeit den Werten des Codesegments von *showimg* entsprechen sei hier wieder einmal auf die Analyse des Codesegments von *showimg* in Abschnitt 5.2.1.1 verwiesen.

5.2.4.2 Datensegmente

Die Daten der Datensegmente des zweiten Threads unterscheiden sich nur wenig. Offensichtlich ist dieser Thread unabhängig von den angezeigten Daten des ersten Threads.

Die größe Ähnlichkeit der Daten führt auch dazu, dass die Kompressionsraten, der Leistungsbedarf und die Kompressionsgeschwindigkeit des zweiten Threads übereinstimmen. Zudem stimmen die Auswertungen auch mit denen des Datensegments von *showimg ohne geöffnetes Bild* (siehe Abschnitt 5.2.1.2) überein.

Auch die Kompressionsrate des Datensegments von *konqueror ohne geladene Webseite* (siehe Abb. A.18) entspricht der des Datensegments von *showimg ohne geöffnetes Bild*. Jedoch steigt die Kompressionsrate mit der Komplexität der angezeigten Webseite. So ist die Kompression des Datensegments von *konqueror mit einfacher Webseite* um 4-7 % schlechter (siehe Abb. A.19). Die des Datensegments mit *komplexer Seite* sogar um über 13 % (siehe Abb. A.20).

Diese geringere Kompression ist darauf zurückzuführen, dass die Webseiten Grafiken enthalten die nur wenig komprimiert werden können.

Entsprechend der Abnahme des Kompressionsverhältnisses nimmt auch die Kompressionsgeschwindigkeit ab. (siehe Abb. A.21, A.22 und A.23) und der Leistungsbedarf pro MB zu (siehe Abb. A.24, A.25 und A.26).

Das Verhältnis der Kompressionsalgorithmen zueinander bleibt aber gleich.

5.2.5 Fazit

Trotz der Zunahme der Kompressionsrate und der Abnahme der Geschwindigkeit ist wieder **WKdm** der Algorithmus mit der höchsten Kompressionsgeschwindigkeit und **LZO** derjenige mit dem besten Kompressionsverhältnis.

Die Tabellen 5.18 und 5.19 zeigen den Kompressionserfolg bei der Kompression mit **WKdm** und **LZO**. In den Tabellen 5.20 und 5.21 sind die dafür benötigten Kosten gelistet.

	WKdm	Segment			
		Code	Daten		
			leer	einfach	komplex
Thread 1	unkomprimiert	10,95 MB	1,31 MB	1,96 MB	3,23 MB
Thread 1	komprimiert	9,72 MB	0,42 MB	0,77 MB	1,85 MB
Thread 1	Einsparung	1,23 MB	0,89 MB	1,19 MB	1,38 MB
Thread 2	unkomprimiert	10,95 MB	0,93 MB	0,93 MB	0,93 MB
Thread 2	komprimiert	9,72 MB	0,30 MB	0,30 MB	0,30 MB
Thread 2	Einsparung	1,23 MB	0,63 MB	0,63 MB	0,63 MB
gesamt	unkomprimiert	21,90 MB	2,24 MB	2,89 MB	4,16 MB
gesamt	komprimiert	19,45 MB	0,71 MB	1,07 MB	2,15 MB
gesamt	Einsparung	2,45 MB	1,53 MB	1,82 MB	2,01 MB

Tabelle 5.18: Originalgröße und mit WKdm komprimierte Größe der Segmente

	LZO	Segment			
		Code	Daten		
			leer	einfach	komplex
Thread 1	unkomprimiert	10,95 MB	1,31 MB	1,96 MB	3,23 MB
Thread 1	komprimiert	5,74 MB	0,31 MB	0,58 MB	1,33 MB
Thread 1	Einsparung	5,21 MB	1,00 MB	1,38 MB	1,90 MB
Thread 2	unkomprimiert	10,95 MB	0,93 MB	0,93 MB	0,93 MB
Thread 2	komprimiert	5,74 MB	0,31 MB	0,31 MB	0,31 MB
Thread 2	Einsparung	5,21 MB	1,00 MB	1,00 MB	1,00 MB
gesamt	unkomprimiert	21,90 MB	2,24 MB	2,89 MB	4,16 MB
gesamt	komprimiert	11,48 MB	0,62 MB	0,89 MB	1,64 MB
gesamt	Einsparung	10,42 MB	2,00 MB	2,38 MB	2,90 MB

Tabelle 5.19: Originalgröße und mit LZO komprimierte Größe der Segmente

	Segment	Leistung	Zeit
	Codesegment	1,72 J	1,64 s
Thread 1	Datensegment leer	0,14 J	0,16 s
Thread 1	Datensegment einfache Seite	0,25 J	0,28 s
Thread 1	Datensegment komplexe Seite	0,76 J	0,48 s
Thread 2	Datensegment leer	0,10 J	0,11 s
Thread 2	Datensegment einfache Seite	0,10 J	0,11 s
Thread 2	Datensegment komplexe Seite	0,10 J	0,11 s
gesamt	Codesegmente	3,44 J	3,28 s
gesamt	Datensegmente leer	0,24 J	0,17 s
gesamt	Datensegmente einfache Seite	0,35 J	0,39 s
gesamt	Datensegmente komplexe Seite	0,86 J	0,59 s

Tabelle 5.20: Kosten der Kompression mit WKdm

	Segment	Leistung	Zeit
	Codesegment	4,49 J	4,76 s
Thread 1	Datensegment leer	0,31 J	0,36 s
Thread 1	Datensegment einfache Seite	0,51 J	0,57 s
Thread 1	Datensegment komplexe Seite	1,07 J	1,19 s
Thread 2	Datensegment leer	0,22 J	0,26 s
Thread 2	Datensegment einfache Seite	0,22 J	0,26 s
Thread 2	Datensegment komplexe Seite	0,22 J	0,26 s
gesamt	Codesegmente	8,98 J	9,52 s
gesamt	Datensegmente leer	0,53 J	0,62 s
gesamt	Datensegmente einfache Seite	0,73 J	0,83 s
gesamt	Datensegmente komplexe Seite	1,29 J	1,45 s

Tabelle 5.21: Kosten der Kompression mit LZO

5.3 Fazit

Eine eindeutige Entscheidung für einen Algorithmus gibt es nicht, da diese Entscheidung vom jeweiligen Anwendungsszenario abhängt.

Wird die Anwendung lange deaktiviert, so ist eine niedrigere Kompressionsrate trotz des hohen Leistungsbedarfs bei der Kompression sparsamer, da während der Zeit in der die Anwendung deaktiviert ist mehr Strom gespart werden kann. Hier sollte also **LZO** verwendet werden.

Wird die Anwendung aber nur kurz deaktiviert, so muss die Kompression sehr sparsam erfolgen, da die Zeit, die zum Sparen von Energie zur Verfügung steht, nur kurz ist. In diesem Fall sollte **WKdm** zum Einsatz kommen.

Die Untersuchung der Code- und Datensegmente hat ergeben, dass diese sich stark unterscheiden. Dies ist jedoch nur an der Kompressionsrate zu sehen. Diese ist bei Codesegmenten im Schnitt ca. 30 Prozentpunkte höher als bei Datensegmenten. Allerdings werden die besten Kompessionsergebnisse mit den selben Algorithmen erzielt, die auch im Datensegmente die besten Ergebnisse erzielen.

Somit ist eine Unterscheidung der Segmenttypen nicht unbedingt nötig, kann aber zu besseren Kompressionsergebnissen führen, wenn z. B. wie in *madplay* das Datensegment nicht komprimierbar ist, das Codesegment aber schon.

Die Betrachtung der Anwendungsklassen hat ergeben, dass bis auf die Audioanwendung alle Klassen gute Kompressionsergebnisse erzielen. Somit sollte zukünftig bei der Kompression entweder die komplette Klasse der Audioanwendungen, oder aber nur die Audiofiles in den Datensegmenten dieser Klasse von der Kompression ausgeschlossen werden.

Bei Betrachtung der CPU-Geschwindigkeit hat sich herausgestellt, dass der geringste Leistungsbedarf bei der höchsten CPU-Geschwindigkeit (400 MHz) erzielt wird. Auch bei der Blockgröße hat sich herausgestellt, dass für die besten Resultate ein möglichst großer Wert (64 KB) gewählt werden soll.

Tabelle 5.22 gibt für eine ausgewählte Gruppe von Anwendungen einen Überblick über den Platzbedarf vor und nach der Kompression und die dafür benötigte Leistung. Aus ihr ist ersichtlich, dass das Kompressionsverhältnis von LZO ungefähr 2:4 ist, während es bei WKdm ungefähr 3:4 ist. Außerdem ist zu sehen, dass LZO ungefähr 2,5mal so viel Energie benötigt wie WKdm um das bessere Kompressionsverhältnis zu erreichen.

Anwendung	Segment	Größe Original	Größe LZO	Größe WKdm	Leistung LZO	Leistung WKdm
showimg mit Bild	Code	7,62 MB	3,99 MB	6,43 MB	3,12 J	1,07 J
showimg mit Bild	Daten	2,42 MB	0,27 MB	0,38 MB	0,63 J	0,29 J
textedit einfach	Code	7,62 MB	3,99 MB	6,43 MB	3,12 J	1,07 J
textedit einfach	Daten	1,20 MB	1,33 MB	0,36 MB	0,29 J	0,13 J
konqueror komplex T1	Code	10,95 MB	5,74 MB	9,72 MB	4,49 J	1,72 J
konqueror komplex T1	Daten	3,23 MB	1,33 MB	1,85 MB	1,07 J	0,76 J
konqueror komplex T2	Code	10,95 MB	5,74 MB	9,72 MB	4,49 J	1,72 J
konqueror komplex T2	Daten	0,93 MB	0,31 MB	0,30 MB	0,22 J	0,10 J
Summe		44,92 MB	22,70 MB	35,19 MB	17,43 J	6,86 J

Tabelle 5.22: Kompressionserfolg und -aufwand für eine ausgewählte Gruppe von Anwendungen

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Da der Anteil des Hauptspeichers am Gesamtenergieverbrauch immer weiter steigt, besteht die Notwendigkeit diesen Anteil zu reduzieren. Dies kann dadurch erreicht werden, dass nicht benötigte Teile des Hauptspeichers deaktiviert werden. Die Kompression der aktuell nicht verwendeten Daten kann zu weiteren Einsparungen führen.

Die Kompression des Hauptspeichers diente bisher nur dem Zweck, mehr RAM zur Verfügung zu haben oder aber die Festplattenzugriffe zu reduzieren.

Diese Arbeit betrachtet nun die Kompression[1] unter Energiesparaspekten. Dabei wird der Leistungsbedarf, die Kompressionsgeschwindigkeit und die Kompressionsrate verschiedener Algorithmen verglichen.

Die Kompressionsalgorithmen werden mit unterschiedlichen CPU-Geschwindigkeiten auf den Hauptspeicher angewandt. Dabei wird der Hauptspeicher in verschiedene Anwendungsklassen unterteilt, und aus jeder ein repräsentatives Programm untersucht. Die Speicherbereiche der untersuchten Programme sind zusätzlich in Daten- und Codesegmente aufgeteilt, um eine Aussage darüber treffen zu können, ob sich für die Codesegmente andere Algorithmen als optimal herausstellen als für die Datensegmente. Die Segmente werden für die Kompression in gleich große Blöcke aufgeteilt, wobei mehrere Testreihen mit unterschiedlichen Blockgrößen erstellt werden.

Die Auswertung der gewonnenen Daten zeigt, dass sich bis auf die Datensegmente des Audioplayers alle Segmente komprimieren lassen. Dabei stellt sich heraus, dass die Codesegmente mit den hier verwendeten Algorithmen stets schlechter komprimiert werden können als die Datensegmente.

Dass der Audioplayer nicht komprimiert werden kann, liegt daran, dass der größte Teil des Datensegments aus der Audio-Datei besteht. Diese ist in diesem Fall eine mp3-Datei, die sich offensichtlich nicht mehr komprimieren lässt, da sie schon stark komprimiert ist.

Als optimale CPU-Geschwindigkeit hat sich die maximale Geschwindigkeit von 400 MHz herausgestellt. Dort ist im Vergleich zu anderen CPU-Geschwindigkeiten der Leistungsbedarf der Kompression am geringsten.

[1] Auch hier ist mit Kompression wieder Kompression plus Dekompression gemeint.

Auch für die Blockgröße ist der größte getestete Wert (64 KB) derjenige, der im Vergleich zu anderen Blockgrößen die besten Resultate erzielt.

Eine eindeutige Auswahl des „besten" Algorithmus kann nicht getroffen werden, da es Algorithmen gibt, mit denen ein gutes Kompressionsverhältnis erzielt werden kann, dafür aber sehr lange Laufzeiten und somit einen hohen Leistungsbedarf aufweisen. Der effizienteste Algorithmus dieser Klasse ist LZO. Auf der anderen Seite gibt es Algorithmen, deren Kompressionsverhältnis schlechter ist, die dafür aber durch eine sehr kurze Laufzeit und einen geringen Leistungsbedarf auffallen. Der beste Algorithmus aus dieser Klasse ist WKdm.

Um eine eindeutige Auswahl treffen zu können, müsste das Anwendungsszenario bekannt sein, für das das Gerät, dessen Hauptspeicher komprimiert werden soll, eingesetzt wird.

Wenn ein Prozess längere Zeit inaktiv ist, so eignet sich LZO besser als WKdm, da durch das bessere Kompressionsverhältnis über die Zeit mehr Energie eingespart wird als die zusätzliche Kompression kostet.

Wenn jedoch die Prozesse häufig umgeschaltet werden, die Inaktivitätsperiode also nur kurz ist, so ist WKdm vorzuziehen, da hier die Kosten für die Kompression wesentlich geringer sind, dafür allerdings auch weniger Energie gespart werden kann.

6.2 Ausblick

Eine Möglichkeit, die Kompression praktisch einsetzen zu können, wäre, den Scheduler zu modifizieren. Dieser sollte dann entscheiden, ob der zu verdrängende Prozess komprimiert werden soll oder nicht. Sollte eine positive Entscheidung getroffen werden, so ist der Prozessspeicher zu komprimieren und danach zu prüfen, ob der zu aktivierende Prozess komprimiert oder unkomprimiert vorliegt. Ist der Prozessspeicher komprimiert, so muss dieser vor dem Kontextwechsel dekomprimiert werden.

Die Entscheidung ob komprimiert werden soll, hängt davon ab, wie lange der Prozess voraussichtlich inaktiv bleibt. Ist die Periode zu kurz (Energieverbrauch der Kompression größer als Energieeinsparung durch Kompression, sollte der Prozessspeicher nicht komprimiert werden.

In einem System, in dem es sowohl Prozesse gibt, die lange inaktiv sind und solche, die nur kurz inaktiv sind, könnte ein Verfahren eingesetzt werden, das überprüft, wie lange der Prozess das letzte Mal inaktiv war, und dementsprechend den passenden Algorithmus für die nächste Kompression selektiert.

Auch sollte bei der Kompression darauf geachtet werden, ob das gerade zu komprimierende Segment Teil einer Bibliothek ist und wenn ja, wie viele andere Prozesse auf diese zugreifen. Eine Kompression lohnt nur dann, wenn sie nur von sehr wenigen Prozessen verwendet wird, da sie sonst bei jedem Prozesswechsel komprimiert und sofort wieder dekomprimiert werden würde.

Eine andere Möglichkeit wäre, die Daten erst dann zu komprimieren, wenn der Rechner gerade im Leerlauf ist. Dann wäre die Zeit, die für die Kompression benötigt würde nicht kritisch, dafür müsste die Dekompression wie in allen anderen Fällen auch, sehr schnell erfolgen.

Alternativ könnte auch der in [HPS03] vorgestellte Ansatz weiter verfolgt werden und direkt auf Seitenebene komprimiert werden. Dabei könnte beim Ablegen der Seite geprüft werden, ob sie aus dem Code- oder Datensegment stammt, und die Kompression dementsprechend erfolgen.

Ein weiterer Ansatz wäre, die Kompression direkt in der Hardware zu implementieren, so dass Codesegmente erst direkt vor der Ausführung dekomprimiert werden müssten. Dabei könnten Teile des Hauptspeichers explizit für Codesegmente reserviert werden und diese dort immer in komprimierter Form ablegt werden. Diese Segmente werden dann komprimiert zur CPU übertragen und dort dekomprimiert und ausgeführt.
Der Vorteil an dieser Methode ist, dass gleichzeitig ein Schutz vor Änderung am Codesegment integriert werden kann. Dies kann z. B. durch CRC-Summen für die komprimierten Daten geschehen. Auch kann damit verhindert werden, dass Code ausgeführt wird, der nicht im Codeteil des Hauptspeichers liegt.

Um die Kompression im Codesegment weiter zu verbessern, könnte ein neuer Algorithmus entwickelt werden, der sowohl ein statisches als auch ein dynamisches Wörterbuch enthält. Dabei sollte aber das statische Wörterbuch nicht jedes Mal neu erzeugt werden sondern fest einprogrammiert sein (z. B. die häufigsten Instruktionen).

Ein weiterer Punkt, den zu betrachten interessant wäre, wäre die Verwendung des Kompressionsalgorithmus whsd16(256) im Bereich der WLAN-Kompression. Dieser ist zwar bei der Kompression sehr langsam, bei der Dekompression ist er jedoch der schnellste. Dieses asymmetrische Verhalten lässt ihn für diesen Bereich geeignet erscheinen. Dort gibt es Ansätze, bei denen Daten auf dem Server komprimiert und auf dem Client (mobiler Rechner) dekomprimiert werden. Dabei wird darauf geachtet, dass beim Transport und auf dem Client so wenig Energie wie möglich verbraucht wird.

Anhang A

Anhang

Nachfolgend werden die Ergebnisse der Messungen präsentiert. In jedem Abschnitt wird eine Anwendung ausgewertet. Dabei werden zuerst die Kompressionsraten, danach die Kompressionsgeschwindigkeit und zuletzt der Leistungsbedarf dargestellt.

A.1 showimg

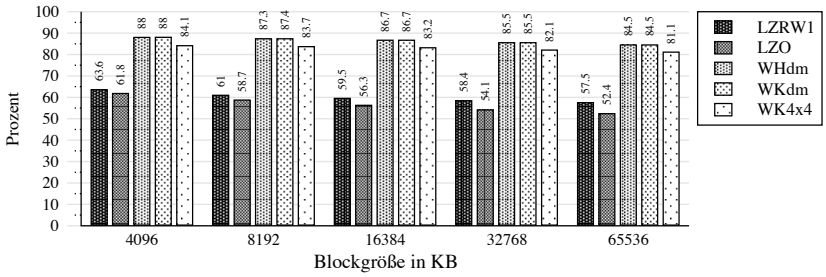

Abbildung A.1: showimg - Codesegment: Kompressionsrate des Segments

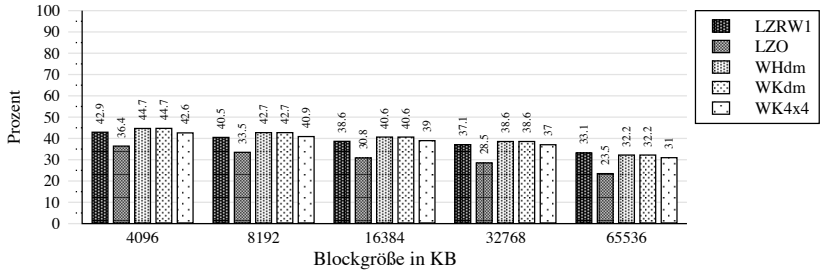

Abbildung A.2: showimg leer - Datensegment: Kompressionsrate des Segments

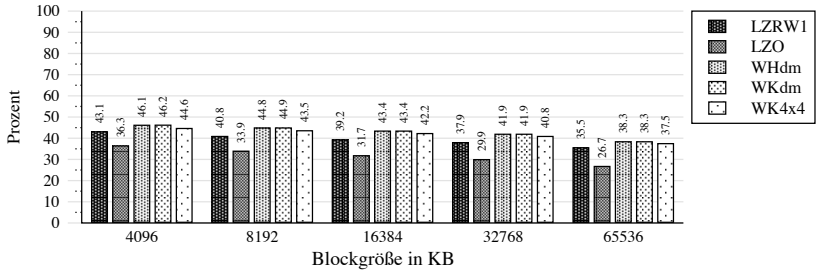

Abbildung A.3: showimg mit Bild - Datensegment: Kompressionsrate des Segments

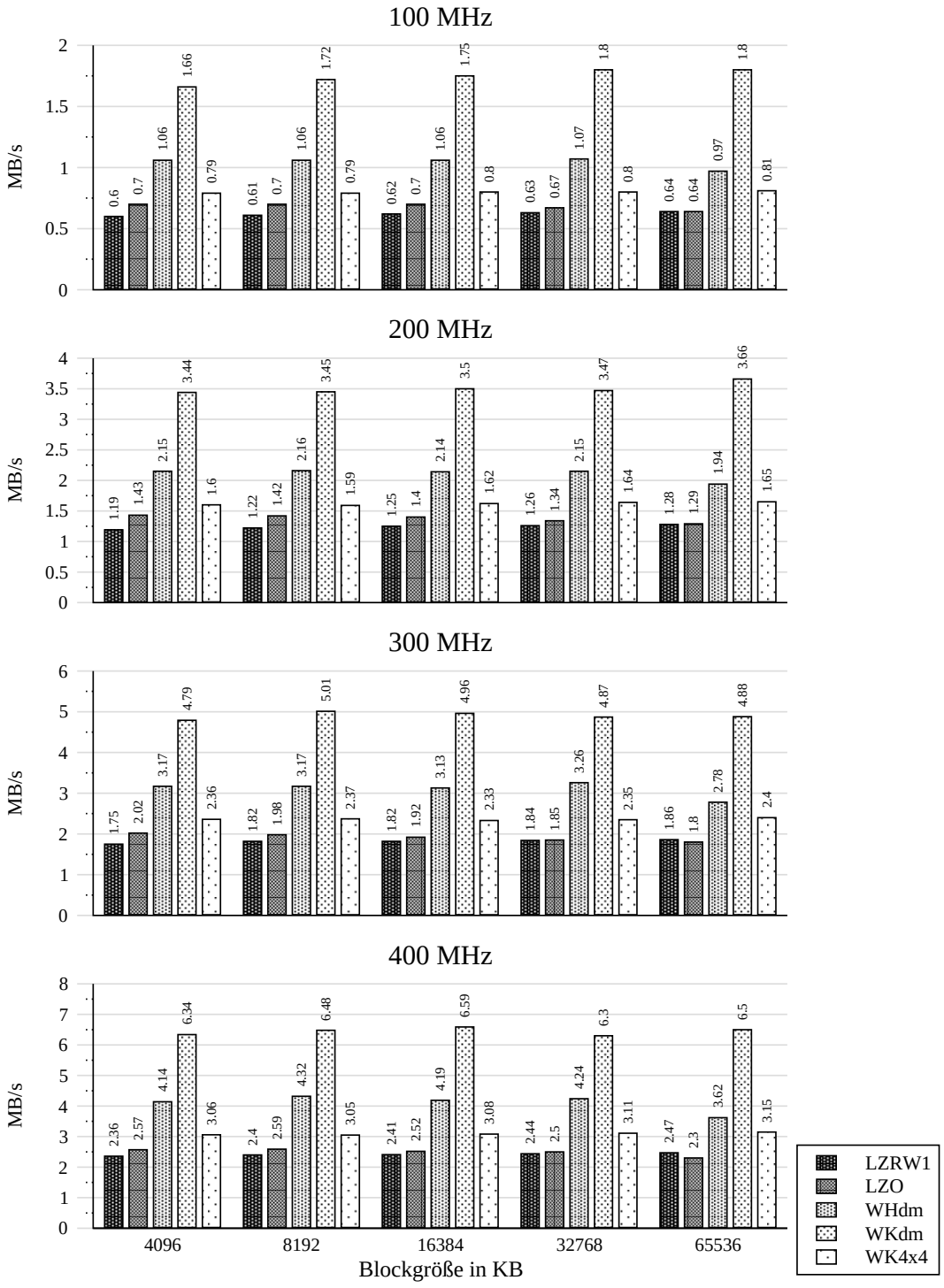

Abbildung A.4: showimg - Codesegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

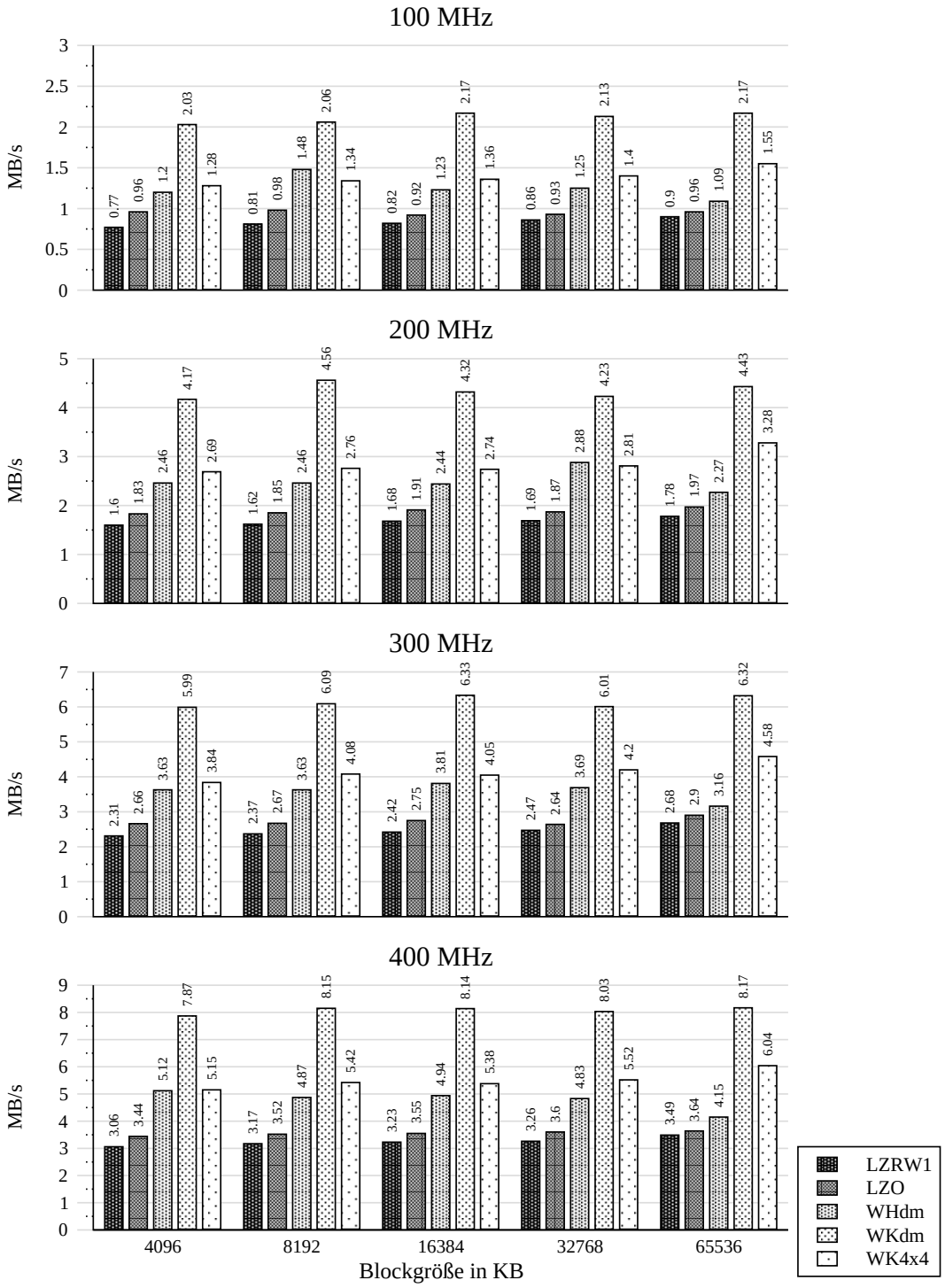

Abbildung A.5: showimg leer - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

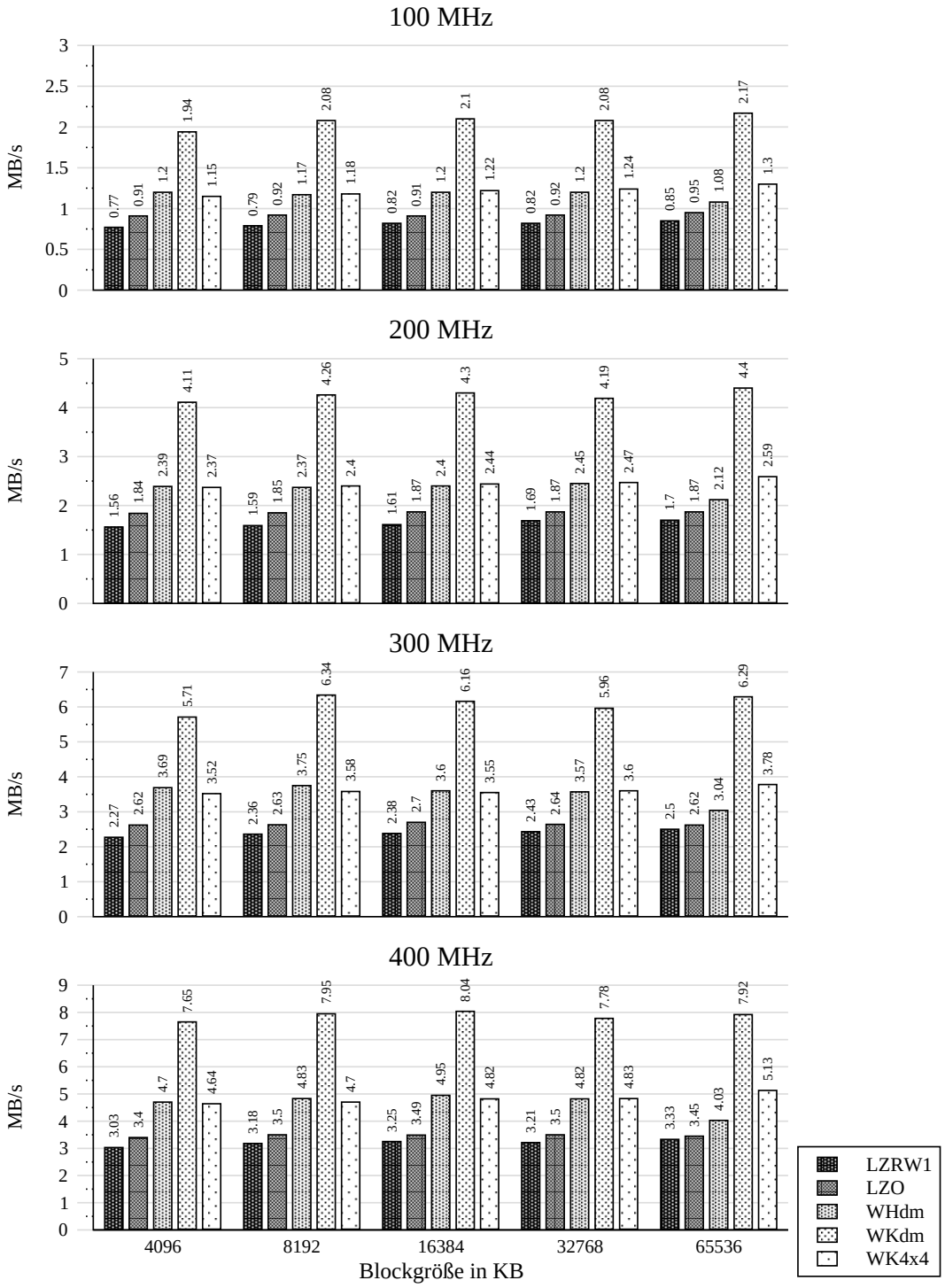

Abbildung A.6: showimg mit Bild - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

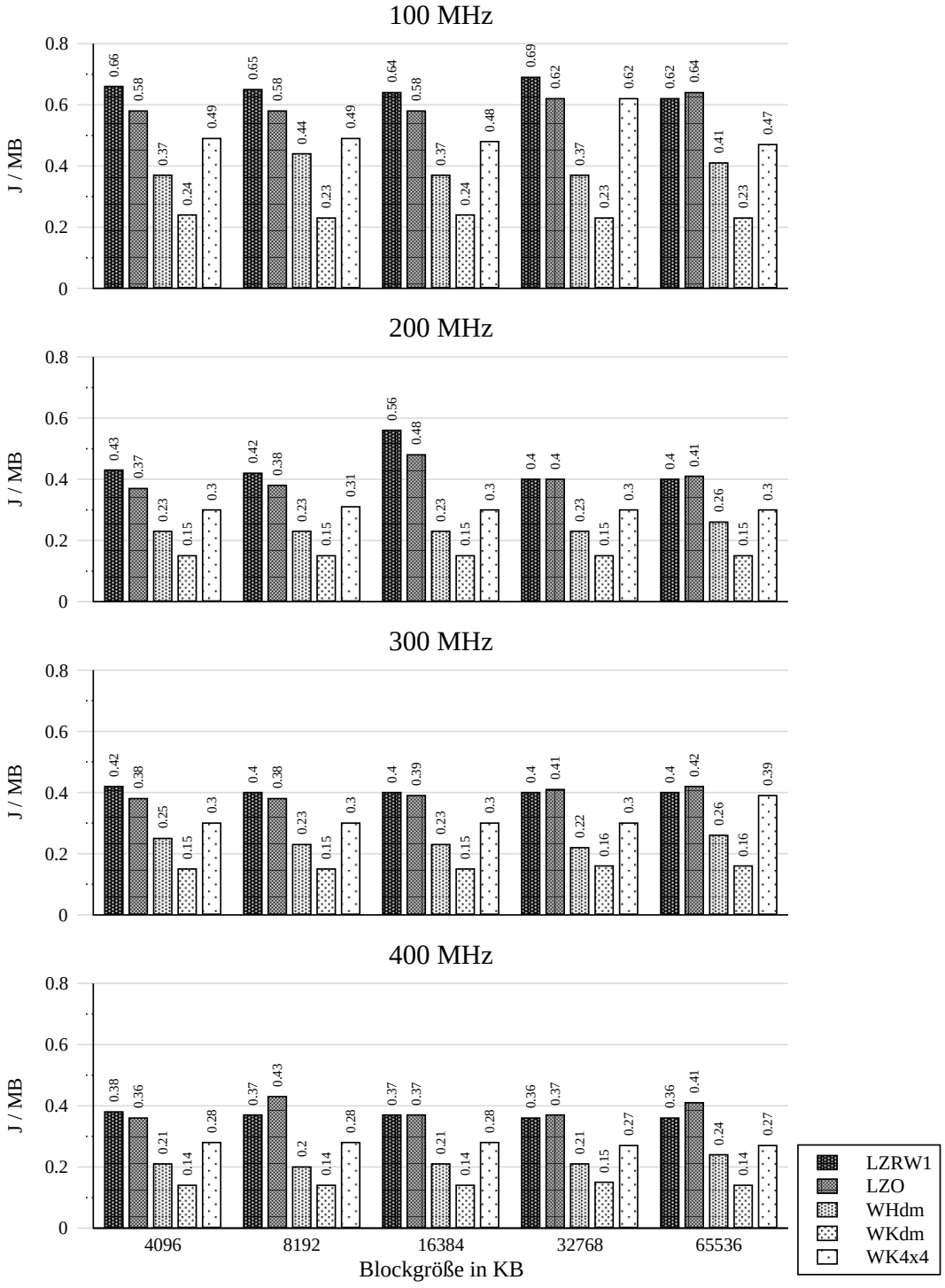

Abbildung A.7: showimg - Codesegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

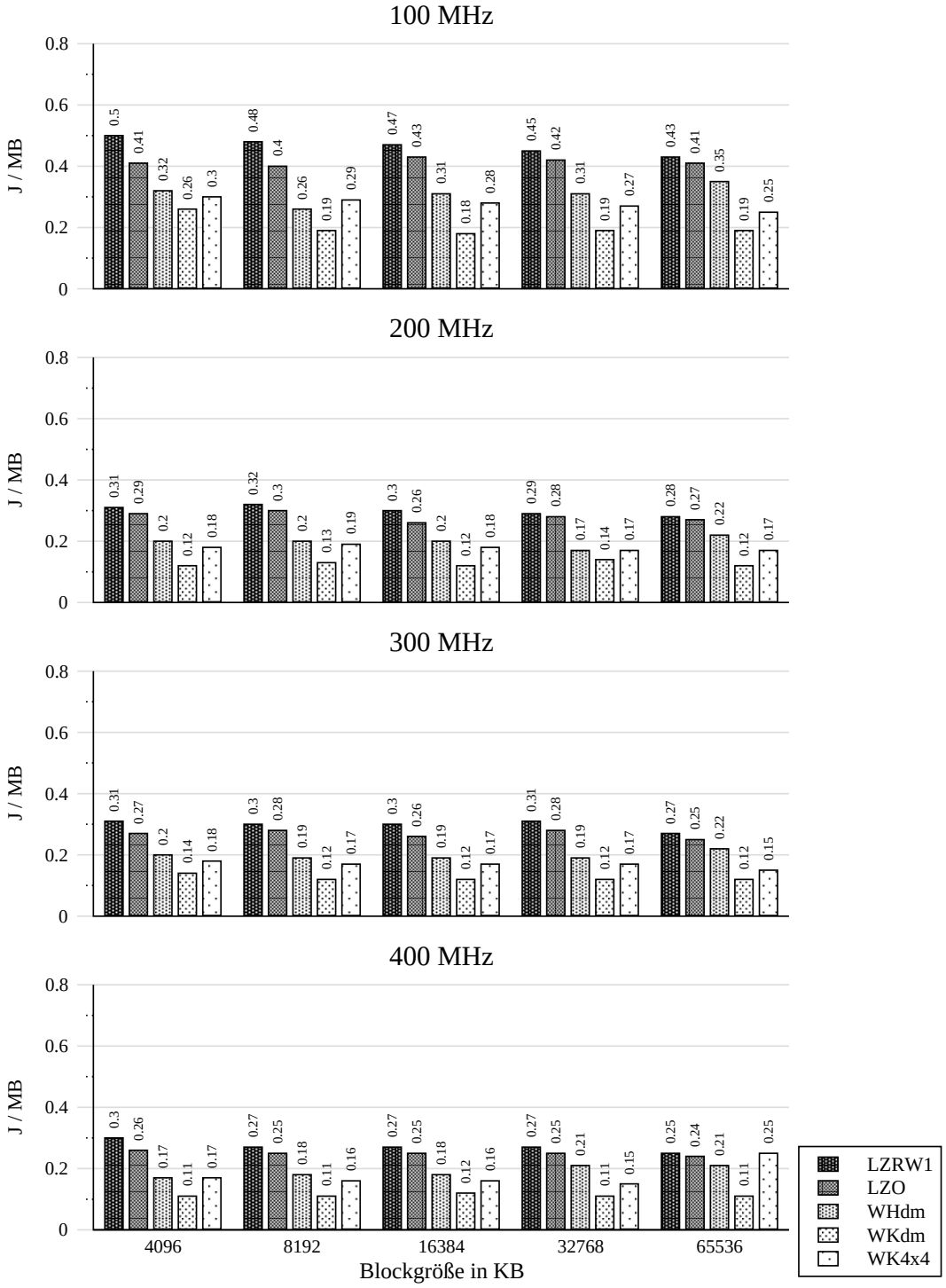

Abbildung A.8: showimg leer - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

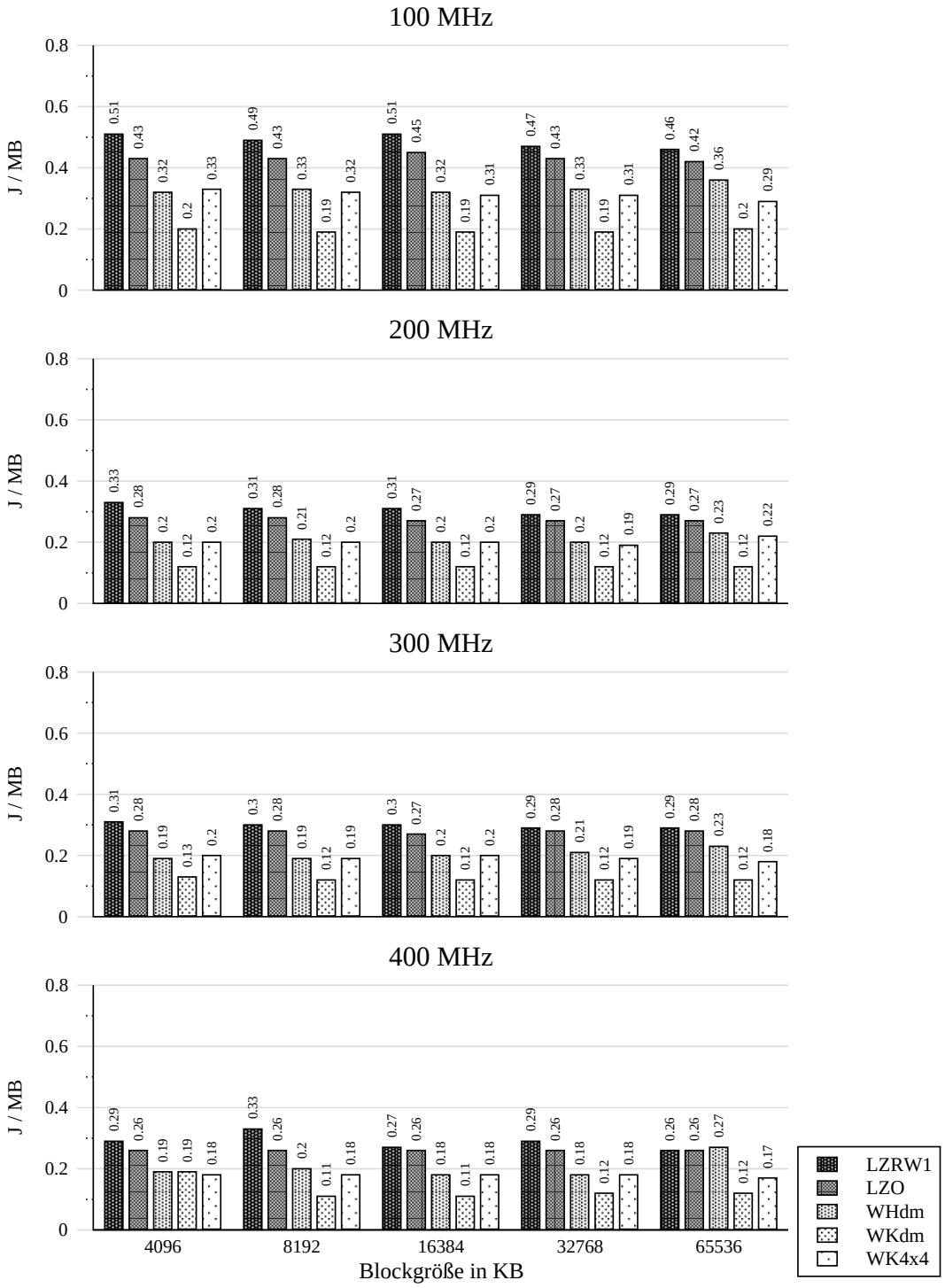

Abbildung A.9: showimg mit Bild - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

A.2 textedit

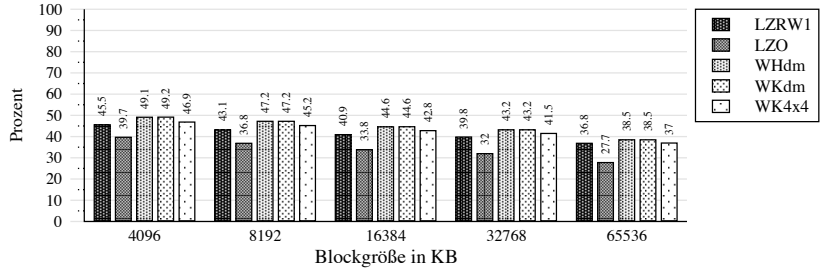

Abbildung A.10: textedit mit kurzem Text - Datensegment: Kompressionsrate des Segments

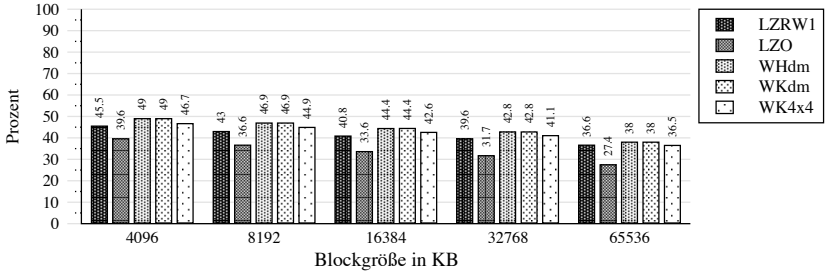

Abbildung A.11: textedit mit langem Text - Datensegment: Kompressionsrate des Segments

A.3 madplay

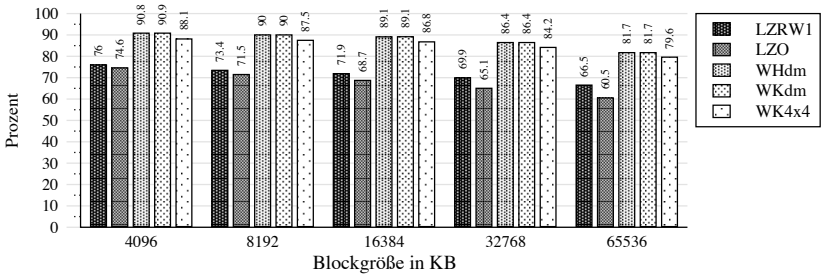

Abbildung A.12: madplay mit Audiodatei - Codesegment: Kompressionsrate des Segments

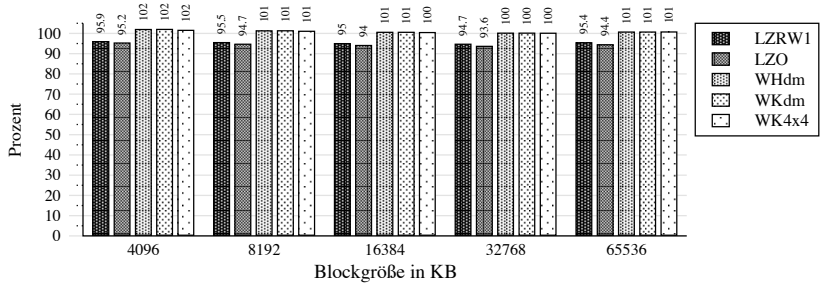

Abbildung A.13: madplay mit Audiodatei - Datensegment: Kompressionsrate des Segments

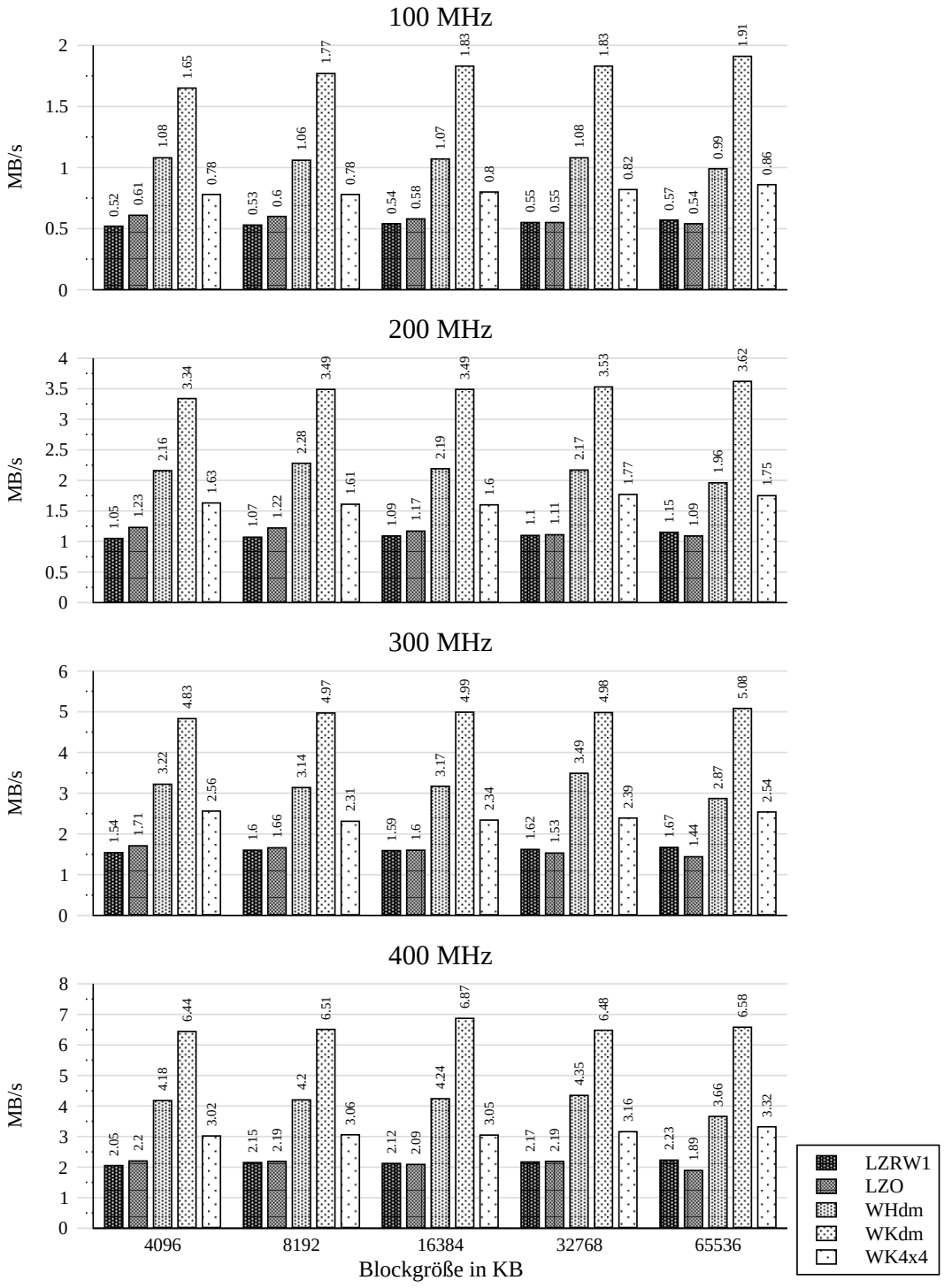

Abbildung A.14: madplay mit Audiodatei - Codesegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

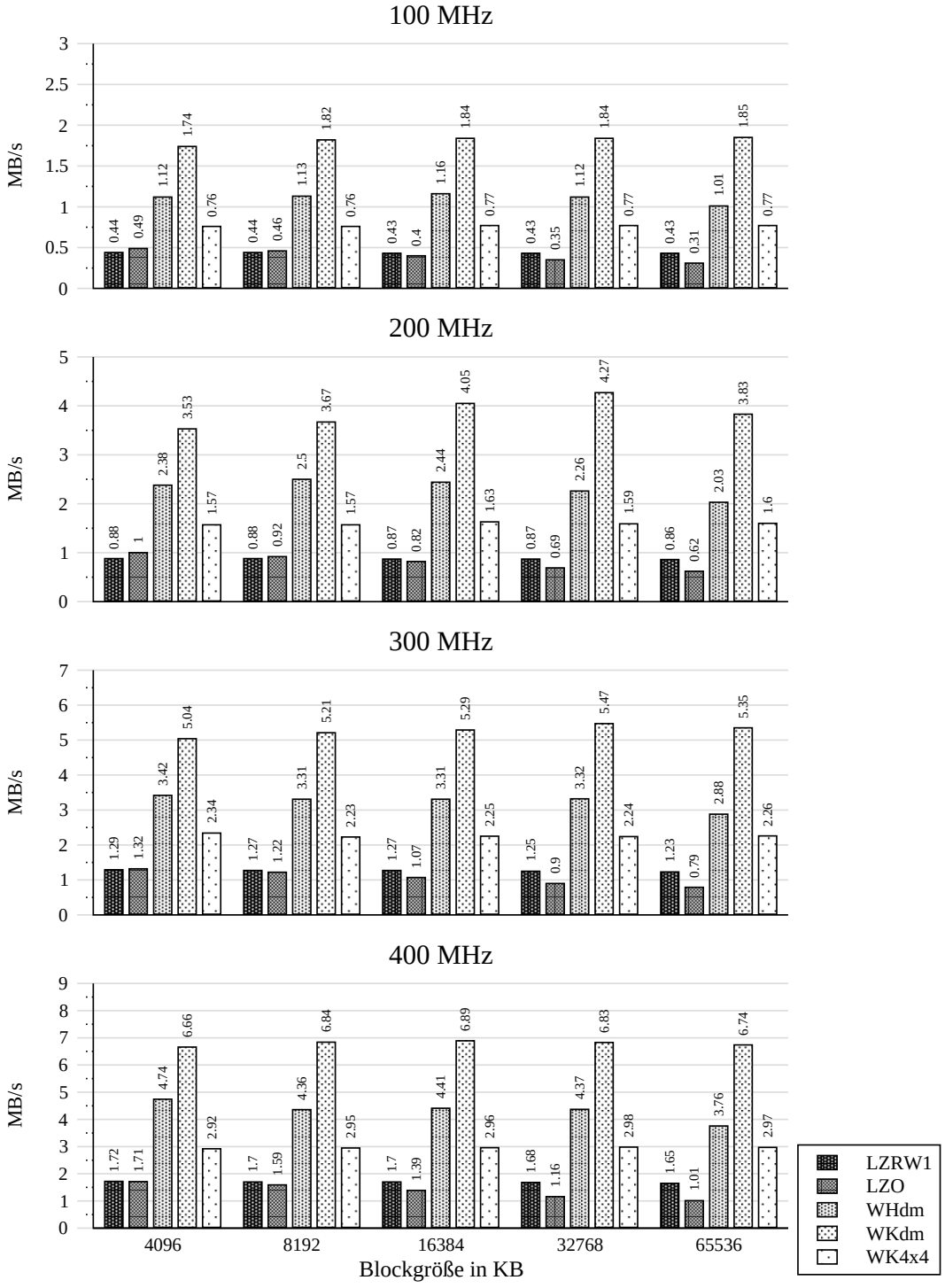

Abbildung A.15: madplay mit Audiodatei - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

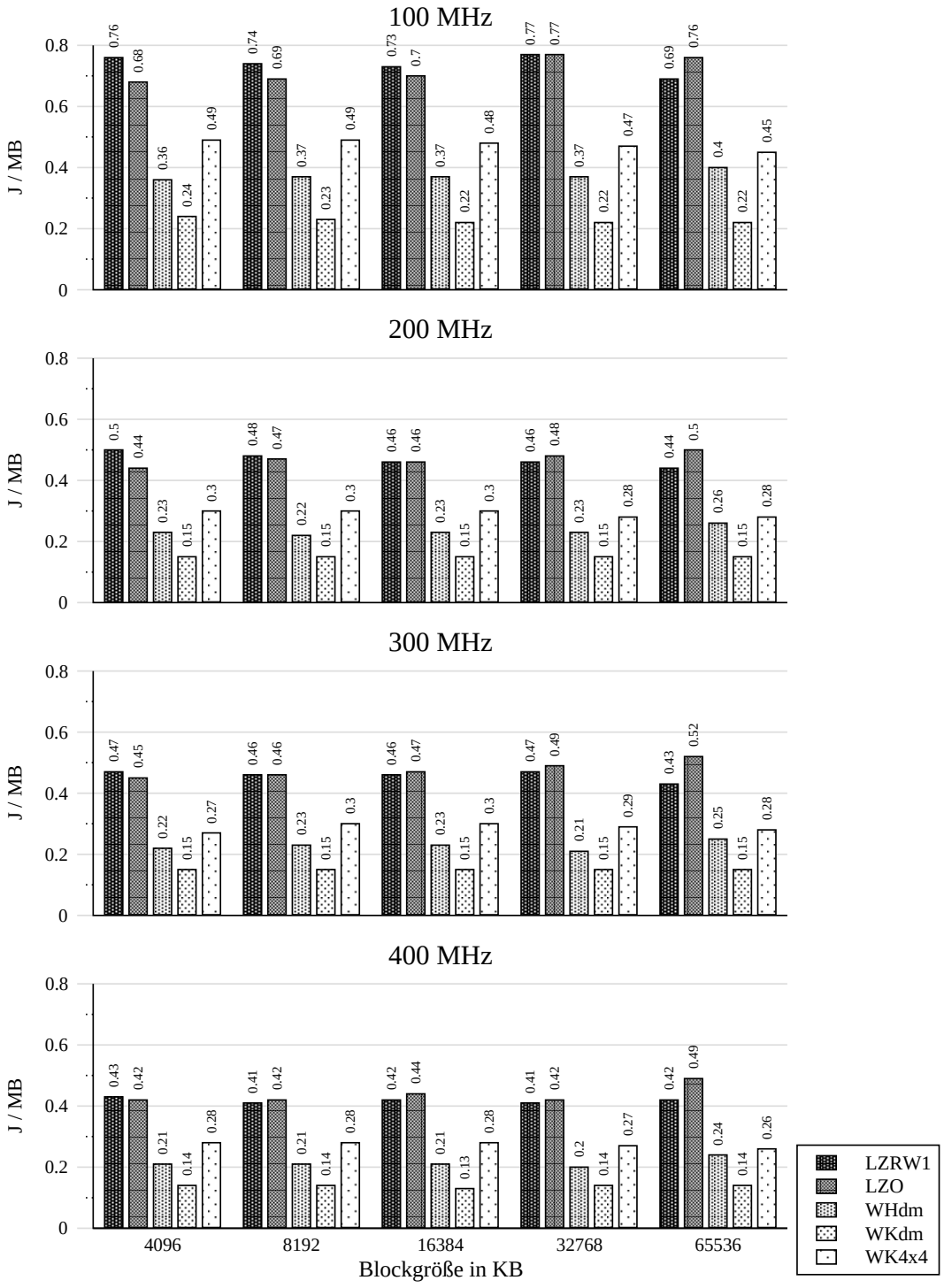

Abbildung A.16: madplay mit Audiodatei - Codesegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

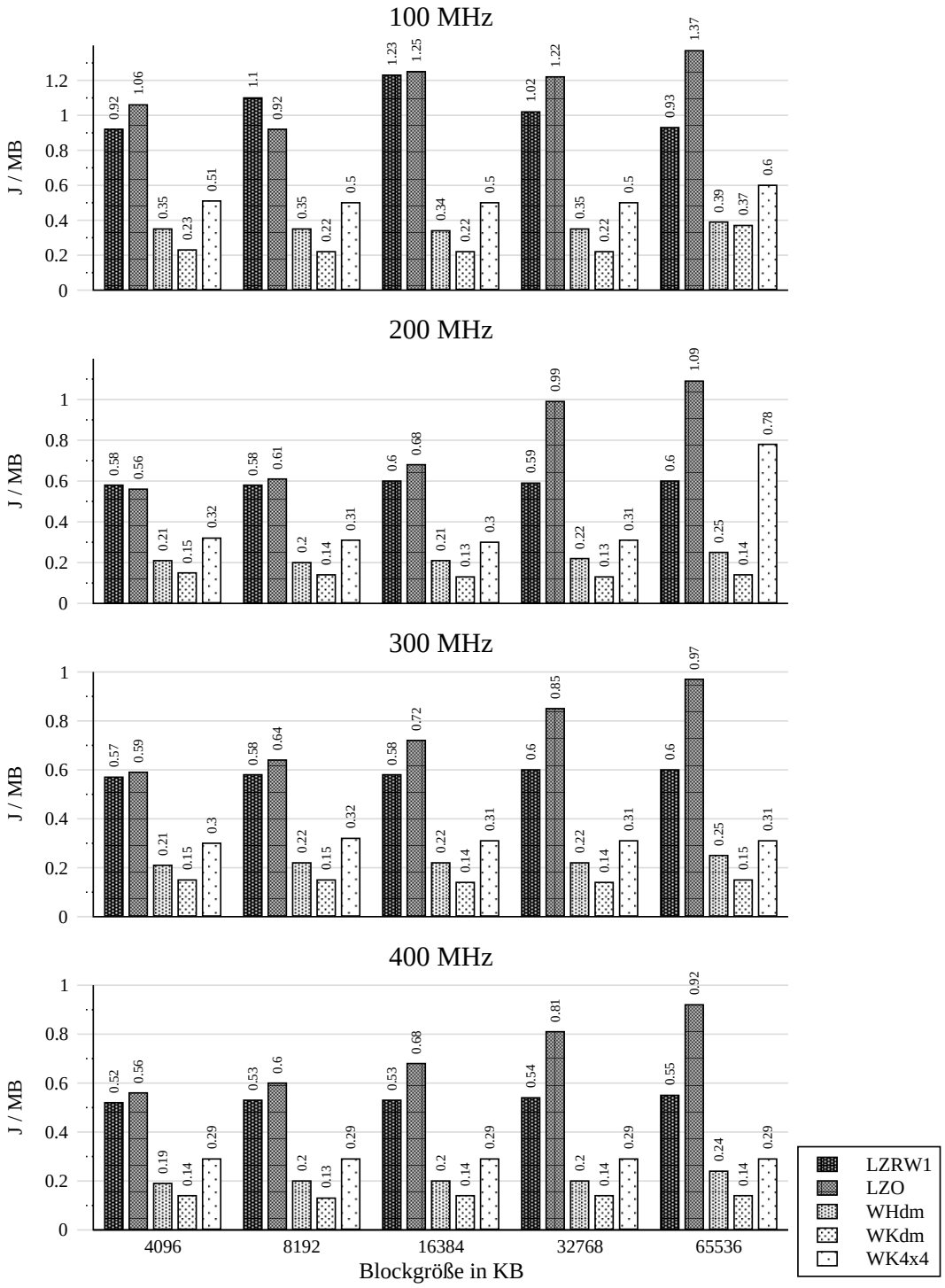

Abbildung A.17: madplay mit Audiodatei - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

A.4 Konqueror

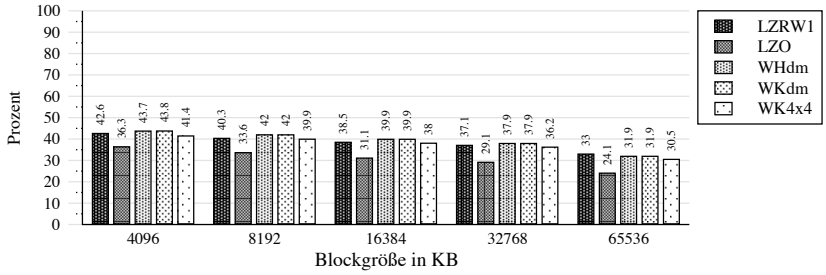

Abbildung A.18: Konqueror leer Thread 1 - Datensegment: Kompressionsrate des Segments

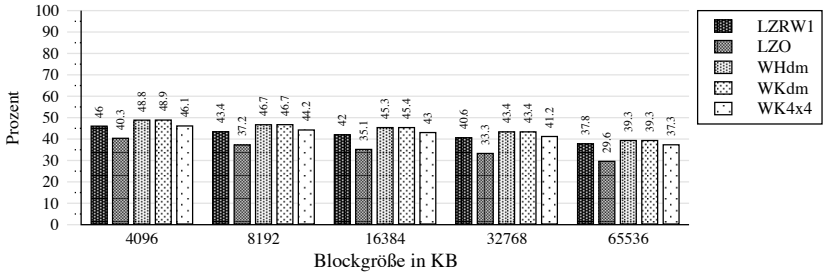

Abbildung A.19: Konqueror mit geladener einfacher Webseite Thread 1 - Datensegment: Kompressionsrate des Segments

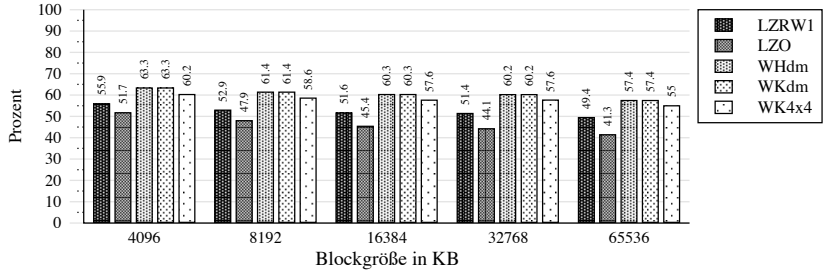

Abbildung A.20: Konqueror mit geladener komplexer Webseite Thread 1 - Datensegment: Kompressionsrate des Segments

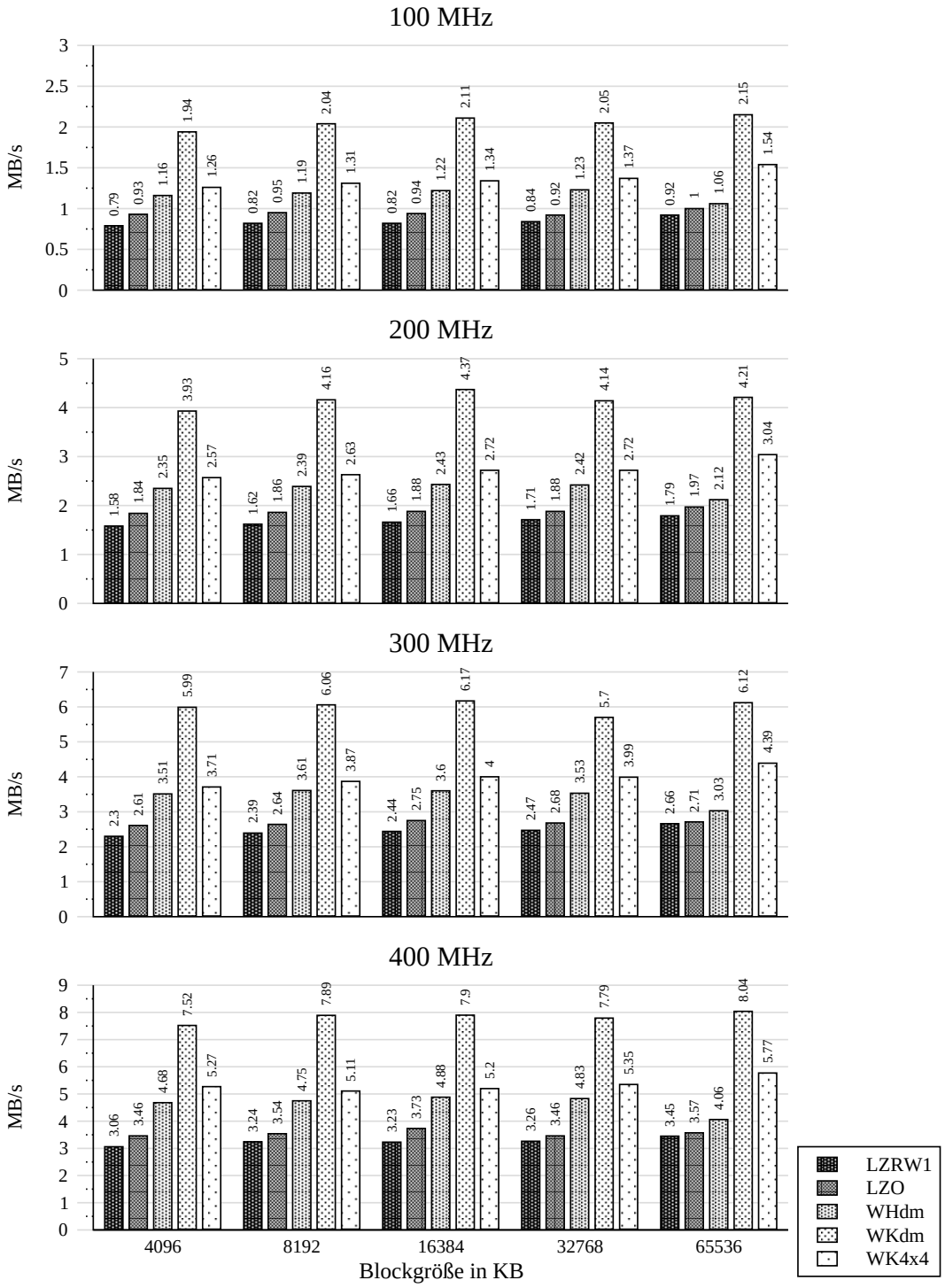

Abbildung A.21: Konqueror leer Thread 1 - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

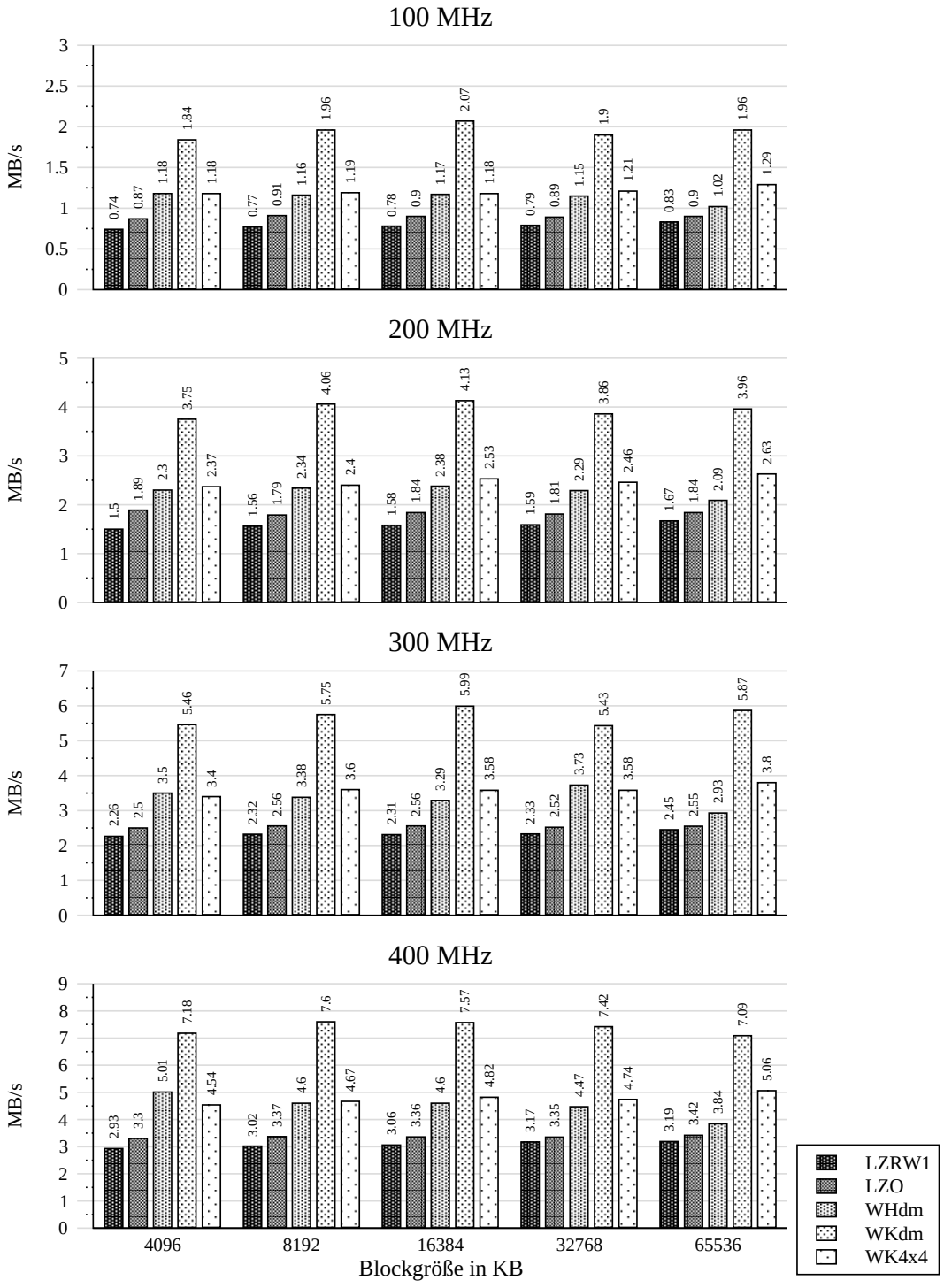

Abbildung A.22: Konqueror mit geladener einfacher Webseite Thread 1 - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

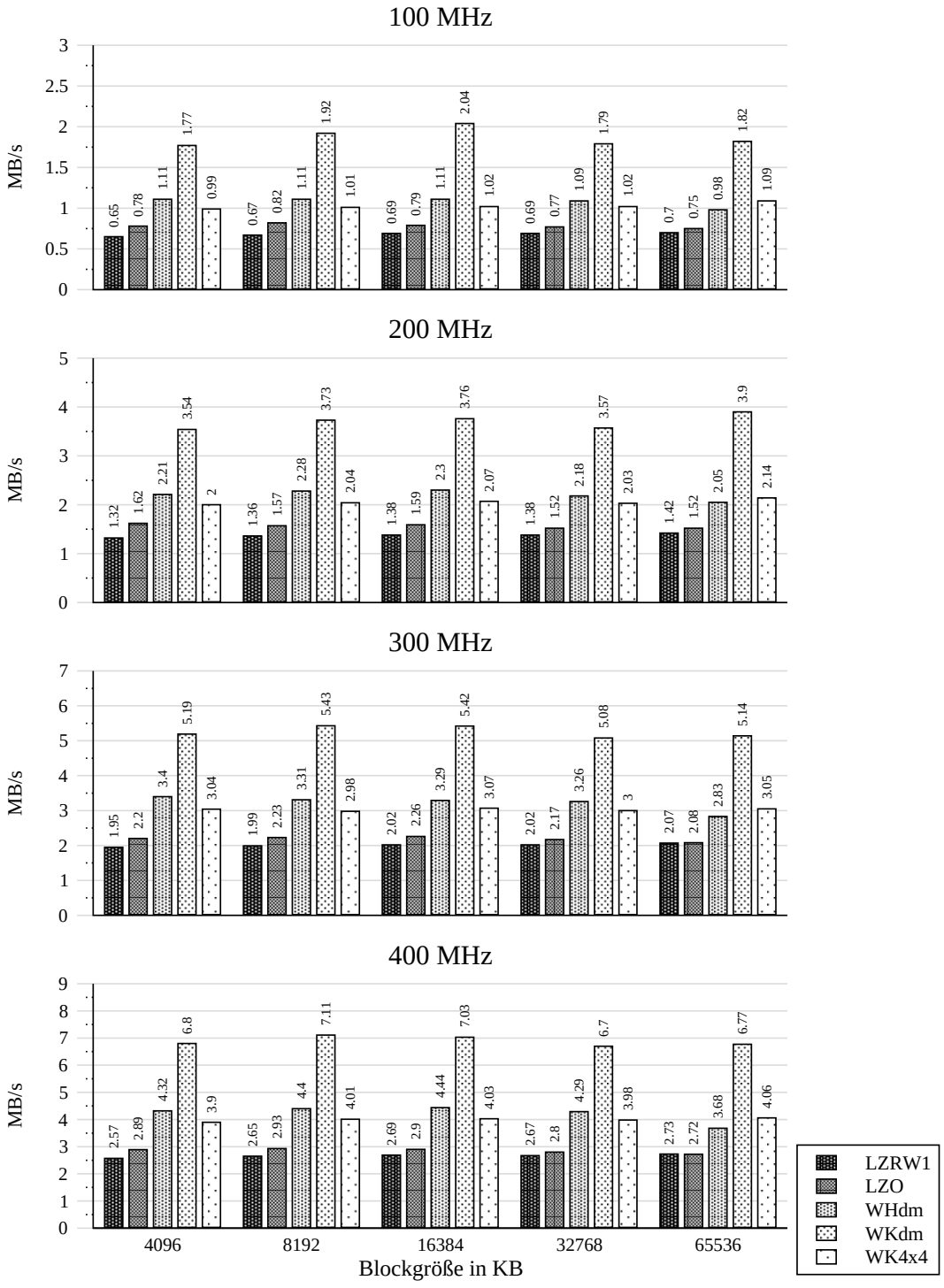

Abbildung A.23: Konqueror mit geladener komplexer Webseite Thread 1 - Datensegment: Geschwindigkeit der Kompression und Dekompression des Segments in MB pro Sekunde

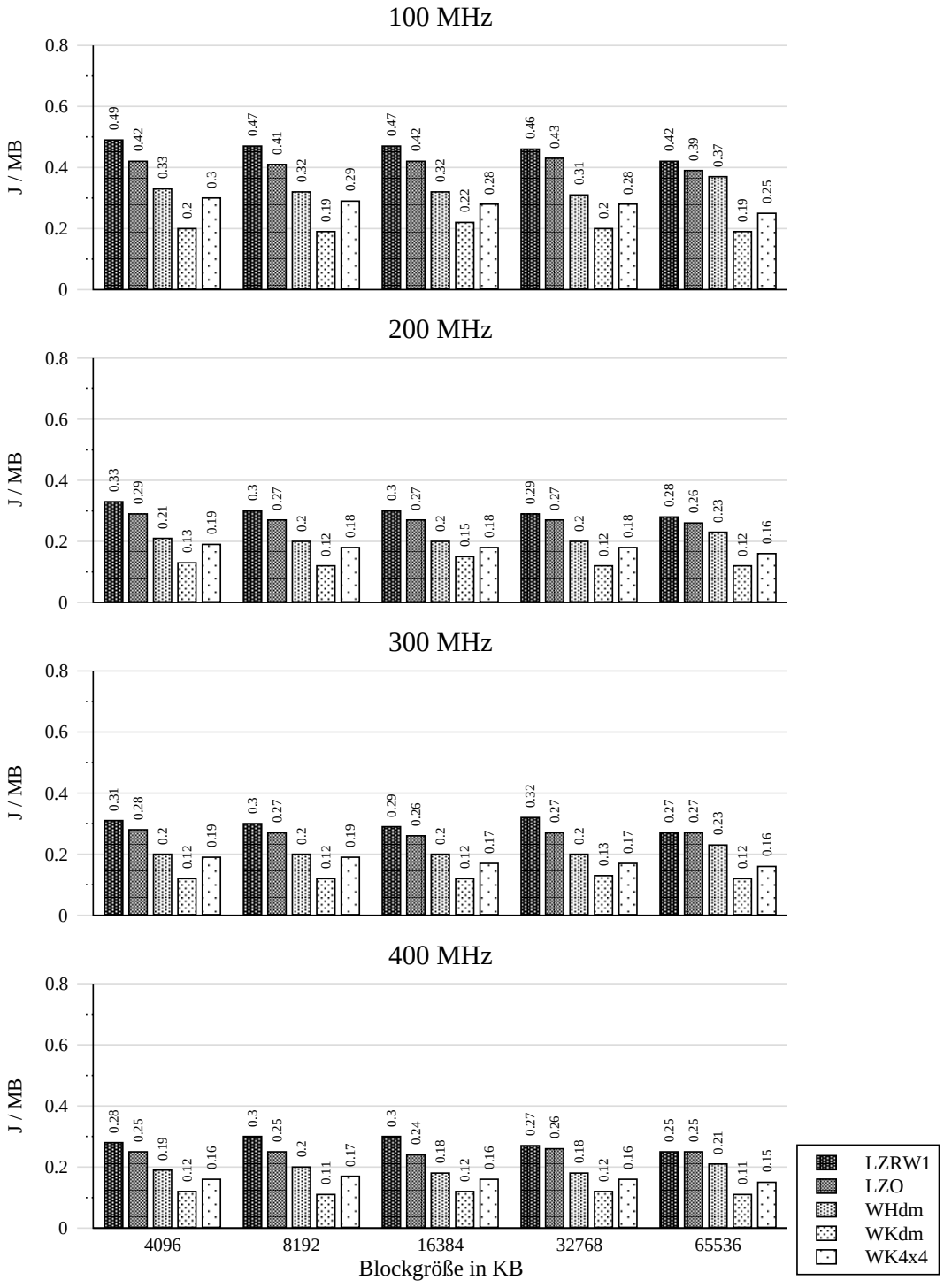

Abbildung A.24: Konqueror leer Thread 1 - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

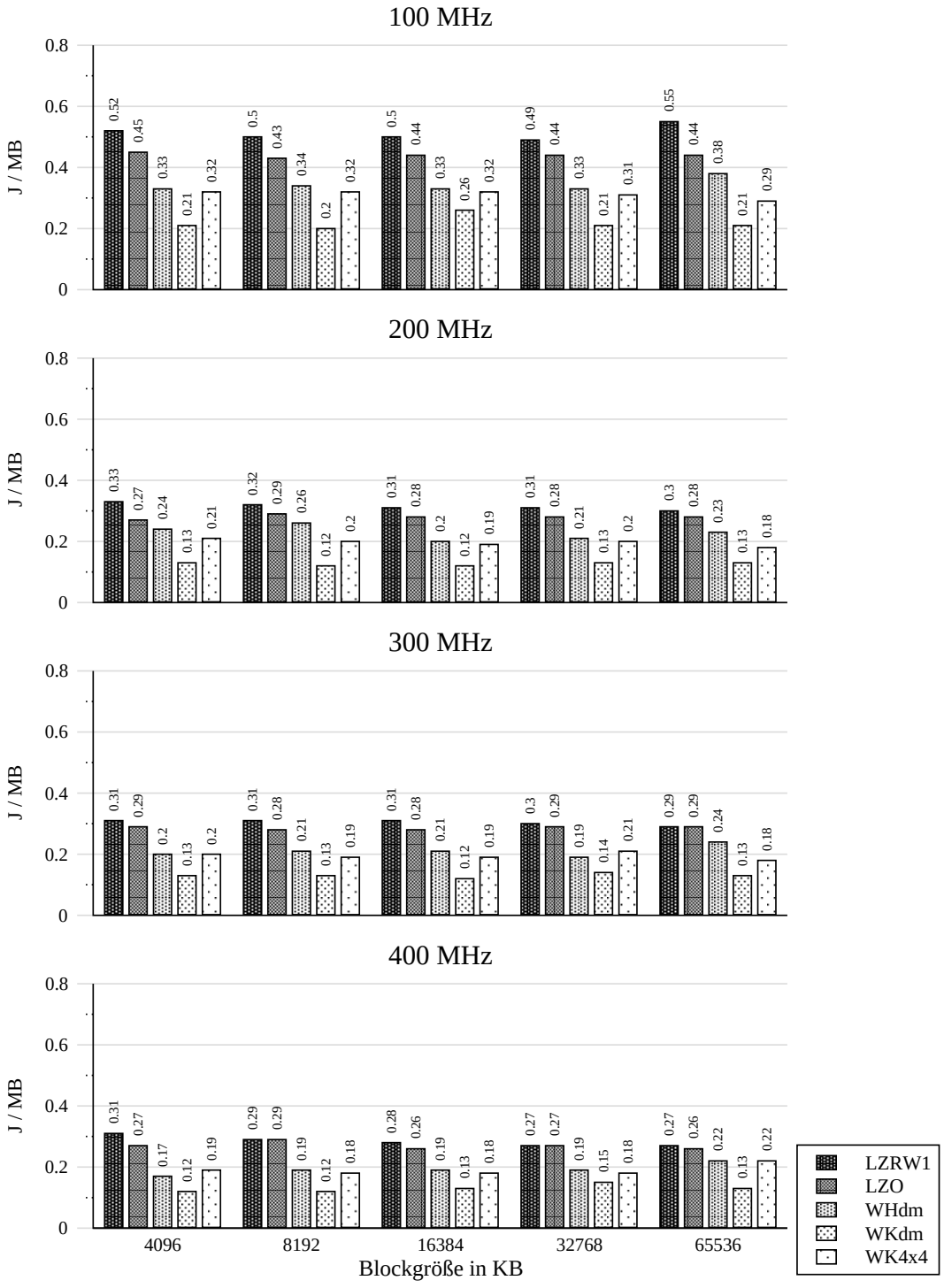

Abbildung A.25: Konqueror mit geladener einfacher Webseite Thread 1 - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

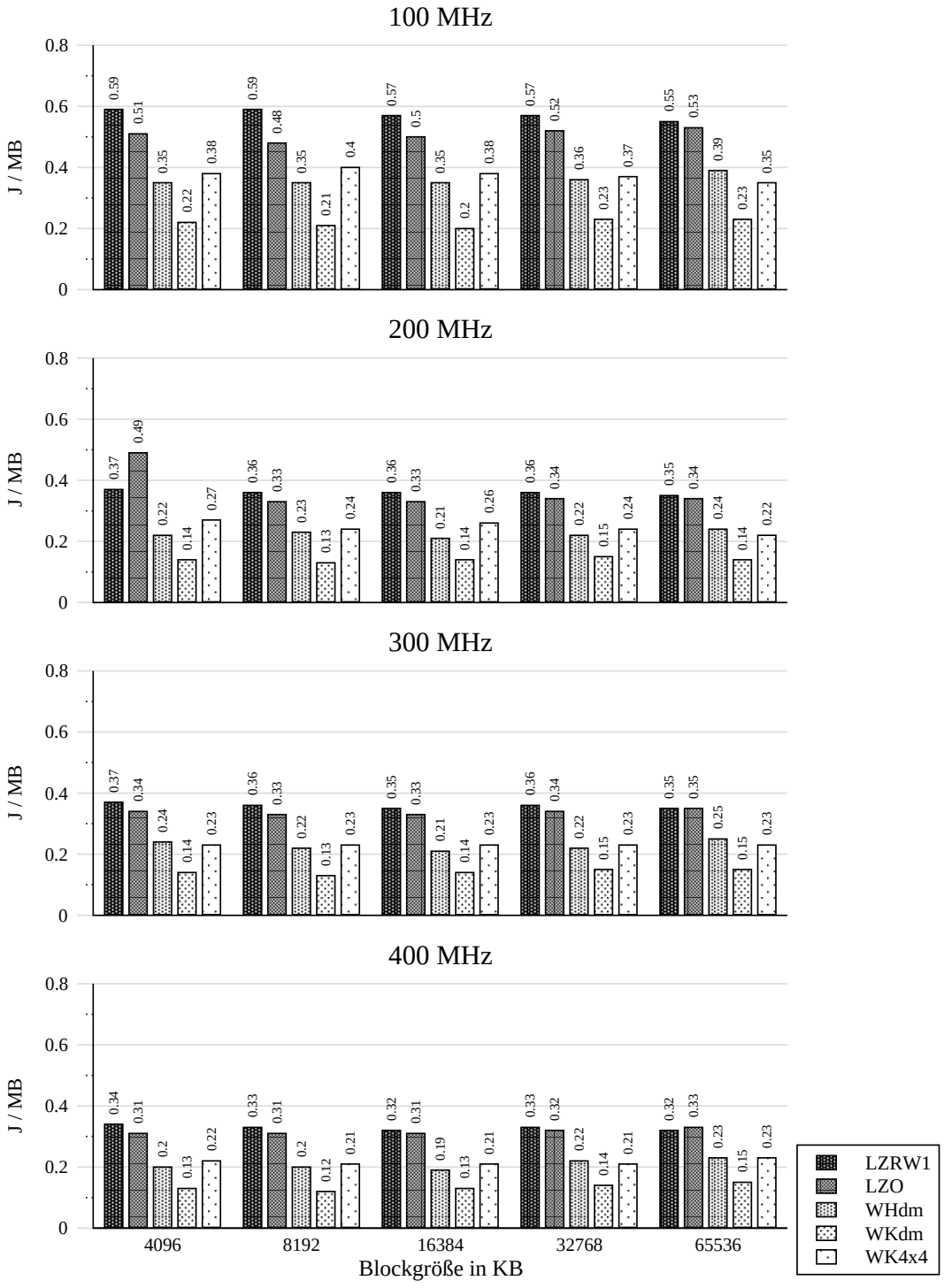

Abbildung A.26: Konqueror mit geladener komplexer Webseite Thread 1 - Datensegment: Benötigte Leistung der Kompression und Dekompression des Segments in Joule pro MB

Literaturverzeichnis

[BMMP99] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'99*, August 1999.

[BW03] Frank Bellosa and Andreas Weißel. Vorlesung: Ausgewählte Kapitel der praktischen Betriebsprogrammierung: Power Management. `http://www4.informatik.uni-erlangen.de/Lehre/SS03/V_AKBP/`, 2003.

[CCB99] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*, 1999.

[dCdLS03] Rodrigo S. de Castro, Alair Pereira do Lago, and Dilma Da Silva. Adaptive Compressed Caching: Design and Implementation. In *15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, November 2003.

[Dou93] Fred Douglis. The Compression Cache: Using Online Compression to Extend Physical Memory. In *Proceedings of the USENIX Winter Technical Conference*, pages 519 – 529, 1993.

[Duf02] David Duffey. DUMP: Dump User Memory, Please. `http://davidduffey.com/dump/`, 2002.

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Mel Gorman, February 2004.

[HPS03] H. Huang, P. Pillai, and K. G. Shin. Design and Implementation of Power-Aware Virtual Memory. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[Hra] Maximilian Hrabowski. Proseminar Redundanz - Vortrag 5: Lempel Ziv. `http://goethe.ira.uka.de/seminare/redundanz/vortrag05/`.

[Int] Intel. StrongArm PXA255. `http://developer.intel.com/design/pca/prodbref/252780.htm`.

[Jäg] Marc-Alexander Jäger. Die Ziv-Lempel-Kompression. `http://www.lempel-ziv.de/`.

[KAM02] Nam Sung Kim, Todd Austin, and Trevor Mudge. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues WMPI'02*, May 2002.

[Kap99] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, December 1999, 1999.

[KARa] KARO. Embedded Computing mit TRITON LP (Low Power Design). `http://www.karo-electronics.de/15.0.html`.

[KARb] KARO. Starter-Kit II für TRITON. `http://www.karo-electronics.de/13.0.html`.

[KI02] Krishna Kant and Ravi Iyer. Compressibility Characteristics of Address/Data Transfers in Commercial Workloads. `citeseer.ist.psu.edu/575541.html`, February 2002.

[LFZE00] Alvin Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power Aware Page Allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS'00*, November 2000.

[Obe] Markus F.X.J. Oberhumer. LZO Version 1.08. `http://www.oberhumer.com/opensource/lzo/`.

[QW04] Markus Quaritsch and Thomas Winkler. Seminar: Linux - Ein Einblick in den Kernel. `http://courses.iicm.edu/~hkrott/site/docs/seminar/sem2003_linux.pdf`, February 2004.

[Ram99] Rambus. Desktop Performance for Mobile PCs. `http://www.rambus.com/downloads/MobileWP%20r2.4.pdf`, 1999.

[TFR+01] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2):271 – 285, 2001.

[Wil91] R.N. Williams. An Extremely Fast Ziv-Lempel Data Compression Algorithm. In *IEEE Data Compression Conference*, pages 362–371, April 1991.

[WKS99] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *USENIX 1999 Annual Technical Conference, Monterey, California*, pages 101–116, 1999.

[YZG00] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent Value Compression in Data Caches. In *33th Annual International Symposium on Microarchitecture (MICRO'00)*, 2000.

[ZL77] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.