Daniel Versick

Verfahren und Werkzeuge zur Leistungsmessung, -analyse und -bewertung der Ein-/Ausgabeeinheiten von Rechensystemen

urn:nbn:de:gbv:28-diss2010-0036-6

Die vorliegende Arbeit wurde von der Fakultät für Informatik und Elektrotechnik der Universität Rostock als Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.) angenommen. Die Verteidigung der Dissertation fand am 17.12.2009 statt.

Gutachter:

Prof. Dr.-Ing. habil. Djamshid Tavangarian
Universität Rostock, Lehrstuhl für Rechnerarchitektur

Prof. Dr. Arndt Bode
Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation

Prof. Dr. Wolfgang E. Nagel
Technische Universität Dresden, Professur für Rechnerarchitektur

Prof. Dr.-Ing. Wolfgang Rehm
Technische Universität Chemnitz, Professur Rechnerarchitektur

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-8325-2393-0

Logos Verlag Berlin GmbH
Comeniushof, Gubener Str. 47,
10243 Berlin
Tel.: +49 (0)30 42 85 10 90
Fax: +49 (0)30 42 85 10 92
INTERNET: http://www.logos-verlag.de

Kurzfassung. Die Komplexität moderner Rechensysteme erschwert immer stärker deren Leistungsbewertung, da einfache Parameter, wie z. B. die Taktfrequenz einer CPU, nicht mehr ausreichen, um die Leistungsfähigkeit der Rechner hinreichend genau zu bewerten. Zur Leistungsbewertung moderner Computer verwendet man, neben modellbasierten Verfahren, oftmals Benchmarks – Programme, die eine möglichst realitätsnahe Last erzeugen und anhand dieser die Leistungsfähigkeit eines Rechensystems bzw. einer Komponente des Systems in seiner gesamten Komplexität vermessen.

Benchmarks, die die Leistung der Recheneinheiten von Computern ermitteln, sind weit verbreitet und wurden in den letzten Jahrzehnten hinreichend wissenschaftlich untersucht. Die Bewertung der Leistungsfähigkeit eines Rechensystems in Hinblick auf dessen Datenein- und ausgabe (I/O) ist hingegen kaum betrachtet worden. Da aber die Datentransferleistung von I/O-Systemen zu den CPUs weniger stark ansteigt, als die Rechenleistung der Computer, werden die CPUs in Zukunft oft nicht ihr volles Rechenpotential nutzen können. Um den negativen Leistungseinfluss der I/O-Systeme zu minimieren, ist eine genaue Vermessung und Optimierung dieser Systeme notwendig.

Inhalt dieser Dissertationsarbeit ist es, die Probleme aktueller I/O-Benchmarks zur Leistungsermittlung beim Zugriff auf den Sekundärspeicher aufzuzeigen und zu lösen. Es wird ein neuer Ansatz entwickelt, der realitätsnahes, nutzerrelevantes, vergleichbares und dennoch einfaches I/O-Benchmarking insbesondere in Hinblick auf die Leistungsermittlung beim Zugriff auf Sekundärspeicher mittels der MPI-IO-Schnittstelle ermöglicht.

Ausgehend von den notwendigen Schritten bei der Leistungsanalyse wird eine neue Benchmark-Architektur entwickelt, die insbesondere Lösungen für die gefundenen Probleme der *geringen Repräsentativität von Benchmarkergebnissen* und *der fehlenden Nutzerunterstützung beim Benchmarking* bietet und damit über vorhandene Arbeiten in diesem Themenbereich deutlich hinausgeht. Es wird ein Benchmarksystem erstellt, das nutzerrelevante Ergebnisse ermittelt, indem es dem Nutzer ermöglicht, das Lastverhalten MPI-IO-basierter Applikationen als Messgrundlage zu verwenden. Außerdem wird eine realitätsnahe und einfach nutzbare I/O-Lastbeschreibung präsentiert, deren Möglichkeiten existierende I/O-Lastbeschreibungen in Hinblick auf Genauigkeit bei Verwendung komplexer paralleler I/O-Lasten übersteigen. Die Funktionsfähigkeit und Genauigkeit des I/O-Benchmarking-Ansatzes wird mit Messungen anhand von zwei Beispielapplikationen gezeigt.

Abstract. The complexity of current computer systems impedes their performance analysis because simple parameters such as the CPU clock rate are not sufficient to evaluate systems performance. Aside from model-based methods, benchmark-programs are employed for performance evaluation of modern computers. Benchmarks measure the performance of computer systems or system components with full complexity using a reality-based workload.

Computational benchmarks evaluating the performance of computing functional units are widely-used and have been examined sufficiently during the last decades. However, the evaluation of computer system performance with regard to its data input and output (I/O) had been marginally examined. As the performance of data transfer from I/O systems to CPUs doesn't increase as fast as the processor speed, future CPUs are not able to exploit full computational capability. An accurate measurement and optimization of I/O systems is necessary to minimize their negative impact on performance.

Purpose of this dissertation is to discover and solve problems of current I/O benchmarks employed for evaluating the performance of secondary storage access. It is developed a new approach for realistic, user-relevant, comparable, and simple I/O benchmarking especially in regard to performance evaluation of secondary storage access using the MPI-IO interface.

Based on the required steps of performance analysis a novel benchmark architecture is developed, that provides solutions for the discovered problems of *small representativeness of benchmark results* and *missing assistance for benchmarking users* and thus, exceeds the state of the art in this field of work. Therefore, a benchmarking system is presented allowing the user to define workloads of MPI-IO based applications for evaluating user-relevant results. A realistic and easy-to-use I/O workload description is developed that enhances existing I/O workload descriptions with regard to accuracy when using complex parallel I/O workloads. The functionability and accuracy of the I/O benchmarking approach is shown by means of two example applications.

Keywords. High Performance Computing, I/O, High Performance I/O, Workload Specification, I/O-Benchmark, PRIOmark

Danksagungen. Ich bedanke mich herzlich bei allen Personen, die mich bei der Erstellung dieser Arbeit unterstützt haben. Besonderer Dank gilt dabei meinem Mentor Prof. Dr.-Ing. habil. Djamshid Tavangarian, der mir jederzeit bei Fragen und Problemen zur Seite stand. Ebenso bedanke ich mich bei Prof. Dr. Arndt Bode, Prof. Dr. Wolfgang Nagel und Prof. Dr.-Ing. Wolfgang Rehm für ihre Mühen bei der Begutachtung dieser Dissertation.

Ich bedanke mich außerdem bei Ronny Bartsch, Ralf Behnke, Clemens Holzhüter, Christian Rataj-Weinreben, Wahe Sedrakian, Quiye Wang und ganz besonders bei Michael Krietemeyer, die als Studierende und Mitarbeiter im IPACS-Projekt an der Universität Rostock tätig waren, sowie natürlich den Förderern und Mitarbeitern der weiteren Forschungseinrichtungen des IPACS-Projektes, ohne die diese Arbeit gar nicht möglich gewesen wäre.

Weiterer Dank gilt den zahlreichen fleißigen Augen, die mir unzählige Hinweise zur Erstellung der Arbeit gaben, insbesondere Dr. Robil Daher, Dr. Heiko Kopp, Martin Krohn und Dr. Ulrike Lucke.

Keine Arbeit gelingt ohne die Unterstützung der Familie. An dieser Stelle geht ganz besonderer Dank an meine Frau Daniela, die mich auch zur Arbeit schubste, wenn die Motivation nachließ. Ganz herzlich bedanke ich mich bei meinen Eltern: bei meinem Vater, dem ich die Freude an der Wissenschaft verdanke, bei meiner Mutter und meiner Großmutter, die mich immer unterstützt haben - und natürlich bei meinem Bruder Dirk, auf den ich mindestens genauso stolz bin, wie er auf mich.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Anfang ist die Hälfte des Ganzen.
– ARISTOTELES (384 - 322 V. CHR.)

Ziele der Leistungsermittlung

Die Leistungsanalyse von Rechensystemen hat seit der Entwicklung der ersten Computer einen hohen Stellenwert bei Systemadministratoren und -betreibern. Leistungsanalysemethoden, wie die *Simulation* von Rechensystemen, ermöglichen die Vorhersage der Systemleistung noch bevor die Systeme entwickelt oder beschafft werden. Dadurch können Entscheidungen, ob ein Rechensystem in einem bestimmten Anwendungsfall sinnvoll eingesetzt werden kann, bereits in einem sehr frühen Stadium der Entwicklung getroffen werden, bevor eine Fehlentscheidung ökonomischen Schaden anrichtet [1].

Messungen als weitere Analysemethode können verwendet werden, um in einem existierenden System Leistungsengpässe zu erkennen, sowie Optimierungen durchzuführen und zu bewerten. Insbesondere Systembetreiber versuchen die Leistung ihrer Rechensysteme stetig zu steigern, um so den Kosten-Nutzen-Faktor zu erhöhen. Messungen ermöglichen auch einen direkten Vergleich verschiedener existierender Rechensysteme. So kann bei der Beschaffung neuer Rechenanlagen bereits vor der Kaufentscheidung anhand von Testsystemen die Systemleistung der Alternativen verifiziert werden, um optimale Rechensysteme bei möglichst geringen Kosten zu finden [1]. Leistungsanalysemethoden verfolgen also mehr als nur den Zweck der reinen Leistungsermittlung. Im Regelfall werden sie aus ökonomischen Gründen durchgeführt und geben den Fachkräften für Informationstechnik eines Unternehmens Möglichkeiten an die Hand, bei einem begrenzten finanziellen Budget eine hohe Arbeitseffizienz zu erreichen.

Hardware-Hersteller

Hardware-Hersteller haben dieses Bedürfnis der Anwender bereits erkannt und bewerben ihre Produkte mit Leistungswerten, die sie mit Hilfe verschiedener Analysemethoden während und nach der Hardware-Entwicklung ermittelt haben. Bei Prozessoren wird u. a. mit der Taktfrequenz geworben und bei Festplatten neben deren Kapazität mit dem Durchsatz und der Latenz. Ob und wie gut diese Leistungswerte wirklich

die Bedürfnisse der Anwender treffen, bleibt oft ungeklärt, da nicht beschrieben ist, welche Grundlagen der Hersteller bei der Vermessung seiner Systeme verwendet hat. Prozessorhersteller geben beispielsweise nicht an, ob die hohe Taktrate des Prozessors durch eine Pipeline mit besonders vielen Pipelinestufen erreicht wurde, die ihrerseits unter bestimmten Bedingungen Leistungseinbußen infolge einer höheren Anzahl an Pipelinehemmnissen zur Folge hat. In einem solchen Fall steigt die Leistung eines Prozessors nicht in gleichem Maße wie seine Taktfrequenz [2].

Ähnlich verhält es sich auch bei anderen Produzenten. Festplattenhersteller geben oft nicht an, ob der Durchsatz der Festplatte z. B. beim sequentiellen Lesen von Daten oder beim Schreiben von Daten auf zufällige Bereiche der Festplatte ermittelt wurde. Beide Zugriffsarten verursachen jedoch signifikant unterschiedliche Leistungswerte. Um also Ergebnisse zu erhalten, die der Nutzer einschätzen und in Bezug zu seinen eigenen Anforderungen setzen kann, sollte er sich nur in wenigen Fällen auf die Informationen der Hardware-Hersteller verlassen. Dies mag im Bereich der privaten Nutzung von Rechensystemen und Systemkosten von wenigen Tausend Euro noch vertretbar sein – wenn aber ein Hochleistungsrechner für einen Betrag von mehreren Hunderttausend Euro für eine ganz spezifische Simulationsaufgabe beschafft wird, ist eine vorherige Analyse der existierenden Alternativen in Hinblick auf den geplanten Einsatzzweck sehr wichtig.

1.1 Leistungsanalyse von Rechensystemen

Zu Zeiten der ersten digitalen Rechenmaschinen war die Ermittlung der Leistung dieser Maschinen eine relativ einfache Aufgabe. Damalige Systeme hatten weder Cache-Speicher, die den Speicherzugriff durch Zwischenspeicherung in einem schnellen Zwischenspeicher optimieren, noch hatten die Prozessoren komplexe Eigenschaften, wie Pipelining oder Superskalarität, die mittels erhöhter Parallelität innerhalb der CPU Leistungsverbesserungen erzielen.

Aufgrund der verhältnismäßig niedrigen Komplexität der frühen Rechner konnte die Leistung des Systems durch nur wenige Werte charakterisiert werden. Klassische Performance-Metriken waren *Millionen Instruktionen pro Sekunde* (MIPS) und *Fließkommaoperationen pro Sekunde* (FLOPS). Diese Performance-Metriken wurden bereits in den 80er Jahren stark kritisiert, da (a) selbst auf einer einzelnen Rechnerarchitektur die Instruktionslängen und Ausführungszeiten von Instruktionen stark variieren können, was bedeutet, dass unterschiedliche Applikationen unterschiedliche Anzahlen von Instruktionen pro Zeiteinheit bearbeiten und (b) der Vergleich zwischen Rechnerarchitekturen unmöglich ist. Bereits der Vergleich eines *Reduced Instruction Set Computers* (RISC) mit einem *Complex Instruction Set Computer* (CISC) ergibt keine eindeutigen Ergebnisse, da RISC-Systeme kürzere Instruktionsausführungszeiten haben, aber im Allgemeinen für die Ausführung einer Aufgabe mehr Instruktionen benötigen, als ein

CISC. Entsprechend kann ein RISC eine höhere MIPS-Rate besitzen als ein zu vergleichender CISC, trotz nicht zwangsläufig kürzerer Applikationslaufzeit. Gerade die Applikationslaufzeit ist es aber, die für den Benutzer interessant ist. Ob diese auf dem RISC tatsächlich kürzer ist, als auf dem CISC-Vergleichssystem, hängt von den Aufgaben ab, die die Applikation durchführt [3]. Dieser Sachverhalt zeigt, dass selbst bei diesen einfachen Performance-Metriken eine starke Abhängigkeit der Ergebnisse von den Applikationen existiert.

1.2 Systemleistung

Abbildung 1.1 zeigt den grundlegenden Aufbau eines Rechnersystems mit dem Prozessor (Central Processing Unit - CPU) als zentraler Komponente. Die CPU besteht aus Steuerwerk und Rechenwerk zur Verarbeitung von Programm- und Rechendaten, sowie I/O-Schnittstellen, an denen Ein-/Ausgabeeinheiten angeschlossen sind, um die Datenein- und Ausgabe für die CPU zu realisieren.

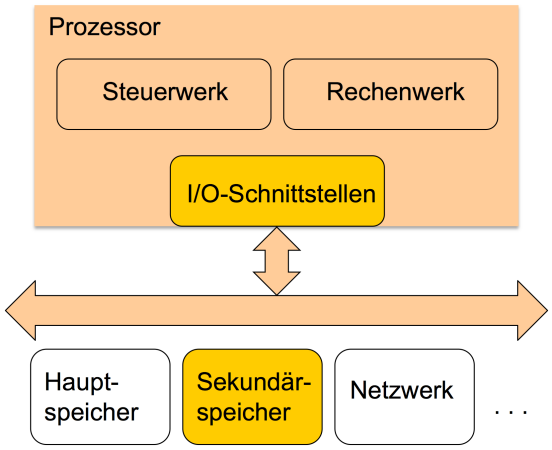

Abbildung 1.1: Aufbau eines Rechnersystems

Möchte der Anwender eine Leistungsanalyse dieses Systems durchführen, ist für ihn die Leistung des Gesamtsystems bei der Ausführung der für ihn relevanten Applikationen von Bedeutung. Die Leistung des Systems wird dabei von zahlreichen Faktoren beeinflusst, da ein Rechnersystem aus einer großen Anzahl unterschiedlicher Komponenten besteht, von denen nur einige in Abbildung 1.1 dargestellt sind. Diese Komponenten besitzen ihrerseits unterschiedliche Leistungswerte.

Leistungsbewertung der Datenverarbeitung

Aufgaben, die Applikationen auf Rechensystemen durchführen, bestehen klassischerweise aus drei Phasen. Nach der Eingabe der Rechendaten folgt die Verarbeitung, der ihrerseits die Ausgabe der Ergebnisse folgt. Die Verarbeitung der Daten ist in diesem Zusammenhang eine ganz wesentliche Aufgabe des Rechners, die typischerweise von

dem Prozessor übernommen wird. Aus diesem Grund wurden Prozessoren (und insbesondere deren Rechenwerke) in der Vergangenheit bereits in großem Umfang als leistungscharakterisierende Komponente eines Rechensystems untersucht. Benchmarks, die die Leistungsfähigkeit der Recheneinheit vermessen, sind Stand der Technik und hinreichend verfügbar. Die TOP500 der leistungsstärksten Computer der Welt werden mit einem weitverbreiteten Benchmark vermessen, der als Rechenbenchmark gilt [4]. Der dafür eingesetzte *High Performance Linpack* berechnet die Lösung eines linearen Gleichungssystems, ohne die Ein- und Ausgabe der in der Berechnung verwendeten Matrix, die mehrere Hundert Megabyte groß sein kann, in die Leistungsbetrachtung einzubeziehen [5]. Eine realistische Leistungsbewertung für Applikationen, die viele Ein- und Ausgabeoperationen durchführen, kann allein mit einem solchen Benchmark nicht durchgeführt werden.

Leistungsbewertung der Ein-/Ausgabe

Um dennoch die komplette Systemleistung moderner Rechensysteme anhand von Benchmarks einschätzen zu können, widmet sich diese Arbeit einem bisher wenig wissenschaftlich untersuchten Bereich der Leistungsbewertung: der Ein- und Ausgabe von großen Datenmengen. Abbildung 1.1 macht deutlich, dass es verschiedene Ein-/Ausgabekomponenten gibt. Da große Datenmengen oft auf Sekundärspeichersystemen abgelegt werden, liegt der Fokus dieser Arbeit auf der Leistungsanalyse von Sekundärspeichersystemen mit sogenannten I/O-Benchmarks. Um den Fokus der Arbeit in diesem Umfeld nicht einzuschränken, werden insbesondere die hochkomplexen Sekundärspeichersysteme von Hochleistungscomputern betrachtet, die einen parallelen Zugriff vieler Applikationen von verschiedenen Prozessoren unterstützen. Die dabei entstehenden Konzepte sind auf weniger komplexe I/O-Systeme übertragbar.

Die Arbeit stellt somit eine Erweiterung aktueller Methoden der Leistungsbewertung um die I/O-Komponente dar, so dass im Ergebnis durch die Kombination bisher verfügbarer Benchmarks mit der neuen I/O-Komponente eine relevante und komplette Leistungsbewertung aktueller Computer durchgeführt werden kann.

IPACS-Projekt

Diese Arbeit entstand im Kontext des BMBF-geförderten Verbundprojektes *Integrated Performance Analysis of Computer Systems* (IPACS), an dem neben der Universität Rostock, das Fraunhofer Institut für Techno- und Wirtschaftsmathematik Kaiserslautern, das Rechenzentrum der Universität Mannheim, die T-Systems Enterprise Services GmbH und anfänglich auch die Pallas GmbH beteiligt waren und dessen Zielstellung die Entwicklung neuer nutzerrelevanter und einfach zu bedienender Benchmarks für Rechnersysteme war. Viele der Gedanken, die Teil dieser Arbeit sind, wurden mit den Projektpartnern, die ausnahmslos zahlreiche nationale und internationale wissenschaftliche Erfolge im Bereich des Hochleistungsrechnens nachweisen können, diskutiert und validiert.

1.3 Das Memory-Wall-Problem des Sekundärspeichers

Mit der starken Leistungssteigerung moderner Rechner geht auch eine größer werdende Leistungslücke zwischen Speicher und Prozessoren einher, die darin begründet ist, dass die Geschwindigkeit von Prozessoren in den letzten Jahren deutlich stärker steigt als die Leistung von Speichern [6]. Um dem entgegen zu wirken, ist Speicher in modernen Rechensystemen hierarchisch aufgebaut, wie Abbildung 1.2 zeigt. Beginnend mit CPU-Registern als höchster Ebene der Speicherhierarchie folgen unterschiedliche Ebenen von Cache-Speichern und der Arbeitsspeicher oder Hauptspeicher des Systems [2]. Neben den in der Abbildung dargestellten Hierarchie-Ebenen können Speicherhierarchien aktueller Rechensysteme weitere Ebenen enthalten. Beispielsweise können 3rd-Level-Caches existieren oder auch eigene Caches in den Sekundärspeichersystemen, um den langsamen Zugriff der im Allgemeinen mechanisch arbeitenden Sekundärspeicher zu beschleunigen.

Speicherhierarchie

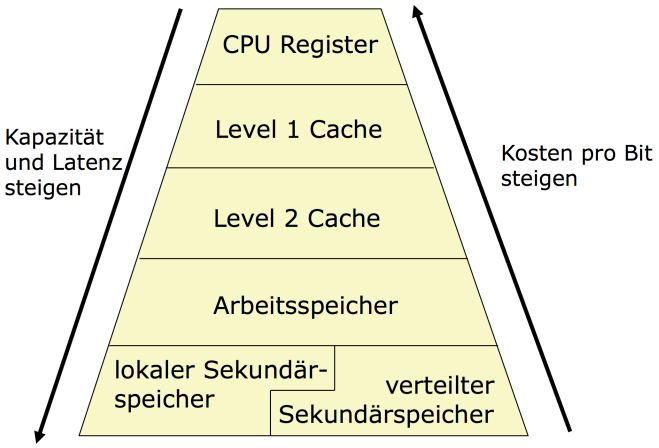

Abbildung 1.2: Speicherhierarchie

Die in Abbildung 1.2 dargestellte hierarchische Strukturierung des Speichers hat den Vorteil, dass mit verhältnismäßig geringen Kosten eine hohe Leistung erreicht werden kann. Sowohl die Leistung als auch der Preis pro Bit steigen in den höheren Ebenen der Speicherhierarchie. Entsprechend wird diese hierarchische Anordnung genutzt, um große Speicher günstig anbieten zu können, deren Daten bei einem Zugriff der Applikation in den kleinen aber schnellen oberen Ebenen der Speicherhierarchie zwischen gespeichert werden. Dieses *Caching* basiert auf den Prinzipien der zeitlichen und örtlichen Lokalität, d. h. wenn eine Applikation zu einem bestimmten Zeitpunkt auf eine Speicherstelle des Speichers zugegriffen hat, ist es sehr wahrscheinlich, dass sie wenig später eine Speicherzelle in kurzem örtlichen Abstand zur ersten Speicherzelle referenziert. Häufig referenzierte Daten werden also in höheren Ebenen der Speicherhierarchie abgelegt, um Zeit beim Datenzugriff zu sparen.

Caching

Memory-Wall des Hauptspeichers

Dennoch löst die Speicherhierarchie nicht das Problem der steigenden Leistungslücke zwischen den Hierarchieebenen. Anfang der 90er Jahre wurde mit dem *Memory-Wall-Problem* das Verhalten vorausgesagt, dass die Hauptspeicher-Leistung aufgrund der weniger starken Leistungssteigerung einen wachsenden Einfluss auf die für Applikationen erreichbare Leistung eines Rechensystems haben wird [7]. Das Memory-Wall-Problem basiert auf der Aussage, dass sich die durchschnittliche Zugriffszeit auf den Hauptspeicher eines Systems t_{avg} in Abhängigkeit von der Wahrscheinlichkeit eines Cache-Hits p_c (also der Wahrscheinlichkeit, ein gesuchtes Datum im Cache anzutreffen) und der Cache-Zugriffszeit t_c sowie der Speicherzugriffszeit t_m wie folgt berechnet [8]:

$$t_{avg} = p_c \cdot t_c + (1 - p_c) \cdot t_m$$

Unter den Annahmen, dass 1995 die Zeit zum Zugriff auf den Hauptspeicher ca. 4 mal so groß war wie ein Taktzyklus des Prozessors und damit die Zugriffszeit auf den Cache, dass die Cache-Miss-Rate bei unter einem Prozent liegt und die jährlichen Leistungssteigerungen bei rund 7 Prozent für Speicher und rund 80 Prozent für Prozessoren lagen, sagte Wulf [8] voraus, dass die durchschnittliche Anzahl der Prozessorzyklen pro Speicherzugriff zwischen den Jahren 2000 und 2005 von 1.52 auf 8.25 steigen würde. Tatsächlich stimmen die Größenordnungen dieser Werte bei einigen Applikationen mit den eingetretenen Werten überein. So besagt eine Studie aus dem Jahr 2004, dass bei einer Cray X1 ein Speicherzugriff durchschnittlich 5 bis 6 Prozessorzyklen benötigt [9]. Sobald die Anzahl der Zyklen eines Speicherzugriffs größer wird als die Anzahl der Prozessorzyklen, die zwischen einzelnen Speicherzugriffen für andere Aufgaben verwendet werden, beginnt die Leistung des Speichers die Applikations-Laufzeit zu dominieren, da die Applikation vor einem erneuten Speicherzugriff auf die Ergebnisse der bereits laufenden Anfrage warten muss. Die Anwendung trifft auf die *Memory Wall*. Wenn angenommen wird, dass 20 bis 40 % aller Instruktionen auf Speicher zugreifen [2] und jede Instruktion einen Prozessorzyklus dauert, liegen weniger als 4 Takte zwischen Instruktionen, die den Speicher referenzieren. Die *Memory Wall* des Hauptspeichers ist also längst erreicht. Durch die Hardwarehersteller wird dieses Problem häufig ignoriert, da auch bei einem dominanten Einfluss der Speichergeschwindigkeit eine Verbesserung der Applikationslaufzeit durch verbesserte Hardware erreicht werden kann. Der verringerten Auslastung der Recheneinheiten versuchen die Hardwarehersteller durch andere Konzepte entgegenzuwirken.

Die Wissenschaft sucht seit der Definition des Problems nach Möglichkeiten, wie die Memory Wall überwunden werden kann. Im Jahr 2004 wurde in der *Conference On Computing Frontiers* eine Spezialsitzung zum Memory-Wall-Problem einberaumt [10]. Ideen, die in dieser Sitzung vorgestellt wurden, betrafen zum Beispiel die Verringerung des Datenverkehrs zum Speicher [11] oder bessere Cache-Verdrängungsmechanismen [12]. Diese Ideen verringern die Probleme, sind aber keine endgültigen Lösungen, da die große Diskrepanz zwischen Rechen- und Speicherleistung bleibt. Noch in der November-Ausgabe 2008 der *IEEE Spectrum* wird das Memory-Wall-Problem als ein zentrales Problem bezeichnet, das die Leistung zukünftiger Supercomputer auf Basis von Mehr-Kern-Prozessoren begrenzen wird [13].

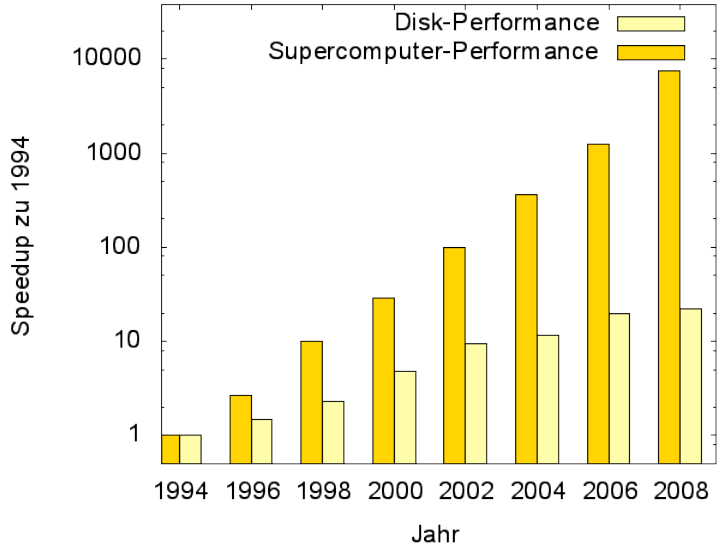

Abbildung 1.3: Entwicklung der Rechen- und Massenspeicherleistung

Entwicklung der Rechen- und Massenspeicher-leistung

Abbildung 1.3 zeigt die Entwicklung der Rechen- und der Massenspeicherleistung seit 1994. Es ist für die aufgetragenen Jahre jeweils der Leistungsgewinn (Speedup) gegenüber 1994 auf einer logarithmischen Skala dargestellt. Es ist zu sehen, dass die Rechenleistung seit 1994 um mehr als den Faktor 7500 gestiegen ist, während die Sekundärspeicherleistung handelsüblicher Festplatten lediglich um den Faktor 22 stieg. Die für die Grafik verwendeten Daten für die Rechenleistung basieren auf der Summe der Rechenleistungen (R_{max}) aller Rechner der Top500-Liste der leistungsstärksten Rechner der Welt in den jeweiligen Jahren [4], während die Daten für die Sekundärspeicher-Leistung dem Archiv der Fachzeitschrift *c't* und Online-Quellen entnommen wurden und jeweils die maximale Datentransferrate handelsüblicher Festplatten als Grundlage verwenden [14, 15, 16, 17, 18].

Memory Wall des Sekundär-speichers

Die gezeigten unterschiedlichen Leistungszuwächse zwischen Rechen- und Sekundärspeicherleistung lassen eine Memory Wall auch beim Sekundärspeicher vermuten. Um dies zu prüfen, werden im Folgenden Betrachtungen der Memory Wall des Hauptspeichers auf den Sekundärspeicher übertragen. Es wird so eine Erweiterung des bereits in der Literatur als *Memory-Wall-Problem des Hauptspeichers* bekannten Phänomens erreicht, um an dieser Stelle die Wichtigkeit einer Leistungsuntersuchung bei Sekundärspeichern zu unterstreichen.

Dazu wird die Berechnung der durchschnittlichen Speicherzugriffszeit so erweitert, dass der Hauptspeicher als Sekundärspeichercache eingesetzt wird. t_{sekc} gibt die durchschnittliche Zugriffszeit auf den Sekundärspeicher bei Verwendung des Hauptspeichers als Cache an, während t_{avg} die durchschnittliche Sekundärspeicherzugriffszeit bei Verwendung sowohl des Sekundärspeicher- als auch des Hauptspeichercaches darstellt. Die durchschnittliche Zugriffszeit auf den Sekundärspeicher t_{sekc} berech-

net sich aus der Zugriffszeit auf den Sekundärspeichercache (den Hauptspeicher), der mit der Sekundärspeicher-Cache-Hit-Wahrscheinlichkeit p_{sekc} stattfindet und der Zugriffszeit auf den Sekundärspeicher mit der entsprechenden Cache-Miss-Wahrscheinlichkeit.

$$\begin{aligned} t_{avg} &= p_c \cdot t_c + (1-p_c) \cdot \underbrace{(p_{sekc} \cdot t_m + (1-p_{sekc}) \cdot t_{sek})}_{t_{sekc}} \\ &= p_c \cdot t_c + (1-p_c) p_{sekc} \cdot t_m + (1-p_c)(1-p_{sekc}) \cdot t_{sek} \end{aligned}$$

Unter der Annahme, dass sich die Cache-Hit-Wahrscheinlichkeiten p_c und p_{sekc} in den nächsten Jahren kaum ändern und somit in der Formel als konstant angesehen werden, existiert nur ein Einfluss der Speicherhierarchie-Zugriffszeiten auf die Gesamtzugriffszeit. Wenn in den kommenden Jahren die Zugriffszeiten auf den Sekundärspeicher weiterhin weniger stark sinken, als die Zugriffszeiten auf die anderen Speicherhierarchie-Ebenen, wie das bisher der Fall war, ist ein steigender Einfluss der Zugriffszeit des Sekundärspeichers auf die gesamte Zugriffszeit und damit die Applikationsleistung erkennbar, weil dessen prozentualer Anteil an der gesamten Zugriffszeit steigt. Dieser Effekt wird noch durch die, verglichen mit dem Hauptspeichercache, geringere Cache-Hit-Wahrscheinlichkeit des Sekundärspeichercaches verstärkt. Diese Cache-Hit-Wahrscheinlichkeit kann zwar abhängig von dem Lastverhalten der ablaufenden Applikation stark variieren, wird aber in [19] bei Sekundärspeicherzugriffen einer parallelen Applikation in einer ressourcenreichen Umgebung mit 84,8 % angegeben und ist damit signifikant geringer als die Cache-Hit-Rate des Hauptspeichercaches, die in [8] mit über 99 % angenommen wird.

HPC und das Memory-Wall-Problem

Da Applikationen wesentlich seltener Sekundärspeicheranfragen als Hauptspeicherzugriffe durchführen, wird das Memory-Wall-Problem des Sekundärspeichers zuerst nur bei Applikationen zu bemerken sein, die besonders viele Zugriffe auf den Massenspeicher durchführen. Im Bereich des High-Performance-Computings gibt es zahlreiche Applikationen, die mit Datenmengen von mehreren Gigabyte oder Terabyte rechnen müssen. Dieser Wissenschaftsbereich ist deshalb einer der ersten, der sich intensiv mit der Verbesserung der Sekundärspeicherleistung der Großrechenanlagen beschäftigte. Moderne parallele Dateisysteme sind ein Ergebnis dieser Arbeiten. Diese erhöhen den Durchsatz der Sekundärspeichersysteme durch Nutzung mehrerer Datenspeicher, auf die parallel zugegriffen werden kann. Aber auch bei herkömmlichen Arbeitsplatzrechnern wird angesichts des sich ständig vergrößernden Problems ein stärker werdender Einfluss der Sekundärspeicherleistung auf die Applikationen deutlich werden.

Das Memory-Wall-Problem ist also nicht nur beim Hauptspeicher ein leistungsbegrenzender Faktor, es ist auch bei Sekundärspeicherzugriffen absehbar. Zukünftig ist mit einem steigenden Einfluss der Sekundärspeicherleistung auf die Applikations-Laufzeit zu rechnen. Es ist deshalb von großer Wichtigkeit, die Leistung moderner Sekundärspeicher genauestens zu analysieren, zu bewerten und zu optimieren, um dem Memory-Wall-Problem des Sekundärspeichers vorzubeugen.

1.4 Gegenstand und Aufbau dieser Arbeit

Gegenstand

Bisher wurde gezeigt, dass die Bedeutung der Leistungsanalyse von Sekundärspeichersystemen zukünftig immer stärker steigen wird. Der limitierende Einfluss der Sekundärspeicherleistung auf die Applikationslaufzeit wird zunehmen, wenn keine genauen Leistungsanalysen und Optimierungen stattfinden. Eine der wichtigsten Leistungsanalysemethoden ist das Benchmarking, bei dem Applikationen zur Vermessung der Leistung (Benchmarks) auf dem Rechensystem in seinem normalen Arbeitsumfeld verwendet werden. Aktuell existierende Benchmarks zur Messung der Sekundärspeicherleistung haben mehrere Probleme, wie Kapitel 2 aufzeigen wird. Es wird demonstriert, dass die meisten aktuellen Benchmarks in diesem Umfeld neben einigen anderen Einschränkungen das wesentliche Problem besitzen, dass ihre Ergebnisse nur für wenige Applikationen repräsentativ sind und nicht allgemein als Leistungsbewertung des Sekundärspeichers gelten können. Es ist Ziel der vorliegenden Arbeit, insbesondere dieses Problem durch die Definition einer neuartigen Benchmark-Architektur zur beheben. Die Komponenten der neuen Architektur erlauben neben einer umfangreichen Nutzerunterstützung auch die nutzergesteuerte Definition von Sekundärspeicher-Lasten insbesondere für parallele Sekundärspeichersysteme, wie sie im Bereich des Hochleistungsrechnens eingesetzt werden. In diesem Zusammenhang ist die Definition eines Lastmodells zur Beschreibung derartiger Lasten, die als Basis der eigentlichen Messung eingesetzt werden, ein wesentlicher Anteil dieser Arbeit.

Aufbau

Im Kapitel 2 werden dazu Grundlagen beschrieben, die für das Verständnis der weiteren Arbeit von Bedeutung sind. Neben der Beschreibung der grundlegenden Architektur von Sekundärspeichersystemen wird eine Einführung in die Leistungsanalyse von Rechensystemen und insbesondere das Benchmarking von Sekundärspeichersystemen gegeben. Es wird der aktuelle Stand der Technik vorgestellt, anhand dessen die Probleme aktueller Sekundärspeicher-Benchmarks formuliert werden. Ziel von Kapitel 2 ist die Definition von drei Kernfragen, die die gefundenen Probleme aktueller Benchmark-Architekturen klar umreißen und deren Beantwortung sich in den folgenden Kapiteln anschließen wird.

Kapitel 3 präsentiert detailliert die theoretische Basis der in dieser Arbeit definierten Benchmark-Architektur. Ausgehend von den für einen korrekten Benchmarking-Vorgang notwendigen Schritten, wird eine Architektur beschrieben, die aus Komponenten besteht, die fehlerfreies Benchmarking als Ziel haben. Die Komponenten ihrerseits benötigen theoretisches Basiswissen, das in diesem Kapitel erarbeitet wird. Dabei handelt es sich einerseits um ein Lastmodell zur Beschreibung paralleler Sekundärspeicher-Last und andererseits um die Bewertung der existierenden Benchmark-Messmethoden für den Einsatz in einem Sekundärspeicher-Benchmark anhand einer neuen Messmethoden-Klassifikation.

Das im Anschluss folgende Kapitel 4 stellt die Implementierung der Komponenten des Parallel I/O-Benchmarks (PRIOmark) vor, bevor im Kapitel 5 gezeigt wird, dass dieser

neue Benchmark in der Realität alle gestellten Anforderungen an einen Sekundärspeicher-Benchmark erfüllt.

Kapitel 6 fasst schließlich die Ergebnisse der Arbeit zusammen und liefert Hinweise für die zukünftige Entwicklung in diesem Forschungsbereich.

Kapitel 2

Grundlagen & Stand der Forschung

An investment in knowledge always pays the best interest.

– BENJAMIN FRANKLIN (1706 - 1790)

Bevor sich die vorliegende Arbeit eingehend mit dem Thema der Leistungsanalyse von Sekundärspeichern beschäftigt, werden einige Grundlagen vorgestellt, die für das weitere Verständnis der Arbeit von Bedeutung sind. Es wird der Aufbau und die Funktionsweise aktueller Sekundärspeichersysteme dargelegt, um sowohl deren Komplexität darzustellen als auch deren Einfluss auf die Sekundärspeicherleistung zu erläutern. Anschließend wird der aktuelle Stand der Forschung in den Themenbereichen Leistungscharakterisierung, Lastbeschreibungen und Leistungsanalysemethoden mit besonderem Fokus auf Benchmarks zur Vermessung der Sekundärspeicherleistung (I/O-Benchmarks) vorgestellt.

2.1 Sekundärspeichersysteme

Sekundärspeicher

Wie in Abbildung 1.2 dargestellt wird, bildet der Sekundärspeicher bezüglich des Hauptspeichers die nächstniedrige Ebene der Speicherhierarchie. Er speichert Daten permanent und erlaubt einen wahlfreien Zugriff. Beispiele für Sekundärspeicher sind Festplatten in lokalen Systemen, aber auch entfernte Speichersysteme wie Network Attached Storage (NAS)). Network Attached Storage, also Speicher, der an das Datennetz angeschlossen wird und Datenspeicher über klassische Datendienste wie NFS (Network File System) oder die Windows-Dateifreigabe zur Verfügung stellt [20], ist eine in vielen Unternehmen bereits eingesetzte Speicherlösung, die aufgrund der Nutzung des klassischen Datennetzes einige Nachteile aufweist. Mit separaten Netzwerken zur Speicheranbindung (Storage Area Network— SAN-Technologie) sollen diese Nachteile verschwinden [21].

Begriffsklärung I/O

Entgegen der allgemeinen Nutzung der Abkürzung I/O (Input/Output) in vielen Bereichen der Technik, insbesondere in Bezug auf Peripheriekomponenten von Computersystemen, bezieht sich der Begriff in dieser Arbeit ausschließlich auf den Sekundärspeicher. Das heißt, ein I/O-Zugriff ist ein Zugriff auf den Sekundärspeicher. Respektive wird die Leistung, die beim Zugriff auf den Sekundärspeicher erreicht wird, als I/O-Leistung oder I/O-Performance bezeichnet.

Die in Abschnitt 1.3 dargelegten Betrachtungen zur Memory Wall des Sekundärspeichers zeigen den Zusammenhang zwischen Sekundärspeicher und Hauptspeicher. Da der Hauptspeicher eines Systems als Cache für den Sekundärspeicher dient, kann die I/O-Leistungsermittlung nicht ohne Betrachtung der Leistungsfähigkeit von Hauptspeicher und Hauptspeichercaches durchgeführt werden. Damit wird die Sekundärspeicheranalyse schnell komplex und schwer nachvollziehbar. Der Systemanalysator muss sich ständig der gewollten und ungewollten Caching-Effekte des Systems bewusst sein. Neben den hardwareseitigen Effekten beim Zugriff auf Sekundärspeicher hat auch die Software (insbesondere die Betriebssystemsoftware) einen erheblichen Einfluss auf die Leistung des Sekundärspeichers. Auch sie besitzt eine Reihe von Ebenen, die angesprochen werden müssen, bevor eine Anforderung an den Sekundärspeicher tatsächlich von diesem bearbeitet wird. Im Folgenden wird dargelegt, wie das I/O-Subsystem eines typischen Betriebssystems aussieht, um im späteren Verlauf der vorliegenden Arbeit auftretende Phänomene besser zu verstehen.

2.1.1 Architektur eines I/O-Subsystems

Selbst lokale Sekundärspeicherzugriffe, die nicht den Zugriff auf Netzwerk-Ressourcen benötigen, sind aus Sicht einer Applikation komplexe Vorgänge. Auf den Datenträgern werden die Daten strukturiert in Dateisystemen abgelegt, deren Metadaten unter Umständen aufwändig auszuwerten und zu pflegen sind. Der Datenträgerzugriff besteht ebenfalls aus mehreren Phasen, die ihrerseits von der Hardware selbst abhängen. Um diese hohe Komplexität vor der Applikation zu verbergen, werden durch die Betriebssysteme Schnittstellen angeboten, die einen von der verwendeten Hardware abstrahierenden Zugriff ermöglichen. Wenngleich aus diesem Grund die Zugriffe aus Applikationssicht vereinfacht werden, führt das Betriebssystem bei jedem Zugriff eine ganze Reihe von Aktionen durch, deren Effizienz sich deutlich auf die I/O-Leistung von Applikationen auswirkt. Um die Komplexität des I/O-Zugriffs auch für das Betriebssystem handhabbar zu gestalten, wird das I/O-Subsystem in Form einer Schichtenarchitektur implementiert. Jede Schicht bearbeitet eine bestimmte Aufgabe, die zur Abspeicherung der Daten durchgeführt werden muss. Dies hat den Vorteil der leichteren Wartbarkeit einzelner Softwareschichten, kann aber durch zusätzliche Kopiervorgänge, die in einigen Fällen für den Datentransport zwischen den Schichten notwendig sind, zusätzliche Latenz bei der Datenverarbeitung verursachen.

Schichtenarchitektur

Abbildung 2.1 gibt einen Überblick über die verschiedenen Schichten des I/O-Subsystems, die im Einzelnen in den folgenden Abschnitten vorgestellt werden. Das in

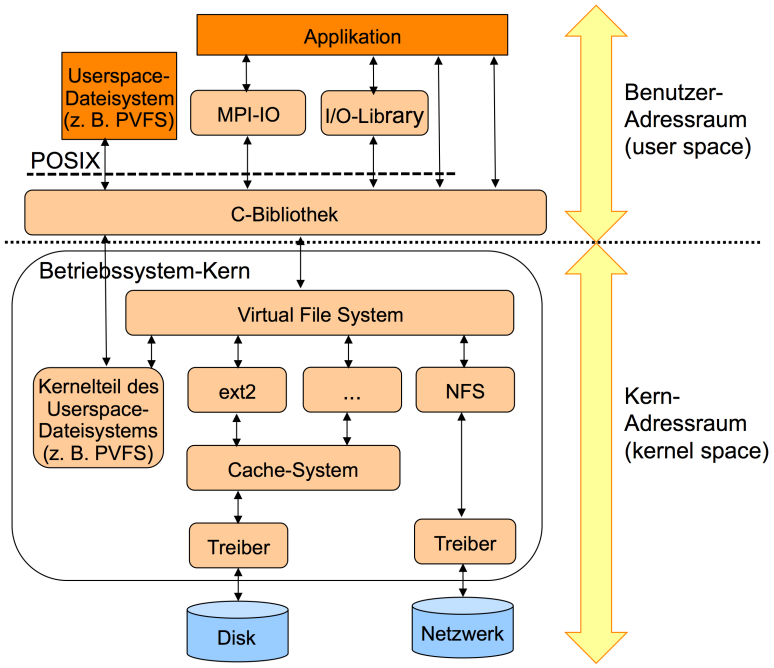

Abbildung 2.1: Schichten des I/O-Subsystems eines Linux-Betriebssystems

I/O-System bei monolithischen Betriebssystemen

Abbildung 2.1 dargestellte I/O-Subsystem ist in dieser oder ähnlicher Form in typischen Betriebssystemen mit monolithischem Kern implementiert. Betriebssysteme mit monolithischem Kern sind die häufigste Organisationsform von Betriebssystemen und sind dadurch gekennzeichnet, dass jede Funktion im Betriebssystem-Kern jede andere Funktion sehen und ausführen kann. Es existiert also keine Kapselung der einzelnen Funktionalitäten [22]. Die existierenden Linux-Derivate, aber auch viele BSD-basierte Betriebssysteme, besitzen eine I/O-Architektur der dargestellten Art [23, 24, 25]. In anderen Betriebssystemen kann die I/O-Architektur von der hier beispielhaft dargestellten in einzelnen Punkten abweichen. Größere Unterschiede gibt es bei Mikrokernbetriebssystemen, die den Betriebssystem-Kern nur mit den wichtigsten Funktionen ausstatten und große Teile der Funktionalität in herkömmliche Anwendungen im Benutzeradressraum auslagern [22]. Entsprechend finden sich bei diesen Systemen viele der Schichten, die in Abbildung 2.1 dem Betriebssystem-Kern zugeordnet sind, als Applikationen außerhalb des Kerns [26, 22]. Dennoch sind alle wichtigen Funktionalitäten wie Bibliotheken, Dateisysteme, Dateisystemcache und Treiber auch bei Mikrokernbetriebssystemen vorhanden, so dass die Anzahl der Softwaresysteme, die für eine Sekundärspeicheranfrage durchlaufen werden muss, ähnlich zu der Anzahl in monolithischen Betriebssystemen ist.

I/O-System bei Mikrokernbetriebssystemen

2.1.2 I/O-Schnittstellen

Aus Sicht einer Applikation findet ein Zugriff auf den Sekundärspeicher über eine einfach nutzbare Programmierschnittstelle (API - Application Programming Interface)

statt. Neben proprietären I/O-Schnittstellen, die in Abbildung 2.1 als *I/O-Library* bezeichnet sind, gibt es standardisierte I/O-Schnittstellen wie POSIX-I/O (Portable Operating System Interface-I/O), welches durch die C-Bibliothek des Systems zur Verfügung gestellt wird. MPI-IO (Message Passing Interface-I/O), eine weitere standardisierte I/O-Schnittstelle, wird durch das separat im System vorliegende MPI-System verwendet. Beide Schnittstellen werden im Folgenden kurz vorgestellt.

POSIX

POSIX-I/O

POSIX-I/O. Als Teil des POSIX-Standards (Portable Operating System Interface) definiert POSIX-I/O die Standard-Schnittstelle zum Zugriff auf Sekundärspeicher, die in klassischen UNIX-Systemen Einsatz findet. Die POSIX-Applikations-Schnittstelle ist durch die *IEEE* (Institute of Electrical and Electronics Engineers, Inc.) in Zusammenarbeit mit der *Open Group* in verschiedenen Standards aus der Gruppe der IEEE1003.X festgelegt worden. Seitdem 1990 die ersten POSIX-Standards verabschiedet wurden, werden in regelmäßigen Zeitabständen neue Versionen veröffentlicht [27, 28]. POSIX-I/O als Teil des POSIX-Standards definiert Funktionen zum Navigieren in und Manipulieren von Verzeichnisstrukturen sowie zum Öffnen, Schließen, Lesen und Schreiben von Dateien. Dabei werden wahlfreie Zugriffe (d. h. Zugriffe auf beliebige Dateipositionen) ebenso wie sequentielle Zugriffe (d. h. in der Datei aufeinander folgende Zugriffe) unterstützt. Zugriffe können blockieren, also die Applikation warten lassen, bis der eigentliche Zugriff durch das I/O-Subsystem durchgeführt wurde, oder asynchron sein, d. h. die Ausführung der Applikation auch ohne Fertigstellung der I/O-Anfrage weiterführen [29].

MPI

MPI-IO

MPI-IO. In der Version 2 des MPI-Standards (Message Passing Interface) wurde MPI-IO als eine I/O-Schnittstelle für den Bereich des High-Performance-Computings (HPC) definiert. MPI beschreibt eine Programmierschnittstelle für die Programmiersprachen Fortran und C, mit der verteilte Applikationen implementiert werden können. MPI liegt als zusätzliche Schicht zwischen Betriebssystem und Applikation vor. Es definiert Funktionen, die die Kommunikation mehrerer parallel arbeitender Prozesse einer verteilten Applikation ermöglichen. Da die Kommunikation über Rechnergrenzen hinweg funktioniert, ist eine Ausführung der Prozesse auf dedizierten Knoten von Clustern oder anderen Hochleistungsrechnern möglich. MPI sieht Möglichkeiten von (Multi-)Punkt-zu-(Multi-)Punkt-Kommunikation vor und deckt damit ein breites Spektrum an möglichen Kommunikationsformen ab. MPI-IO hat als Ziel [30, 31], den parallelen Zugriff mehrerer Prozesse einer Applikation auf gemeinsame Daten zu optimieren. In einem typischen Szenario einer HPC-Applikation schreiben die einzelnen Prozesse der Applikation ihre vollständigen Daten parallel auf den Sekundärspeicher, um im Falle eines Systemfehlers einen konsistenten Status vom Speicher zurücklesen zu können. Diese unter Umständen sehr große Datenmenge kann von modernen parallelen Dateisystemen sehr effizient geschrieben werden, wenn die Dateisystemschnittstelle Unterstützung dafür bietet [32].

2.1.3 Dateisysteme

Die C-Bibliothek des Rechensystems, die sowohl vom MPI-System als auch von Applikationen selbst genutzt wird, kommuniziert über die Betriebssystemschnittstelle mit dem Betriebssystemkern. I/O-Aufrufe der Anwendung werden auf Aufrufe des Dateisystems abgebildet, das die strukturierte Speicherung der Daten übernimmt. Um die Daten zu strukturieren, werden ihre Bytefolgen in feste Organisationsformen mit Metainformationen zum leichten Finden und zum Schutz vor unbefugten Zugriff organisiert. Diese Dateien sind ihrerseits in Verzeichnissen strukturiert. Verzeichnisstrukturen können flach sein (d. h. es gibt nur ein Verzeichnis für alle Dateien) oder hierarchisch organisiert werden. Moderne Dateisysteme nutzen im Allgemeinen die zweite Form der Organisation, um den Applikationen mehr Flexibilität bei der Strukturierung der Daten zu geben [22].

Dateien und Verzeichnisse

Da viele Betriebssysteme eine parallele Nutzung unterschiedlicher Dateisysteme unterstützen, wird ein Applikations-I/O-Aufruf, wie in Abbildung 2.1 dargestellt, zuerst auf die virtuelle Dateisystemschicht abgebildet. Das VFS (Virtual File System) dient als vom tatsächlich verwendeten Dateisystem abstrahierende Schnittstelle und stellt immer gleiche Methoden zum Zugriff auf Dateien zur Verfügung. Das VFS wählt abhängig vom Speicherort, auf den sich die I/O-Anfrage bezieht, die Funktionen des dazugehörigen Dateisystems aus. Auf diese Art und Weise können auf einem Rechensystem unterschiedliche Verzeichnisse von jeweils anderen Dateisystemen bedient werden. Beispielsweise können private Nutzerverzeichnisse im Verzeichnis `/users` von einem Dateiserver im Netzwerk bezogen werden, während alle anderen Daten lokal auf dem Rechensystem vorliegen. Dateisysteme, die lokale Speicher verwenden, nutzen statt des direkten Treiberzugriffs das systemweite Cache-System, das häufig zugegriffene Sekundärspeicherdaten im Hauptspeicher des Systems zwischenspeichert. Netzwerkdateisysteme hingegen nutzen den Netzwerk-Treiber direkt, um Dateninkonsistenzen, die durch das Caching entstehen, zu vermeiden. Neben Dateisystemen, die im Betriebssystemkern implementiert sind, gibt es auch Dateisysteme, die als Nutzerapplikationen realisiert werden. Das parallele Dateisystem PVFS (Parallel Virtual File System) ist beispielsweise auf diese Art und Weise organisiert. PVFS besteht aus einer Applikation im Benutzeradressraum und einem Kernelteil. Der Kernelteil hat nur eine einfache Funktion: Wie in Abbildung 2.1 dargestellt, gibt er sämtliche Anfragen direkt (oft über eine proprietäre Schnittstelle) an den Userspace-Anteil des Dateisystems zurück und ermöglicht daher die Nutzung des Dateisystems über die standardisierte POSIX-Schnittstelle [33].

Virtual File System

Das Wissen um die Funktionsweise von Dateisystemen ist in weiten Teilen der Arbeit Voraussetzung zum Verständnis. Es werden deshalb im Folgenden wichtige Dateisysteme kurz vorgestellt.

Lokale Dateisysteme

tabellenbasierte Dateisysteme

Die ersten existierenden Dateisysteme legten Daten auf lokal am Rechner installierten Sekundärspeichern ab. Zu diesem Zweck wurden zahlreiche Dateisysteme entwickelt, die in tabellenbasierte und Inode-basierte Dateisysteme unterschieden werden. Das in Microsoft-Betriebssystemen verwendete FAT-Dateisystem (File Allocation Table) ist ein tabellenbasiertes Dateisystem, das in einer Tabelle in verketteten Strukturen die zu einer Datei gehörenden Blöcke speichert, wobei der erste Block einer Datei in einem Verzeichnis, das als spezielle Datei organisiert ist, abgelegt wird [22, 34]. Damit ist ein sequentielles Navigieren in der Datei sehr effizient möglich. Wahlfreier Zugriff hingegen ist mit Leistungseinbußen verbunden, da für den Zugriff auf eine bestimmte Position innerhalb der Datei die Blockliste durchlaufen werden muss.

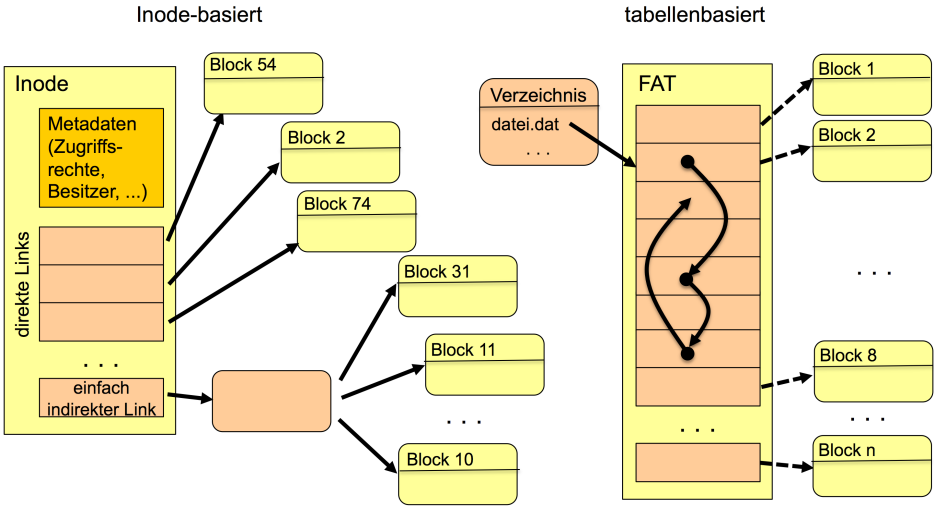

Abbildung 2.2: Dateien in Inode- und tabellenbasierten Dateisystemen

Inode-basierte Dateisysteme

Inode-basierte Dateisysteme, wie sie klassischerweise in UNIX-Systemen eingesetzt werden, umgehen dieses Problem. Sie besitzen bessere Leistungsmerkmale bei einem wahlfreien Zugriff, verursachen jedoch bei anderen Zugriffsmustern, wie dem Einfügen von neuen Blöcken innerhalb einer Datei, einen höheren Aufwand als tabellenbasierte Systeme. Bei Inode-basierten Dateisystemen wird pro Datei eine Inode mit der Größe eines Blocks zur Speicherung der Metainformationen verwendet. Jede Inode enthält neben Metainformationen zu der Datei direkte Verweise auf zugehörende Datenblöcke des Sekundärspeichers und ermöglicht deshalb einen schnellen wahlfreien Zugriff. Da die Anzahl möglicher direkter Verweise in einer Inode aufgrund der festen Inode-Größe beschränkt ist, gibt es pro Inode wenige ein- oder mehrfach indirekte Verweismöglichkeiten, die Blöcke referenzieren, die ihrerseits auf Datenblöcke oder weitere indirekte Blöcke verweisen. Weitverbreitete Inode-basierte Dateisysteme sind das 2nd extended File System (ext2) [25] und dessen Nachfolger, das 3rd extended File System, die beide in Linux-Systemen weit verbreitet sind. Abbildung 2.2 verdeutlicht grafisch die Organisation von Dateien in beiden genannten Dateisystemformen. Lokale Dateisysteme sind eine klassische Art der Datenspeicherung, die eine gemeinsame

Sicht mehrerer Rechner auf das Dateisystem nicht erlauben. Netzwerkdateisysteme, wie sie im Folgenden vorgestellt werden, besitzen diese Einschränkung nicht.

NFS

Das Network File System (NFS) wurde in den 80er Jahren durch Sun Microsystems entwickelt, um festplattenlose Rechner über ein Netzwerk mit einem Dateisystem zu versorgen. NFS ermöglicht für Rechner einen entfernten Zugriff auf ein Dateisystem, so als ob es auf dem Rechner lokal zur Verfügung stände. Das Dateisystem wird durch ein NFS-Serverprogramm im Netz propagiert, das im Allgemeinen auf einem extra für diesen Zweck im Netzwerk installierten Fileserver abläuft. NFS als ein sehr früher Vertreter von Netzwerkdateisystemen hatte zahlreiche Probleme, die in neuen Versionen teilweise behoben wurden. So wurden die Server der ersten NFS-Versionen zustandslos konzipiert – sie merkten sich also keine Informationen über die Clients. Dies hatte die Vorteile, dass der Server weniger Daten verarbeiten und speichern musste und dass er im Fall eines Problems einfach während des Betriebes neu gestartet werden konnte, ohne Verbindungen auf Seiten der Clients zu zerstören. Der wesentliche Nachteil ist allerdings die fehlende Transparenz auf Seiten der Clients. Beispielsweise gab es statt eines `open`-Aufrufes zum Öffnen einer Datei auf dem Server nur einen `lookup`-Aufruf, der Informationen zum Speicherungsort der Datei (Inode und Gerätenummer) zurücklieferte. Bei jedem Zugriff musste der Client also neben einer absoluten Zugriffsposition innerhalb der Datei auch die durch den `lookup`-Aufruf zurückgegebenen Daten an den Server senden. Neben diesem Mehraufwand bei den Funktionsaufrufen wurde in frühen Versionen von NFS ausschließlich UDP als Transportprotokoll verwendet, was nur den Einsatz in lokalen Netzen möglich machte. Von Anfang an wurde bei NFS nur **ein** Server vorgesehen [35, 36, 37], der Daten für die Clients zur Verfügung stellt, was diesen zu einem kritischen Ausfallpunkt und einem Flaschenhals macht, der bei einer großen Anzahl von Clients starke Leistungseinbrüche verzeichnen kann. Aktuell ist die Version 4 des Dateisystems, die einige der genannten Probleme behebt: NFS-Server der aktuellen Protokollversion sind zustandsbehaftet, so dass sich das Dateisystem aus Nutzersicht transparent darstellt und TCP als Transportprotokoll unterstützt wird [38].

Probleme von NFS

Eine klassische NFS-Installation in einem Netzwerk besitzt allerdings nach wie vor nur einen NFS-Server. Zwar gibt es Möglichkeiten, die Daten von NFS-Servern zu spiegeln und durch mehrere Server zur Verfügung zu stellen, um die damit verbundenen Probleme zu verringern – da diese Möglichkeiten aber nicht durch das Protokoll vorgesehen werden, können leicht Inkonsistenzen der Daten auf den gespiegelten Servern entstehen, wenn mehrere Nutzer unabhängig voneinander auf unterschiedlichen Datenbeständen arbeiten. Um diese Probleme zu beheben, und Dateisysteme zu schaffen, die sicher und performant den Zugriff mehrerer Clients auf gleiche Datenbestände realisieren, wurden neue Dateisysteme entwickelt, deren Konzepte mehrere Server unterstützen.

GFS

Das Global File System (GFS) von *RedHat* wurde als Cluster-Dateisystem für Linux entwickelt und unterstützt Anwendungsszenarien mit unterschiedlichen Anforderungen. GFS kann als Alternative zu NFS verwendet werden, bietet aber auch weitreichende Konfigurationsmöglichkeiten, um Sekundärspeicher für Hochleistungsrechner zur Verfügung zu stellen. Dabei können mehrere GFS-Server verwendet werden, die nicht nur höhere Performance durch parallelen Zugriff versprechen, sondern auch größere Ausfallsicherheit. GFS unterstützt eine nahtlose Integration von Speicherlösungen wie Storage Area Networks, die z. B. mit dem Hochgeschwindigkeitsdatennetz Fibre-Channel angeschlossen sind, aber auch die Verknüpfung der lokalen Speicher verschiedener GFS-Server zu einem gemeinsamen Speicher. In jedem Fall stellt GFS allen verbundenen Knoten eine gemeinsame Dateisystemsicht zur Verfügung und kann je nach den Möglichkeiten der darunterliegenden Speicherlösung hoch skalierbare und ausfallsichere Sekundärspeicherlösungen schaffen [39, 40].

Lustre

Lustre von *Cluster File Systems, Inc.* ist ein weiteres bekanntes Cluster-Dateisystem für Linux. Lustre erlaubt die Duplizierung aller Komponenten einer Lustre-Installation und vermeidet so kritische Punkte, die bei Fehlfunktion einen kompletten Systemausfall zur Folge haben (single point of failure). Ein Lustre-System besteht neben den Clients, die auf das Lustre-Dateisystem zugreifen, aus Metadatenservern und Object-Storage-Targets (OST). Dateien werden in der Lustre-Semantik als Objekte bezeichnet, deren Namen und Verzeichnisinformationen von den Metadatenservern gespeichert werden. Die eigentlichen Daten der Objekte werden auf den Object-Storage-Targets gehalten. Die OSTs führen also die eigentlichen I/O-Zugriffe durch und stellen dem Lustre-System so eine Abstraktion des darunterliegenden Speichersystems zur Verfügung. Prinzipbedingt können deshalb sämtliche Speicherlösungen in Lustre eingesetzt werden, die auch als lokale Speicherlösungen in den OSTs verwendet werden. Damit ist wie bei GFS die Verwendung von SANs ebenso möglich, wie der Einsatz lokaler Speicher [41, 42].

PVFS

Das Parallel Virtual File System (PVFS) des *Parallel Architecture Research Laboratory* der *Clemson University* ist ein Dateisystem, das für den parallelen Zugriff mehrerer Knoten auf wenige Dateien optimiert wurde. Um die Bandbreite beim Zugriff auf eine Datei durch mehrere Clients zu erhöhen, werden die Dateien auf mehrere Knoten aufgeteilt. Abbildung 2.3 zeigt dieses Striping der PVFS-Dateien an einem Beispiel. Eine PVFS-Datei wird in 9 Segmente gleicher Größe zerlegt, die auf den I/O-Knoten 4 bis 6 des Systems gespeichert werden. Jeder I/O-Knoten nimmt dabei mehrere Blöcke auf,

die zyklisch auf die drei beteiligten I/O-Knoten aufgeteilt werden. Auf den I/O-Knoten werden die Daten in Dateien der jeweils lokal vorhandenen Dateisysteme abgelegt, so dass PVFS als alleiniges Dateisystem in einem verteilten Rechnersystem nicht einsetzbar ist. Jeder I/O-Knoten besitzt als Serverkomponente einen Hintergrundprozess, den

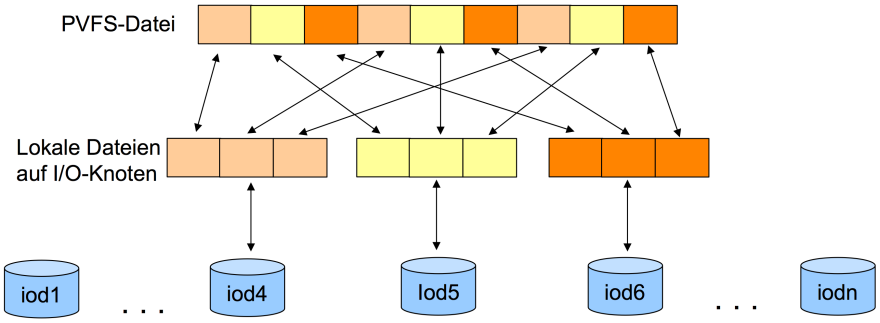

Abbildung 2.3: Striping in einem PVFS-Dateisystem

sogenannten I/O-Daemon, der von den Clients zur Kommunikation kontaktiert wird. Clients nutzen zur Kommunikation mit dem I/O-Daemon entweder eine PVFS-Client-Bibliothek, oder andere Zugriffsarten, wie den Zugriff per POSIX-I/O. Neben einer Schnittstellenimplementierung für POSIX-I/O gibt es auch eine MPI-IO-Schnittstelle. PVFS besitzt einen zentralen Metadaten-Server, der die Metadaten zu Dateien (wie zum Beispiel deren Verteilung auf die einzelnen I/O-Knoten) speichert und überwacht. Dieser Metadaten-Server ist der einzige potentielle Flaschenhals des Systems, weil er nicht mehrfach instanziiert werden kann. Allerdings ist ein Zugriff auf den Metadatenserver nur während des Öffnens einer Datei notwendig. Datentransfers zwischen Client und I/O-Knoten finden ohne Zuhilfenahme des Metadatenservers statt [43].

Metadaten-server

2.2 Leistungsanalyse von Rechensystemen

Nachdem grundlegendes Wissen um die Funktionsweise und den Aufbau des I/O-Subsystems moderner Rechensysteme vermittelt wurde, widmet sich dieser Abschnitt der Arbeit nun der Leistungsanalyse von Rechensystemen.

Vor jeder Leistungsanalyse eines Rechensystems sind durch den Analysten die Bedingungen des Umfeldes der Analyse zu klären. Drei wesentliche Probleme sind dabei zu lösen:

1. *Leistungscharakterisierung:* Physikalisch ist die Leistung als Arbeit pro Zeit (bzw. Energie pro Zeit) definiert. Auch bei einem Rechensystem kann die (elektrische) Leistung in Form der verbrauchten elektrischen Energie pro Zeit angegeben werden. Eine derartige Definition zur Beschreibung der Leistungsfähigkeit eines Computers zu verwenden, ist aber wenig sinnvoll, da keine Aussage über die vollbrachten Aufgaben des Rechensystems getroffen wird. Ziel der

Leistungsanalyse ist es, ein charakterisierende Größe zu erhalten, anhand der eine Vorstellung über die realisierbaren Aufgaben pro Zeiteinheit erreicht wird. Es ist also die Frage zu klären, wie die Leistung eines Rechensystems bzw. einer Systemkomponente adäquat und vergleichbar dargestellt werden kann? Welche physikalischen Größen werden durch die Analyse ermittelt?

2. *Lastbeschreibung:* Die Leistungsanalyse eines Rechensystems soll dem Nutzer eine Vorstellung von der Menge der möglichen bearbeitbaren Aufgaben pro Zeit erbringen. Demnach muss das System als Grundlage der Leistungsanalyse bestimmte Aufgaben bearbeiten, die als Grundlage der Analyse dienen. Die Art und Menge der Aufgaben wird als Last bezeichnet. Unter welcher Last soll die Analyse stattfinden? Da jede Last andere Analyseergebnisse erzeugen kann, ist eine genaue Spezifikation der Last notwendig. Wie kann diese Last so spezifiziert werden, dass sie nachvollziehbar wieder erzeugt werden kann?

3. *Methoden der Leistungsanalyse:* Welche Möglichkeiten gibt es, die Leistung eines Rechensystems zu ermitteln? Welche sind für bestimmte Einsatzzwecke nutzbar?

Einen Überblick über den aktuellen Stand der Forschung zu den drei Themenbereichen Leistungscharakterisierung, Lastbeschreibung und Leistungsanalyse-Methoden folgt in den nächsten drei Abschnitten.

2.2.1 Leistungscharakterisierung

Um Leistungsanalysen von Computersystemen durchführen zu können, ist die Existenz von genauen Möglichkeiten der Leistungscharakterisierung notwendig, d. h. es muss definiert werden, was gemessen wird und wie die Messergebnisse dargestellt werden können. Zur Darstellung dienen Performance-Metriken, die häufig physikalische Größen sind.

physikalische Leistung

Physikalisch ist die durchschnittliche Leistung als Arbeit pro Zeit definiert.

$$P = \frac{W}{t} = \frac{E}{t}$$

Leistung eines Rechensystems

Mit dieser physikalischen Definition kann die Leistungsfähigkeit eines Rechensystems kaum charakterisiert werden, da die (elektrische) Leistung eines Computers keine Aussagen zu der Anzahl der mit dem System zu bearbeitenden Aufgaben pro Zeiteinheit ermöglicht. Die Leistungsfähigkeit eines Rechensystems (bzw. dessen Leistung) muss also allgemeiner definiert werden. Sie spezifiziert die Menge bzw. Größe der mit einem Rechensystem lösbaren Aufgaben bzw. Probleme in einer bestimmten Zeitspanne. Die dafür verwendete Leistungsmetrik kann je nach zu vermessendem System oder Systemkomponente variieren.

Dennoch können der physikalischen Leistungsdefinition Anleihen entnommen werden, um sinnvolle Leistungsmetriken für Rechensysteme zu definieren: Durchsatzmetriken, die die Anzahl von „erledigten Aufgaben" pro Zeiteinheit charakterisieren, sind geeignete Leistungsmetriken. Ebenso können benötigte Zeiten zur Durchführung einer Aufgabe Leistungsmetriken darstellen. Im Folgenden werden entsprechende Metriken vorgestellt, die das Ziel haben, die Leistungsfähigkeit bestimmter Systemkomponenten oder ganzer Systeme zu charakterisieren.

Leistungscharakterisierung über technische Daten

Einfache Leistungsmetriken stellen technische Daten wie die Taktfrequenz der CPU oder die Anzahl der vorhandenen parallel arbeitenden CPUs dar. Während derartige Leistungsparameter bei wenig komplexen Systemen durchaus ausreichend sind, um eine gute Vorstellung über die Leistungsfähigkeit zu bekommen, sind sie bei komplexen Systemen wie Hochleistungsrechnern nur noch ein Hinweis auf die erreichbare Performance, da wirkliche Leistungswerte bei komplexen Systemen erheblich vom Zusammenspiel der vorhandenen Komponenten der Architektur abhängen. Eine Möglichkeit, die Komplexität der gesamten Rechensysteme zu erfassen, ist die Kombination mehrerer einfacher Metriken zu einer komplexen. Dies wird zum Beispiel bei Rechnersystem-Taxonomien durchgeführt, von denen zwei wichtige im Folgenden vorgestellt werden. Die Taxonomien haben zwar das Ziel, Rechensysteme nach Merkmalen zu klassifizieren, geben jedoch aufgrund der Merkmale auch Hinweise auf die Leistungsfähigkeit der Systeme.

Taxonomie nach Flynn

Eine der bekanntesten Rechnersystem-Taxonomien ist die Taxonomie nach Flynn [44], die zwei einfache Metriken miteinander kombiniert. Flynns Taxonomie unterteilt Rechner in vier Kategorien in Abhängigkeit der Anzahl paralleler Rechen- und Steuerwerke:

SISD. Single-Instruction-Stream / Single-Data-Stream-Rechner verarbeiten einen Instruktions- und einen Datenstrom. Klassische Von-Neumann-Rechner zählen in diese Kategorie.

SIMD. Single-Instruction-Stream / Multiple-Data-Stream-Rechner verarbeiten viele Daten parallel mittels eines Instruktionsstroms, wie es z. B. bei Vektorrechnern praktiziert wird.

MISD. Multiple-Instruction-Stream / Single-Data-Stream-Rechner sind in der Praxis nicht vertreten. In der Literatur werden gelegentlich Rechner dieser Kategorie zugeordnet, die Pipelining unterstützen, da bei diesen mehrere Instruktionen in unterschiedlichen Phasen auf gleichen Daten arbeiten können.

MIMD. Multiple- Instruction- Stream / Multiple- Data- Stream- Rechner sind Rechner mit mehreren Funktionseinheiten, die unabhängig voneinander unterschiedliche Instruktionen mit unterschiedlichen Daten bearbeiten. Klassische Beispiele sind Clustercomputer oder Multiprozessorsysteme.

Die Taxonomie nach Flynn kann somit einen Hinweis auf die mögliche Parallelität im System geben. Je höher die Parallelität im System, desto mehr Daten können pro

Zeiteinheit bearbeitet werden, was auch einen Hinweis auf die Leistungsfähigkeit des Systems gibt. Dennoch ordnet diese Taxonomie sämtliche Rechensysteme in nur vier Kategorien ein, was als Leistungsbewertung völlig unzureichend ist.

Erlanger Klassifikationsschema

Komplexer als die Taxonomie nach Flynn ist das Erlanger Klassifikationsschema nach Händler (*Erlangen Classification Scheme* - ECS [45]), das neben der Anzahl unabhängig arbeitender paralleler Rechen- und Steuerwerke auch die Parallelität durch Pipelining auf verschiedenen Ebenen einbezieht. Die Taxonomie definiert das folgende 3-Tupel.

$$
\begin{aligned}
\mathrm{t} &= (\mathrm{k} \times \mathrm{k}', \mathrm{d} \times \mathrm{d}', w \times w') \\
\mathrm{k} &- \text{Anzahl der Steuerwerke} \\
\mathrm{k}' &- \text{Anzahl der Teilsteuerwerke, die Teile der Gesamtaufgabe ausführen (Makropipelining)} \\
\mathrm{d} &- \text{Anzahl der Rechenwerke pro Steuerwerk} \\
\mathrm{d}' &- \text{Anzahl der Teilrechenwerke zur parallelen Berechnung} \\
w &- \text{Anzahl der Bits, die parallel verarbeitet werden} \\
w' &- \text{Anzahl der Pipelinestufen zur Bearbeitung eines Befehls (Befehlspipelining)}
\end{aligned}
$$

Mehrere 3-Tupel zur Charakterisierung eines Rechensystems können in der Taxonomie nach Händler auch mittels Operatoren kombiniert werden, um so komplexere Rechensysteme beschreiben zu können, die aus heterogenen Einzelsystemen bestehen. So kann z. B. definiert werden, dass die Komponenten nur gemeinsam (AND: $*$) oder unabhängig voneinander betrieben (OR: $+$) werden können. Die Quantifizierung einzelner Eigenschaften der Rechensysteme beim Erlanger Klassifikationsschema erlaubt im Vergleich zu Flynns Ansatz einen ersten Eindruck von der Leistungsfähigkeit des Systems.

Systemeigenschaften

Zusammengesetzte Metriken, wie das Klassifikationsschema nach Händler können ein Verständnis für die Leistung des Gesamtsystems liefern, sind aber oftmals schwer zu interpretieren. Eine weitere Möglichkeit bieten deshalb einfache Metriken, die durch die Vermessung des gesamten komplexen Systems ermittelt werden. So wird von den Eigenschaften einzelner Komponenten abstrahiert und das System im Gesamtkontext betrachtet. In [1] werden häufig benutzte Performance-Metriken dieser Art beschrieben:

Antwortzeit. Die Antwortzeit ist definiert als die Zeitspanne, die zwischen einer Nutzeranfrage und der Systemantwort vergeht. Da sowohl während der Nutzereingabe zur Formulierung der Anfrage als auch während der Systemausgabe zur Darstellung der Systemantwort Zeit vergeht, gibt es zwei unterschiedliche Definitionen der Antwortzeit. Die Zeit der Nutzereingabe wird bei beiden Definitionen ignoriert, da sie oft nicht durch die Leistungsfähigkeit des Systems beeinflusst wird. Allerdings ist die Zeit, die das System zur Ergebnisausgabe benötigt,

häufig ein Leistungsmerkmal und wird deshalb bei einer der beiden möglichen Antwortzeiten betrachtet. Die Antwortzeit ist also die Zeit, die ein System vom Absenden der Nutzeranfrage bis zum Ende der Ausgabe des Ergebnisses benötigt. Die zweite Definition der Antwortzeit vernachlässigt die Zeit der Ausgabe des Systems und ist deshalb definiert als die Zeitspanne zwischen Absenden einer Nutzeranfrage und Eintreffen des Ergebnisses. Die *Turnaround-Zeit* ist eine Antwortzeit nach der ersten Definition. Sie gibt also die Zeitspanne zwischen Absenden eines Batchjobs und der Darstellung seiner Ausgabe an.

Reaktionszeit. Während die Antwortzeit den gesamten Zeitraum beinhaltet, der zur Bearbeitung einer Anfrage erforderlich ist, besteht die Reaktionszeit nur aus dem Teil der Antwortzeit, die zwischen Nutzeranfrage und Start der Aktion vergeht, die die Nutzeranfrage bewirkt.

Stretch-Faktor. Dieser Performance-Wert gibt den Quotienten der Antwortzeit eines System bei einer bestimmten Last und der Antwortzeit des Systems bei minimaler Last an. Er ist damit ein Wert, der die Lastabhängigkeit der Leistung des Systems charakterisiert.

Durchsatz. Der Durchsatz eines Systems ist definiert als die Anzahl von Systemanfragen, die innerhalb einer Zeiteinheit bearbeitet werden. Abhängig vom System wird der Durchsatz unterschiedlich angegeben. Bei Batch-Systemen zur Verarbeitung einer großen Anzahl von unabhängigen Aufgaben (Jobs) ist er als Jobs pro Sekunde definiert, bei CPUs kann er in MIPS angegeben werden. Bei Netzwerken wird der Durchsatz häufig in Megabits pro Sekunde und bei Sekundärspeichern in Megabytes pro Sekunde gemessen. Typischerweise erhöht sich der Durchsatz eines Systems mit seiner Last bis zu einem Maximum, ab dem der Durchsatz dann wieder abfällt.

Effizienz. Die Effizienz gibt prozentual an, mit welchem Grad die Kapazität eines Systems oder einer Systemkomponente für die Zielstellung des Systems genutzt wird. Als Grundlage zur Ermittlung der Effizienz können Metriken verwendet werden, die eine Kapazität darstellen. Ziel ist zu erkennen, welcher Anteil der Kapazität durch Verwaltung verwendet wird. Bei Netzwerken wird oftmals der Durchsatz zur Berechnung der Effizienz verwendet. Wenn eine Netzwerkverbindung also nach Abzug aller Verwaltungsdaten einen Durchsatz von 80 MBit/s bei einem maximal möglichen Durchsatz von 100 MBit/s erreicht, entspricht dies einer Effizienz von 80 Prozent.

Auslastung. Wenn dargestellt werden soll, mit welchem Anteil eine Kapazität insgesamt verwendet wird, ist die Auslastung eine häufig verwendete Metrik. Sie wird beispielsweise für die Darstellung der Ausnutzung der Rechenzeit einer Rechenanlage verwendet. In diesem Fall gibt sie an, mit welchem Anteil von der Gesamtzeit ein System belastet war, wobei im Gegensatz zur Effizienz auch die Belastung durch Verwaltung einbezogen wird. Ähnlich verhält es sich bei

Speichern. Ein Speichersystem, das zu 100 Prozent ausgelastet ist, besitzt keinen freien Speicher mehr.

Zuverlässigkeit. Diese Metrik wird oft in Form einer Wahrscheinlichkeit eines auftretenden Fehlers oder mittels einer Zeitangabe der fehlerfreien Funktion (MTBF - *Mean Time Between Failure*) angegeben. Ein System mit einer MTBF von 2 Jahren arbeitet im Schnitt 2 Jahre fehlerfrei.

Verfügbarkeit. Die Verfügbarkeit wird als prozentualer Zeitanteil angegeben, in dem der Dienst des Systems dem Nutzer zur Verfügung steht. Netzwerkprovider geben die Verfügbarkeit des Netzwerkes oftmals in Prozent an (bei Netzwerken sollte sie deutlich höher als 99 Prozent sein).

Die genannten Metriken erfassen einzelne Eigenschaften von Systemen in bestimmten Systemzuständen. Unterschiedliche Zustände des Systems können die Werte der Metriken verändern. Offensichtlich wird eine Erhöhung der Last auf das System auch deren Auslastung steigern. Weniger offensichtlich ist, dass eine Erhöhung der Last auch einen Einfluss auf die Reaktionszeit und den Durchsatz des Systems hat, da bei zu starker Last einzelne Systemkomponenten vollständig ausgelastet werden. Ein Teil der durchzuführenden Arbeit kann durch die Komponenten nicht mehr erledigt werden. Andere Komponenten wiederum benötigen unter Umständen die Ergebnisse der überlasteten Komponenten, so dass das gesamte System nicht mehr optimal arbeitet. Die Last ist also ein wesentliches Kriterium, um die Analyseergebnisse bewerten zu können. In jedem Fall ist das Analyseergebnis R (gemessen in einer bestimmten Metrik) bestimmt durch eine Systemfunktion f_s, die von einem oder mehreren Lastparametern $L_1 \dots L_n$ abhängt:

$$R = f_s(L_1, L_2, \dots, L_n)$$

Verschiedene analytische Performance-Modelle, die Lasten als Parameter in die Bewertung einbeziehen, werden deshalb im Folgenden vorgestellt.

Vektorpipeline-Performance

Hockney beschreibt in [46] ein Performance-Modell für Vektorrechner, bei dem die Rechenleistung einer Vektorpipeline mittels des Tupels $(r_\infty, n_{1/2})$ angegeben wird. Der Wert r_∞ ist dabei die *asymptotische Vektor-Performance*, die dem Durchsatz der Vektor-Pipeline entspricht, wenn die Länge des Eingabevektors gegen Unendlich strebt. Der Wert $n_{1/2}$ entspricht der Vektorlänge, bei der die Hälfte der asymptotischen Performance r_∞ erreicht wird. Die Ausführungszeit einer Operation mit einem Vektor der Länge n berechnet sich als lineare Funktion mit

$$T(n) = \frac{n + n_{1/2}}{r_\infty} = \frac{1}{r_\infty} \cdot n + \frac{n_{1/2}}{r_\infty}$$

Die Last auf die Vektorpipeline wird in diesem einfachen Modell als Länge des zu berechnenden Vektors definiert. Das von Hockney beschriebene Performance-Modell ist weitgehend auf alle Systeme übertragbar, bei denen eine lineare Abhängigkeit zwischen Datengröße und Ausführungszeit besteht. So kann die Leistung eines Verbindungsnetzwerkes mittels dieses Modells teilweise beschrieben werden, da auch hier die

Übertragungszeit eines Paketes linear mit dessen Größe wächst. Die beiden Parameter r_∞ und $n_{1/2}$ sind in der Aussagekraft äquivalent zur Kapazität des Übertragungskanals und der Grundlatenz der Verbindungsstrecke (der konstante Anteil der Latenz bzw. die Übertragungszeit eines theoretischen Paketes der Länge null). Die Kanalkapazität entspricht r_∞ und die Grundlatenz $\frac{n_{1/2}}{r_\infty}$, wie in der angegebenen linearen Gleichung der Ausführungszeit ersichtlich ist. Die Paketgröße, die in diesem Performance-Modell den einzigen Lastparameter darstellt, kann im realen Netzwerk nicht hinreichend genau die wirkliche Last beschreiben. Diese ist beispielsweise auch von der Anzahl der parallel sendenden Stationen abhängig. Dennoch besitzt das vorgestellte Modell zur Leistungscharakterisierung einen Lastbezug.

Netzwerk-Performance

Mit Einschränkungen kann das Modell auch zur Charakterisierung der Hauptspeicher-Leistung eines Systems verwendet werden. Es kommt in diesem Fall aber aufgrund der hohen Komplexität des Speichersystems schnell an seine Grenzen, da die verschiedenen Cache-Speicherhierarchien nicht mit dem Modell abgebildet werden können. Hier sind komplexere Modelle notwendig. In [47] wird eine mathematische Beschreibung der Transferleistung bei Datenmengen, die nicht mehr in eine Cache-Hierarchieebene passen, präsentiert. Sie sind deshalb in den Übergangsbereichen zwischen verschiedenen Cache-Ebenen angesiedelt. In [48] werden zwei Modelle beschrieben, die die Leistungsverluste beim Zugriff auf Daten in verschiedenen Ebenen der Speicherhierarchie charakterisieren. Das genauere aber sehr komplexe *Memory Hierarchy Model (MH)* kann aufgrund der hohen Komplexität praktisch kaum genutzt werden, weshalb die Autoren in derselben Veröffentlichung das vereinfachte *Uniform Memory Hierarchy Model (UMH)* vorstellen, das die gesamte Speicherhierarchie über 5 Parameter charakterisiert.

Hauptspeicher-Leistung

Die beschriebenen Methoden, die Systemleistung als analytisches Modell in Abhängigkeit von Lasten zu definieren, vertiefen das Verständnis der Abhängigkeit der Leistung von den entsprechenden Lastparametern. Sie können in der Praxis der Leistungsmessung aber nur in vergleichsweise einfachen Systemen mit wenigen Lastparametern verwendet werden. Die Leistung komplexer Systeme ist analytisch schwer beschreibbar. Praktisch wird die Funktion f_S dann häufig als Wertetabelle oder grafische Repräsentation in Form eines Funktionsgraphen dargestellt. Dieser Funktionsgraph ist abhängig von der Anzahl der Lastparameter mindestens zweidimensional.

2.2.2 Lastbeschreibungen

Eine Metrik zur Leistungsbeschreibung ist eine wesentliche Grundlage, um Leistungsbewertungen eines Rechensystems durchzuführen. Um die Leistungsfähigkeit eines Rechensystems anhand einer Leistungsmetrik zu charakterisieren, muss die Leistungsfähigkeit anhand der Erfüllung bestimmter Aufgaben in einem entsprechenden Zeitrahmen ermittelt werden. Diese Menge dieser Aufgaben ist die der Leistungsanalyse zugrunde liegende Last.

Lastbegriff

Die Abhängigkeit der Ergebnisse der Leistungsbewertung von der Last, macht es aber erforderlich, auch diese eindeutig zu beschreiben. Nur so kann eine Bewertung der Systemleistung vergleichbar wiederholt werden. Es ist also eine genaue Definition der Lastsituation wichtig. Einerseits dienen solche Lastbeschreibungen der Dokumentation, so dass die Lasten unter denen Messungen stattfanden, nachvollziehbar dargestellt werden können. Andererseits können sie verwendet werden, um einem Lastgenerations-Tool zu beschreiben, wie die zu generierende Last beschaffen ist. Lastbeschreibungen oder auch Workload-Beschreibungen geben die Anforderungen (oder Requests) von Komponenten an andere Komponenten wieder. Eine Lastbeschreibung für I/O-Last kann beispielsweise alle Anfragen des Prozessors nach Daten der Festplatte innerhalb eines gewissen Zeitraums enthalten.

statische Lastbeschreibung

Lastbeschreibungen werden in der Literatur in *statische* und *dynamische* Beschreibungen klassifiziert [49]. Statische Beschreibungen ermitteln charakteristische Werte einer Last wie beispielsweise Arten von Anforderungen, durchschnittliche Anforderungs-Charakteristika oder Korrelationen zwischen einzelnen Lastparametern, die sich über die Zeit nicht ändern. Klassische Techniken zur Ermittlung von statischen Lastbeschreibungen sind also statistische Analysemethoden, wie die Durchschnittsberechnung oder Korrelation. Da viele Anwendungen in der Praxis allerdings ein Zeitverhalten aufweisen, sind statische Lastbeschreibungen nicht in allen Einsatzszenarien verwendbar. Häufig müssen dynamische Lastbeschreibungen eingesetzt werden, die eine Beschreibung des Verhaltens über die Zeit und damit auch die Voraussage zukünftigen Lastverhaltens ermöglichen. Klassische Techniken zur Beschreibung von dynamischen Lasten sind Markov-Modelle, User-Behavior-Graphs und State-Charts, die jeweils den nächsten Zustand einer Anwendung anhand des aktuellen Zustands und zusätzlicher Parameter beschreiben. Regressionsmethoden, die das Verhalten eines Lastparameters über die Zeit über statistische Analysemethoden ermitteln, sind weitere Techniken zur Ermittlung dynamischer Lastbeschreibungen [49].

dynamische Lastbeschreibung

Im Folgenden wird ein Überblick über wichtige wissenschaftliche Arbeiten gegeben, die sich mit der Beschreibung von Lasten beschäftigt haben.

Seiten-Referenzierungs-String

Als erste Lastbeschreibungen können Beschreibungen des Ressourcenbedarfs einer Applikation über einen gewissen Zeitraum betrachtet werden. Bereits 1966 wird durch Belady in [50] ein Seiten-Referenzierungs-String beschrieben. Er gibt an, welche Speicherseiten durch eine Applikation wann referenziert wurden und kann als dynamische Lastbeschreibung verstanden werden. Da der Seiten-Referenzierungs-String nur die Last beim Zugriff auf den Hauptspeicher auf Seitenebene beschreibt und die Leistung des Hauptspeicherzugriffs wesentlich auch von Caches abhängt, die mit geringeren Größen als ganzen Seiten arbeiten (nämlich mit Cachezeilen von typischerweise ca. 16 Byte Größe), kann das Verfahren heute nicht mehr als einsetzbar zur Charakterisierung der Hauptspeicherlast eingestuft werden.

Angesichts der immer komplexer werdenden Systeme mussten detailliertere Lastbeschreibungen gefunden werden. In [51] wird beschrieben, wie Lasten von Applikationen in den Anwendungsbereichen Batch-Verarbeitung, interaktive Systeme, Datenbanken, Netzwerkanwendungen, Parallel- und Supercomputer ermittelt und dargestellt werden können. Lasten für zahlreiche dieser Anwendungsfelder werden als Folge von Anforderungen verstanden, deren Verteilung durch verschiedene Parameter wie bspw. einem Durchschnittswert für die Frequenz der Anforderungen und einer entsprechenden Varianz angegeben werden. Lasten für Parallelrechnersysteme können als Taskgraphen modelliert werden, in denen Berechnungen als Knoten und Datenabhängigkeiten als Kanten dargestellt werden. Über Kommunikationsgraphen kann auch die Kommunikation zwischen Rechnerknoten abgebildet werden. Das Thema I/O wird in der Arbeit im Kontext der Hochleistungsrechner nicht näher beleuchtet. Spätere Arbeiten der Autoren und die Arbeit [49], die auf die vorher genannte Arbeit aufbaut, betrachten auch das Thema I/O bei parallelen Systemen. So wird die Anzahl der I/O-Anfragen als Workload-Parameter betrachtet. Recht detailliert wird das Thema I/O in parallelen Applikationen in den Arbeiten [52] und [53] behandelt. Die Autoren stellen ein Performance-Modell vor, mit dem Leistungsparameter von Applikationen anhand der Parameter *CPU-Anzahl* und *Disk-Anzahl* ermittelt werden können. Es erfolgt dabei eine Einteilung der Applikation in Phasen, wobei jede Applikation durch 8 Parameter pro Phase bei einer bestimmten Disk- und CPU-Anzahl (jeweils sequentieller und paralleler Anteil von I/O und Berechnungen sowie Skalierungsfaktoren die beschreiben, wie sich einzelne Phasen bei mehr Prozessoren bzw. Disks gegenüber einem Prozessor und einer Disk verlängern) und weiteren 4 Parametern für die Applikation beschrieben wird. Mittels dieser Beschreibung definieren die Autoren ein generalisiertes Amdahlsches Gesetz [54], das auch I/O in die Laufzeitbetrachtungen einbezieht [52]. Aufgrund der hohen Komplexität des Gesamtsystems müssen im Modell Rechenlast und I/O-Last sehr stark vereinfacht werden. Spezifische I/O-Zugriffsmuster können in dem Modell nicht formuliert werden, lediglich welcher Anteil der Applikationslaufzeit durch I/O genutzt wird. Bei Applikationen, die Zugriffsmuster nutzen, die auf verschiedenen Architekturen im Verhältnis zu den Rechenzeiten unterschiedliche I/O-Ausführungszeiten verursachen, erzeugt das Modell entsprechend Ungenauigkeiten in der Vorhersage. Tatsächlich ist dies die Regel und keine Ausnahme. Trotz der stark vereinfachten Modelle werden in der beschriebenen Arbeit noch immer 8 Parameter benötigt, um jede Applikationsphase zu beschreiben. Dies lässt erahnen, welche Komplexität hochgenaue Lastmodelle besitzen müssen, die mehrere Applikationscharakteristiken in die Betrachtung einbeziehen.

Taskgraph

I/O als Workload-Parameter

Phaseneinteilung

Da bislang keine der beschriebenen Vorarbeiten ein hinreichend genaues Lastmodell für I/O präsentierte, wird die vorliegende Arbeit ein Lastmodell vorstellen, das insbesondere I/O verteilter Applikationen genau beschreibt. Der Ansatz der Unterteilung einer Applikation in Phasen, wie er in mehreren der beschriebenen Arbeiten durchgeführt wurde, findet auch in der vorliegenden Arbeit Anwendung.

Die vorherigen Abschnitte über Lastbeschreibungen und Methoden der Leistungscharakterisierung zeigten Möglichkeiten auf, mit denen Last und Leistung in der Vergangenheit beschrieben wurden. Zur Durchführung der eigentlichen Leistungsanalyse gibt es drei verschiedene Methoden, die im Folgenden vorgestellt werden.

2.2.3 Leistungsanalyse-Methoden

Die Leistungsanalyse eines Rechensystems kann grundsätzlich auf zwei verschiedene Arten erfolgen, wie es in Abbildung 2.4 dargestellt ist.

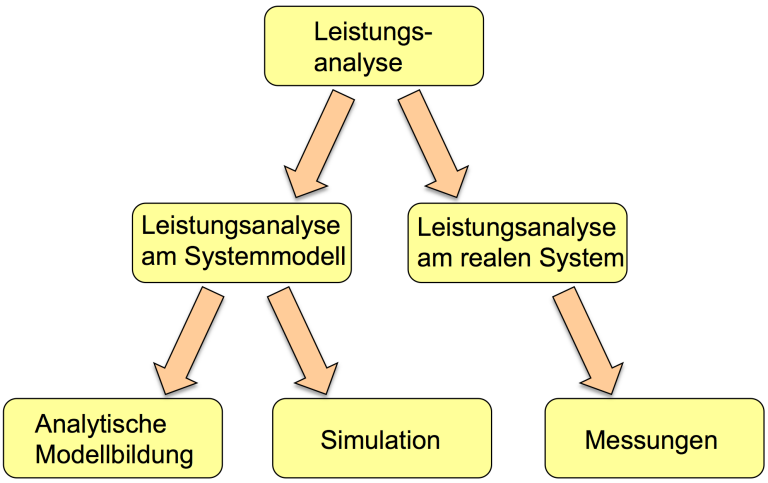

Abbildung 2.4: Methoden der Leistungsanalyse

Modellierung

In der Entwurfsphase eines Systems bzw. immer dann, wenn das System nicht real zur Verfügung steht, werden Leistungsanalysen anhand eines Systemmodells durchgeführt. Die Modellierung des Systems kann einfach gehalten werden, so dass Leistungsvorhersagen mit vergleichsweise einfachen mathematischen Mitteln erfolgen können, sie kann aber auch komplex werden, so dass eine effiziente Leistungsanalyse nur noch mittels Simulationswerkzeugen durchgeführt werden kann.

Messung

Steht ein Rechensystem real zur Verfügung, können Messungen erstellt werden. Es entstehen so die in der Literatur angegebenen drei grundlegenden Methoden der Leistungsanalyse moderner Rechensysteme [55, 1]:

Analytische Modellbildung. Das System wird als mathematisches Modell in Form von Gleichungen beschrieben, an dem Leistungsvorhersagen in Abhängigkeit von Variablen mit Leistungseinfluss getroffen werden können. Die analytische Modellbildung ist für Systeme geringer Komplexität gut einsetzbar und kann bei diesen ein Verständnis der Abhängigkeiten einzelner Leistungsparameter verursachen. Um auch bei komplexen Systemen ein nutzbares und überschaubares analytisches Modell zu erhalten, müssen allerdings unter Umständen so starke

Näherungen durchgeführt werden, dass die Ergebnisse nur noch eine geringe Genauigkeit bieten. In [1] und [55] wird die analytische Modellbildung deshalb als Methode mit geringster Genauigkeit angegeben. Praktisch kann die analytische Modellbildung eingesetzt werden, um schnell zu ersten Näherungen zu gelangen. Diese Leistungsanalyse-Methode bildet auch eine Grundlage der Simulation, da Systemkomponenten im komplexen Simulationsmodell häufig analytisch modelliert werden.

Simulation. Analytische Beschreibungen komplexer Systeme sind nur mit erheblichem Aufwand und zum Teil gar nicht realisierbar. In solchen Fällen ist es sinnvoll, das System mit einem programmierten Algorithmus nachzubilden und in einer rechnergestützten Simulationsumgebung auszuführen. Aufgrund der hohen Genauigkeit der Simulationsmodelle sind Ergebnisse der Simulation oft genauer als Ergebnisse der analytischen Modellbildung [1].

Messungen. Wenn das System real existiert, sind Messungen möglich, die mit dem System selbst durchgeführt werden und deshalb eine sehr hohe Genauigkeit besitzen können.

Messungen sind in vieler Hinsicht die wichtigste Methode der Performance-Analyse, da sie selbst bei analytischer Modellbildung und Simulation notwendig sind, um das entstandene Modell zu kalibrieren und die Korrektheit zu verifizieren [3]. Aus diesen Gründen steht die Messung, das sogenannte Benchmarking, als Methode der Leistungsermittlung im Fokus der vorliegenden Arbeit.

2.3 Benchmarking

Benchmarking

Der Begriff *Benchmarking* existiert in zahlreichen Wissenschaftsbereichen mit unterschiedlichen Bedeutungen. *Benchmarking* entstammt dem Begriff des *Benchmarks*, der ins Deutsche übertragen soviel bedeutet wie „Prüfmarke" oder „Richtgröße". Ursprünglich aus der Geografie stammend, ist *Benchmarking* der Vergleich der aktuellen Höhenposition mit klar erkennbaren Objekten, die als Referenzpunkte dienen, um Höhenlinien zu bilden. Der Begriff des *Benchmarkings* wurde von der Geografie insbesondere in die Wirtschaftswissenschaften übernommen, die darunter den Vergleich der Performance unterschiedlicher Unternehmen hinsichtlich bestimmter Messkriterien verstehen [56]. Mangels existierender Definitionen des Begriffes in den Computerwissenschaften wird eine Definition aus den Wirtschaftswissenschaften von Michael J. Spendolini hier besonders hervorgehoben. Sie beschreibt sehr genau, was *Benchmarking* in Wirtschaftsunternehmen charakterisiert, und kann aufgrund des allgemeinen Ansatzes leicht auch auf die Informatik übertragen werden:

Definition 2.1: Benchmarking nach Spendolini [57]

Benchmarking ist ein fortlaufender systematischer Prozess zur Bewertung von bewährten Produkten, Diensten und Arbeitsprozessen in Organisationen, mit dem Ziel, die gesamte Organisation zu verbessern.

Während viele Definitionen in der Literatur den Vergleich einer Organisation mit anderen in den Vordergrund stellen, ist es ein entscheidender Vorteil der vorgenannten Definition, dass sie die Verbesserung der Arbeitsprozesse innerhalb der Organisation hervorhebt.

Wenn ein Rechensystem mit einer bestimmten Aufgabe als Organisation im Sinne der Definition 2.1 verstanden wird, kann die Definition direkt in die Computerwissenschaft übernommen werden. Auch hier ist *Benchmarking* ein kontinuierlicher systematischer Prozess zur Bewertung der Systemdienste des Rechensystems, die bis zum aktuellen Zeitpunkt als beste Realisierung der Dienste verstanden werden, mit dem primären Ziel, die Abläufe innerhalb des Rechensystems zu verbessern. Der Vergleich mit anderen Rechensystemen, der in den meisten anderen Definitionen im Vordergrund steht, kann dabei ein zentrales Mittel der Verbesserung sein, muss es aber nicht. Auch interne Verbesserungen ohne Vergleich mit anderen Systemen sind Zielstellung des *Benchmarkings*.

2.3.1 Benchmarks

Benchmark

Während der Begriff *Benchmarking* recht einheitlich in den unterschiedlichen Wissenschaftsbereichen behandelt wird, besteht Uneinigkeit bei der Deutung des Begriffes *Benchmark*. Während der *Benchmark* in der Geografie genau wie in den Wirtschaftswissenschaften einem Referenzwert entspricht, an dem Vergleichsmessungen durchgeführt werden, stellt er in der Informatik das Programm dar, das die Messung unter Einfluss einer generierten Last durchführt. Benchmarks in der Informatik bestehen also aus Rechen- und Programmdaten, die nach bestimmten Vorschriften die Last für Rechner erzeugen, um Messungen zur Bewertung der Leistungsfähigkeit durchzuführen. Im Folgenden wird der Begriff *Benchmark* ausschließlich mit dieser Bedeutung verwendet. Messwerte, die durch einen Benchmark ermittelt wurden, sind in dieser Arbeit zur besseren Unterscheidung als Benchmark-Ergebnisse bezeichnet.

Computer-Benchmarks messen Leistungswerte auf unterschiedlichen Abstraktionsebenen der erzeugten Last. So werden Benchmarks in der Literatur häufig in die Klassen Low-Level-Benchmarks und Applikationsbenchmarks eingeteilt [58, 46].

Low-Level-Benchmark

Low-Level-Benchmarks dienen der Ermittlung der Leistungsfähigkeit eines Rechensystem bzgl. spezifischer Eigenschaften, wie bspw. die maximale Speicherbandbreite [59]. Häufig ist es Ziel von Low-Level-Benchmarks, die maximale Leistungsfähigkeit des Systems bzgl. der zu vermessenden Eigenschaft zu ermitteln und die Last, die

durch Applikationen und Betriebssystem entsteht, zu minimieren. Derartige Benchmarks ermitteln nicht die Gesamtleistung einer Applikation. Dazu sind Applikations-Benchmarks besser geeignet. Diese Benchmarks führen die Applikation selbst aus und vermessen beispielsweise die Applikationslaufzeit. Applikations-Benchmarks geben die Leistung der entsprechenden Applikation oder Applikationsklasse recht genau wieder, sind aber schlecht auf andere Anwendungsszenarien übertragbar.

Applikations-Benchmark

Oft werden neben den beiden genannten Klassen noch die synthetischen Benchmarks und die Kernelbenchmarks als weitere Benchmarktypen genannt. Die Bezeichnungen dieser Benchmarktypen beziehen sich weniger auf den Abstraktionsgrad der Last, sondern vielmehr auf die Grundlage der Lasterzeugung. Synthetische Benchmarks verwenden synthetische Lasten, die keine reale Applikation als Grundlage haben. Vielmehr werden die Lasten von den Anforderungen des Benchmark-Entwicklers definiert. Oft bilden Statistiken typischer Lasten bei der Lastdefinition des synthetischen Benchmarks den Ausgangspunkt. Diese Benchmarks erzeugen so Benchmarkergebnisse, die oftmals wenig aussagekräftig sind und deren Übertragbarkeit auf reale Applikationen als problematisch gelten kann.

synthetischer Benchmark

Kernelbenchmarks hingegen verwenden den Kern einer Applikation, um so Leistungscharakteristika der an der Ausführung dieses Kerns beteiligten Rechnerkomponenten unter der Applikationslast herauszufinden. Andere Applikationskomponenten (wie zum Beispiel die grafische Ausgabe), die ebenfalls Last erzeugen, werden ignoriert. Kernelbenchmarks sind deshalb in der Komplexität geringer als Applikations-Benchmarks, ermöglichen aber bezüglich der Applikation, aus der ihr Kern stammt, genauere Leistungsanalysen als Low-Level-Benchmarks. Mittels der Kernelbenchmarks können leichter Schlussfolgerungen bzgl. der durch den Kern verwendeten Komponenten gefunden werden. Allerdings besteht auch bei Kernelbenchmarks das Problem, dass ihre Ergebnisse auf andere Anwendungsszenarien nur schlecht übertragbar sind.

Kernelbenchmark

Benchmarking hat das Ziel, durch Bewertung von Rechnerkomponenten zur Optimierung der Rechentechnik beizutragen. Der Vergleich verschiedener Rechensysteme durch jeweils gleiche Benchmarks kann Möglichkeiten der Verbesserung aufzeigen, ist aber dennoch nicht ohne Schwierigkeiten möglich. Beispielsweise kann es passieren, dass ein Benchmark nicht auf allen Plattformen lauffähig ist, die miteinander verglichen werden. In [59] werden deshalb Anforderungen definiert, die ein Benchmark besitzen muss, um nutzbringend eingesetzt werden zu können:

Portabilität. Um den Benchmark auf unterschiedlichen Plattformen ausführen zu können, muss er leicht portabel sein. Da reale Applikationen häufig aber spezifisch auf Rechensysteme angepasst werden, um eine möglichst hohe Leistung zu erreichen, wird bereits an dieser Stelle deutlich, dass Portabilität der Entwicklung von realitätsnahen Benchmarks entgegenwirken kann.

Repräsentativität. Benchmarks sollen für eine möglichst große Klasse an Applikationen repräsentative Ergebnisse ermitteln.

Skalierbarkeit. Benchmarks müssen auch für zukünftige Rechnergenerationen relevante Ergebnisse liefern, also mit der sich steigernden Systemleistung skalieren.

Systemverständnis. Benchmarks sollen dem Nutzer ein Verständnis für die Vorgänge im System verschaffen. Der Nutzer soll auf diese Weise befähigt werden, Quellen für Leistungsunterschiede zu entdecken und so Optimierungen durchführen zu können.

Benchmarks der beschriebenen Benchmark-Klassen können bei deren sorgfältiger Implementierung in der Regel drei der vier vorgenannten Eigenschaften erfüllen. *Portabilität* wird durch eine Programmierung erreicht, die systemabhängige Anteile so weit wie möglich vermeidet und kapselt, während *Skalierbarkeit* durch eine vom Nutzer anpassbare Problemgröße gewährleistet wird. Benchmarks tragen in der Regel zur Erhöhung des *Systemverständnisses* bei, wenn einzelne Parameter (wie die Problemgröße) variiert werden können, um somit die Abhängigkeit der Leistung von diesen Lastparametern herauszufinden.

Lediglich die *Repräsentativität* der Benchmark-Ergebnisse kann bei den beschriebenen Benchmarkklassen nicht immer gewährleistet werden. Low-Level-Benchmarks vermessen eine spezifische Eigenschaft. Da es sich dabei um ein bestimmtes Kriterium handelt, das die zu analysierende Komponente möglichst ohne Last charakterisieren soll, ist eine Leistungsvorhersage bei bestimmten Lasten anhand der Leistungscharakterisierung häufig nicht oder nicht genau genug möglich. Beispielsweise ermitteln einige Netzwerk-Benchmarks die asymptotische Performance eines Netzwerkes nach dem in Absatz 2.2.1 beschriebenem Modell. Bei einem Netzwerk entspricht dies der maximal möglichen Bandbreite bei einem unendlich großen Paket. Praktisch ist dieser Wert quasi aussagefrei, da kein Netzwerk eine derartige Bandbreite praktisch erreichen kann. Er kann lediglich als Grundlage für die Leistungsvorhersage bei unterschiedlichen Paketgrößen verwendet werden. Kernelbenchmarks und Applikationsbenchmarks hingegen ermitteln realistische Ergebnisse, da sie originale Lasten von Applikationen verwenden und mittels dieser Leistungsmessungen durchführen. Allerdings entsprechen die Messergebnisse nur genau der Applikation, deren Last während des Messvorgangs verwendet wurde. Sie sind nicht auf andere Applikationen übertragbar.

Diese Schwäche der bekannten Benchmark-Klassen lässt sich nicht mit Verbesserungen von Benchmarks der entsprechenden Klassen beheben, da die Probleme konzeptionell bedingt sind. Sie lassen sich nur beheben, in dem eine neue Benchmark-Klasse eingeführt wird, die von Beginn an die *Repräsentativität* von Ergebnissen als wichtige Benchmark-Eigenschaft in die Betrachtungen einbezieht [60].

applikationsbasierter Benchmark

Im Folgenden wird diese neue Benchmark-Klasse der *applikationsbasierten Benchmarks* mit ihren Eigenschaften vorgestellt. Bei applikationsbasierten Benchmarks handelt es sich um Applikationen, die eine spezielle Form von Workload (beispielsweise Sekundärspeicherlast oder Rechenlast) in jeder beliebigen Ausprägung als Grundlage der Leistungsvermessung verwenden können. Dazu ist es erforderlich, dem Benchmark eine eindeutige Beschreibung der Last zur Verfügung zu stellen, die dann als

Grundlage der Vermessung dient. Beispielsweise kann ein applikationsbasierter I/O-Benchmark das I/O-Verhalten einer Applikation nachbilden, indem eine vorher ermittelte I/O-Last der Applikation als Lastbeschreibung des Benchmarks verwendet wird. Im Fall der Nachbildung des Lastverhaltens einer realen Applikation entspricht ein applikationsbasierter Benchmark also einem Kernelbenchmark, da die spezifische Last einer Applikation in Hinblick auf eine bestimmte Komponente (bspw. Sekundärspeicher) originalgetreu nachgebildet wird. Im Gegensatz zum Kernel-Benchmark hat der applikationsbasierte Benchmark aber den wesentlichen Vorteil, dass beliebige Applikationslasten des entsprechenden Typs nachgebildet werden können, selbst synthetische Lasten, wie sie von synthetischen Benchmarks oder Low-Level-Benchmarks erzeugt werden.

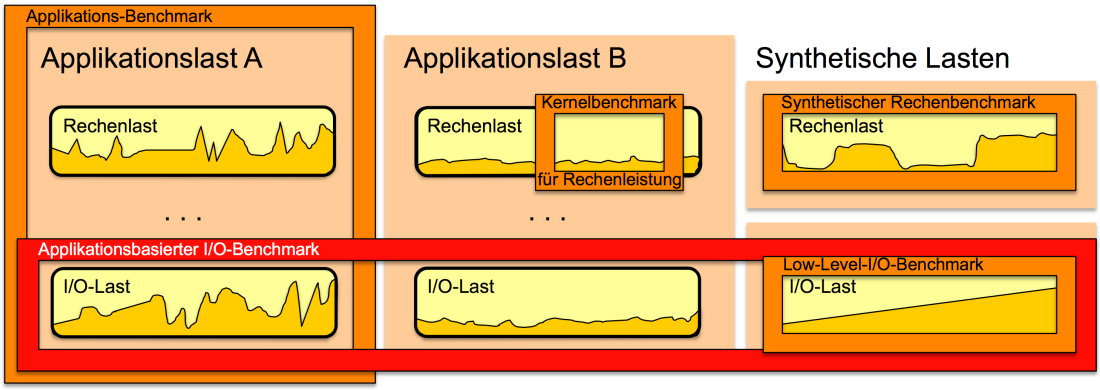

Abbildung 2.5: Arten von Benchmarks

Abbildung 2.5 stellt die Unterschiede der Benchmark-Arten grafisch anhand eines Beispiels dar. Während der Applikations-Benchmark die komplette Applikationslast der Applikation A (also sowohl Rechen- als auch I/O-Last einer Applikation) als Messgrundlage verwendet, analysiert der Kernelbenchmark nur einen Teil der Last einer bestimmten Lastart der Applikation B (die Last, die der zu untersuchende Kernel erzeugt). Der Low-Level-Benchmark vermisst die Leistung bei einer ganz bestimmten Lastsituation, in der alle nicht zu vermessenden Lasten minimiert werden, um die Leistung einer Komponente möglichst lastunabhängig ermitteln zu können. Er verwendet dabei eine synthetische Last, wie es auch der synthetische Benchmark tut. Dieser kann jedoch mehr als nur eine Komponente des Rechensystems vermessen und erzeugt deshalb unter Umständen auch mehrere Lastarten (wenn auch nicht in diesem Beispiel). Lediglich der applikationsbasierte Benchmark kann applikationsübergreifend eingesetzt werden, da er eine Lastart in jeder möglichen Ausprägung nachbilden kann und als Messgrundlage verwendet. Deutlich wird an der Grafik auch, dass der applikationsbasierte Benchmark sogar Lasten anderer (auch synthetischer) Benchmarks, nachbilden kann, sofern der Benchmark die Lastart verwendet, die der applikationsbasierte Benchmark unterstützt.

2.3.2 Der ideale I/O-Benchmark

I/O-Benchmark

Ziel dieser Arbeit ist die Entwicklung eines Benchmarks für die Leistungsanalyse von Sekundärspeichersystemen, der im Folgenden I/O-Benchmark genannt wird. Der erste Schritt wird die Definition von Eigenschaften sein, die ein idealer I/O-Benchmark erfüllen muss.

In Abschnitt 2.3.1 wurden vier Anforderungen an Benchmarks beschrieben, die sie erfüllen müssen, um nutzbringend eingesetzt werden zu können [59]. Diese Anforderungen müssen für I/O-Benchmarks in gleichem Maße gelten. Allerdings kommen bei diesen weitere Anforderungen hinzu. In [61] werden 6 Eigenschaften definiert, die einen idealen I/O-Benchmark kennzeichnen. Vier der sechs Eigenschaften entsprechen den in Abschnitt 2.3.1 genannten Anforderungen an einen Benchmark, zwei weitere Eigenschaften kommen für I/O-Benchmarks hinzu:

idealer I/O-Benchmark

Vergleichbarkeit. Ein guter Benchmark erlaubt den Vergleich verschiedener Rechensysteme. Dies inkludiert die oben genannte Eigenschaft „Portabilität", da ein Vergleich unterschiedlicher Plattformen nur ermöglicht werden kann, wenn die Applikation portabel ist. Vergleichbarkeit bedeutet aber auch, dass die Messvoraussetzungen (bspw. die vom Benchmark erzeugte Last) auf den unterschiedlichen Rechensystemen vergleichbar ist.

Repräsentativität. s. Abschnitt 2.3.1

Skalierbarkeit. s. Abschnitt 2.3.1

Systemverständnis. s. Abschnitt 2.3.1

Reproduzierbarkeit. Weiterhin müssen Ergebnisse reproduzierbar sein, d. h. es muss klar definiert werden, unter welchen Bedingungen der Benchmark ausgeführt wird, um gültige Ergebnisse zu erzielen. Der genannte Punkt ist für alle Benchmarks von Bedeutung und nicht nur für I/O-Benchmarks.

I/O-Bezug. Ein I/O-Benchmark sollte ausschließlich I/O-Last erzeugen und als Messgrundlage verwenden. Die Betrachtung anderer Last-Arten (bspw. Rechenlast) erhöht die Komplexität des Benchmarks und erschwert die Analyse der Messergebnis (also das Systemverständnis).

In der Arbeit [62] wird ein weiteres Problem von I/O-Benchmarks erläutert. Sekundärspeicherzugriffe hängen von einer Vielzahl von Parametern ab, so dass eine Vermessung aller Kombinationen von Parameterwerten zu einer Vielzahl von notwendigen Messungen führt. Eine derartige komplette Vermessung aller Wertekombinationen liefert ein genaues Abbild des I/O-Verhaltens des Systems, würde aber zu lange dauern. Es wird anhand eines Beispiels gezeigt, dass die komplette Vermessung der Wertekombinationen einer Sprite DECstation, die in der Arbeit als Beispielrechnersystem herangezogen wird, fast 2 Monate dauern kann. Dies ist nicht akzeptabel, weshalb in der Praxis Einschränkungen bei der Vemessung des Parameterraums definiert werden

müssen. Dies muss durch den Nutzer geschehen, kann aber angesichts der Vielzahl an Parametern sehr schwierig werden. Der Nutzer hat zu entscheiden, welche Parameter auf welche Parameterwerte eingeschränkt werden und kann bei falscher Definition der Einschränkungen signifikante Abweichungen von relevanten Messergebnissen erhalten. Eine Nutzerunterstützung durch Werkzeuge, die diese Komplexität verringern kann, muss jeder ideale Benchmark zur Verfügung stellen. Dies betrifft nicht nur die Konfiguration des Benchmarks, sondern auch die Ergebnisauswertung. Durch die komplette oder teilweise Vermessung des Parameterraums entsteht eine Vielzahl von Ergebnissen, die ohne technische Unterstützung nur mit viel Aufwand zu analysieren ist. Aus diesen Überlegungen ergeben sich zwei weitere Eigenschaften eines idealen Benchmarks, die nach Kenntnis des Autors bislang in keiner Arbeit erwähnt wurden.

Laufzeit. Der Benchmark muss innerhalb eines für Nutzer akzeptablen Zeitrahmens terminieren, d. h. eine Messung, die mehrere Wochen benötigt, ist im Allgemeinen nicht tolerierbar. Ein Zeitrahmen von einigen Stunden ist akzeptabel und häufig auch notwendig, um valide Ergebnisse zu erhalten.

Nutzerunterstützung. Ein idealer I/O-Benchmark ermöglicht eine leichte Konfiguration und eine leicht verständliche Ergebnisdarstellung für den Nutzer.

Die eingeführten Eigenschaften eines idealen I/O-Benchmarks werden nun verwendet, um vorhandene I/O-Benchmarks gegenüberzustellen.

2.3.3 Vorhandene I/O-Benchmarks

Wie bereits erwähnt gibt es eine Reihe von I/O-Benchmarks, die in verschiedenen Einsatzszenarien verwendet werden. So gibt es I/O-Benchmarks für Workstation-Systeme, die typischerweise die POSIX-I/O-Schnittstelle verwenden. Neben diesen POSIX-I/O-Benchmarks existieren Benchmarks für parallele Systeme, die im Allgemeinen die MPI-IO-Schnittstelle zum Zugriff auf den Massenspeicher verwenden. In diesem Abschnitt werden zuerst bekannte POSIX-I/O-Benchmarks vorgestellt, bevor ein umfassender Überblick über vorhandene MPI-IO-Benchmarks gegeben wird. Es wird besonderer Wert auf die Darstellung der parallelen I/O-Benchmarks gelegt, da die vorliegende Arbeit ihren Fokus auf parallele I/O-Benchmarks legt. Die im Folgenden vorgestellten Benchmarks sind in ihren Kategorien nach Namen alphabetisch sortiert. Die Reihenfolge macht keine Aussage über ihre Verbreitung oder Wichtigkeit.

POSIX-I/O-Benchmarks

Zuerst werden die beiden POSIX-I/O-Benchmarks *Bonnie* und *IOZone* vorgestellt.

Bonnie. Sowohl der *Bonnie*-Benchmark als auch sein Nachfolger, der *Bonnie++* gehörten lange zu den bekanntesten POSIX-I/O-Benchmarks. *Bonnie* besteht aus

4 Prozessen, die parallel einen zufällig ermittelten Block einer zufällig generierten Datei lesen und in 10 Prozent aller Fälle zurückschreiben. Außerdem wird die Leistung bei Verwendung sequentieller Zugriffe, d. h. von aufeinander folgenden Zugriffen ermittelt. Dieses sehr spezielle Zugriffsmuster ist nicht für alle möglichen Applikationen kennzeichnend, weshalb die Ergebnisse des *Bonnie* im Allgemeinen nicht vergleichbar sind. Das genannte Problem existiert in dieser Form bei nahezu allen POSIX-I/O-Benchmarks. Eine Ausnahme bildet der *IOzone*, der im Folgenden kurz dargestellt wird [63].

IOzone. Der *IOzone* ist einer der weithin gebräuchlichsten I/O-Benchmarks im Linux/UNIX-Umfeld. Er besitzt als einer der wenigen existierenden I/O-Benchmarks die Möglichkeit komplexe Workloads basierend auf Applikations-I/O-Verhalten zu definieren. Dazu sind zwei Profildateien zu definieren, die jeweils getrennt die Schreib- und die Lesezugriffe des nachzubildenden I/O-Verhaltens beschreiben. Ein Vermischen der Lese- und Schreiboperationen ist nicht möglich. *IOzone* bietet keine Unterstützung des Benutzers in der Erstellung dieser komplexen Workloads [64].

Die Vorstellung der beiden dargestellten POSIX-I/O-Benchmarks, die zu den gebräuchlichsten I/O-Benchmarks für UNIX-Systeme zählen, soll stellvertretend für die große Menge an existierenden POSIX-I/O-Benchmarks ausreichen, da sich diese Arbeit auf I/O-Benchmarks für parallele I/O-Systeme konzentriert. Es kann an dieser Stelle gesagt werden, dass die meisten existierenden POSIX-I/O-Benchmarks ähnlich wie der *Bonnie*-Benchmark nur wenige Einstellungsmöglichkeiten für den verwendeten Workload besitzen. Benchmarks wie der *IOzone*, die eine flexible Konfiguration ermöglichen, sind die Ausnahme.

MPI-IO-Benchmarks

Im Folgenden wird ein Überblick über wichtige MPI-IO-Benchmarks gegeben, um zu analysieren, ob in diesem Bereich der Benchmarks für Hochleistungsrechensysteme ähnliche Probleme existieren, wie bei POSIX-I/O-Benchmarks.

b_eff_io. Der *b_eff_io* entstand als I/O-Gegenstück zum *b_eff*-Benchmark, der die effektive Kommunikationsbandbreite in einem parallelen Rechnersystem ermittelt. Somit eignet sich der *b_eff_io* zur Vermessung der effektiven Bandbreite beim Zugriff auf den Sekundärspeicher des Systems. Der Benchmark verwendet dabei zahlreiche Zugriffsmuster der MPI-IO-Schnittstelle. Diese Zugriffsmuster sollen das Verhalten verschiedener Applikationen charakterisieren und dem Nutzer auf diese Weise einen guten Querschnitt über die Leistungsfähigkeit eines Rechensystems präsentieren. *b_eff_io* kann Ergebnisse auch als Grafikdateien zur Verfügung stellen, um so bei der Auswertung zu unterstützen [65].

FLASH I/O Benchmark. Zur Simulation astrophysikalischer Phänomene, wie Supernovae, wird der FLASH-Code verwendet, von dem der *FLASH I/O* Benchmark abgeleitet wurde. Der FLASH-Code erzeugt sehr große Datenmengen von etwa 0.5 TByte, so dass alte Versionen der Applikation teilweise 50 % der Zeit des Applikationslaufes nur mit Sekundärspeicherzugriffen beschäftigt waren. Aus diesem Grund wurde die I/O-Funktionalität aus dem FLASH-Code in diesen I/O-Benchmark ausgelagert. Da der *FLASH I/O* Benchmark im Wesentlichen Checkpoint-Daten zur Wiederaufnahme des Applikationslaufes nach einem Fehlerfall und Logdaten zur Visualisierung schreibt, vermisst er die I/O-Leistung nur eines spezifischen I/O-Verhaltens. Die entstehenden Ergebnisse sind damit für die FLASH-Applikation relevant, aber nicht allgemein gültig [66, 67].

HPIO. Der *HPIO* (High Performance I/O), in [68] noch als *NCIO* (Noncontiguous I/O) bezeichnet, ist ein leistungsfähiger I/O-Benchmark, dessen besonderer Fokus auf noncontiguous I/O liegt, d. h. auf I/O-Transfers mit nicht zusammenhängenden Quell- und Zielbereichen. Diese besondere Form des I/O wird in Abschnitt 3.3.4 genauer vorgestellt und ist eine Merkmal des I/O-Verhaltens von Hochleistungsrechen-Applikationen. Der *HPIO* verwendet 4 verschiedene Zugriffsmethoden für noncontiguous I/O bei unterschiedlichen Zugriffsmustern [69]. Der *HPIO* adressiert das Problem der Komplexität von I/O-Zugriffen aufgrund der zahlreichen möglichen einstellbaren Parameter. Allerdings konzentriert sich der Entwickler auf die Realisierung beliebiger Muster von noncontiguous I/O, für die er drei Parameter definiert. Damit können zwar die Zugriffsmuster vieler wissenschaftlicher Applikationen abgebildet werden, es fehlen aber Möglichkeiten der Definition besonderer Zugriffseigenschaften, wie bspw. der asynchrone Zugriff. Die definierten Parameter bilden die Möglichkeiten von I/O-Zugriffen nicht vollständig ab [68].

IOR. *IOR* wurde vom Lawrence Livermore National Laboratory entwickelt. Er verwendet mehrere I/O-Schnittstellen und zahlreiche I/O-Zugriffsmuster. Seit der Version 2, die eine komplette Neuentwicklung des Benchmarks darstellt, ist er sehr gut konfigurierbar. Zahlreiche I/O-Parameter werden unterstützt. Ebenso die Erstellung von I/O-Lastmustern anhand einer Konfigurationsdatei. Allerdings erlaubt der *IOR* auch in der aktuellen Version keine I/O-Zugriffe ohne Blockierung, sowie eine genaue Spezifikation der Zusammensetzung von Lese- und Schreibzugriffen.

LANL MPI-IO Test. Dieser vom Los Alamos National Laboratory (LANL) veröffentlichte I/O-Benchmark wurde für die besonderen Anforderungen am LANL entwickelt. So unterstützt der Benchmark fünf verschiedene Zugriffsmuster für eine große Anzahl an Prozessen, die mittels MPI kommunizieren. Der *LANL MPI-IO Test* ist vielseitig konfigurierbar, wobei die eigentlichen Zugriffsmuster dennoch festgelegt sind [70].

mpi-tile-io. Dieser vom Parallel I/O Benchmark Consortium veröffentlichte Benchmark testet die I/O-Leistung beim Zugriff auf eine zweidimensionale Matrix, die auf Rechnerknoten aufgeteilt ist. Bestimmte Teile der Daten werden zwischen angrenzenden Knoten geteilt, so dass eine Synchronisation beim Zugriff erfolgen muss. Derartige Szenarien treten in verschiedenen Simulationsumgebungen auf, bei denen Rechner einzelne „Regionen" simulieren, es aber zwischen den Regionen Wechselbeziehungen gibt. Der Benchmark kann zwar mit einigen Parametern konfiguriert werden, um beispielsweise die Problemgröße in der x- und y-Dimension zu verändern – am eigentlichen Zugriffsmuster einer zweidimensionalen Matrix kann der Anwender jedoch nichts ändern [71].

NPBIO-2.4-MPI. *NPBIO* als der I/O-Benchmark der NAS Parallel Benchmarks (daher die Abkürzung) berechnet ein lineares Gleichungssystem, dessen Koeffizientenmatrix tridiagonal ist (sie enthält viele Nullen, während die besetzten Felder ein spezielles diagonales Muster darstellen). Die Berechnung derartiger Gleichungssysteme wird auch Block-Tridiagonal-Problem (BT-Problem) genannt [72] und ist der Grund dafür, dass der Benchmark früher auch NAS-BTIO hieß. Die berechnete Ergebnismatrix wird vom Benchmark nach jeweils fünf Berechnungsschritten auf den Massenspeicher geschrieben. Am Ende der Algorithmusausführung müssen die Elemente der Matrix nach der Größe ihrer Vektorkomponenten geordnet werden. Es wird dabei die *effektive Ausgabebandbreite* ermittelt, die der geschriebenen Datenmenge dividiert durch die insgesamt benötigte Zeit zur Berechnung der Ergebnismatrix entspricht. Aufgrund dieser Vermischung von Berechnung und I/O sind die Messergebnisse des *NPBIO-2.4-MPI* ausschließlich spezifisch für das berechnete Problem, dessen I/O-Charakteristik typischen CFD-Applikationen (Computational Fluid Dynamics) zur Strömungsberechnung von Flüssigkeiten oder Gasen entspricht. Aufgrund dieser Vorgehensweise besteht beim *NPBIO* eine Abhängigkeit der Massenspeicher-Leistung von der Rechenleistung der CPU. Außerdem wird lediglich ein Workload betrachtet, nämlich das Schreiben von Ergebnismatrizen [73].

Parallel Input/Output Test Suite. Die *Parallel Input/Output Test Suite* wurde an der Universität von Southhampton entwickelt. Sie entstand im Umfeld der *Parkbench*-Benchmarks gehört aber nicht zur *Parkbench*-Testsuite (da auch der BT-Benchmark zur Parkbench-Suite gehört, enthält diese außer dem *BTIO* keinen I/O-Benchmark). Die *Parallel Input/Output Test Suite* führt eine Reihe von Tests durch, die verschiedene Zugriffsmuster testen. Einerseits werden mittels verschiedener Low-Level-Benchmarks einfache Zugriffsmuster (wie das sequentielle Lesen und Schreiben durch einen oder mehrere Prozesse in synchronem und asynchronem Modus) vermessen. Andererseits werden Zugriffsmuster unterschiedlicher Applikationen verwendet, wie sie häufig in wissenschaftlichen Anwendungen vorkommen. Die fünf untersuchten Applikationszugriffsmuster sind:

- I/O von regulären Feldern, wie sie bspw. in CFD-Applikationen verwendet werden
- nicht-sequentielles I/O, wie es beispielsweise typisch für Datenbanken ist
- Gather/Scatter-Operationen, die mit I/O verbunden sind
- I/O mit gemeinsamen Dateizeigern (shared filepointer), das häufig beim Schreiben von Logdateien verwendet wird
- Transpose-Operationen wie sie bei Rechenoperationen mit großen Matrizen häufig vorkommen

Die *Parallel Input/Output Test Suite* deckt damit ein breites Spektrum an Zugriffsmustern ab. Dennoch können Lasten spezieller Applikationen nicht in jedem Fall von den unterstützten Mustern ausreichend genau angenähert werden, da eine freie Konfiguration der Zugriffsmuster nicht vorgesehen wird. Es ist lediglich eine Anpassung in wenigen Parametern möglich [74].

PIO-Bench. In einer Masterarbeit [75] am Parallel Architecture Research Laboratory der Clemson University entstand ein I/O-Benchmark für parallele Systeme, der zahlreiche bis dahin existierende I/O-Benchmarks integrieren sollte. Der *PIO-Bench* vereinigt mehrere relevante Zugriffsmuster (unter anderem auch das des FLASH-I/O) zu einem I/O-Benchmark, der überdies noch ein Framework zur hochpräzisen Zeitmessung enthält. Der Fokus der Entwicklung lag auf der Entwicklung eines Frameworks zur Integration zahlreicher Benchmark-Workloads, weshalb das Problem eines allgemein verwendbaren I/O-Workloads auch in dieser Arbeit nicht gelöst wird. Der Benchmark erzeugt zahlreiche Ergebnisse, die schwer auszuwerten sind. Es ist deshalb als zukünftige Entwicklung ein Werkzeug zur Unterstützung bei der Auswertung geplant [76].

2.4 Zielstellung

Nachdem vorhandene Benchmark-Systeme zur Leistungsanalyse von I/O-Systemen vorgestellt wurden, werden nun die zentralen Probleme aktueller Systeme aufgezeigt. Diese Probleme dienen als Ausgangspunkt zur Formulierung der Kernfragen, die im Rahmen dieser Arbeit beantwortet werden.

2.4.1 Gegenüberstellung vorhandener Benchmarks

Dieser Abschnitt stellt die bisher in der Arbeit kurz vorgestellten I/O-Benchmarks tabellarisch gegenüber um einen einfachen Vergleich zu ermöglichen. Dabei werden für jeden Benchmark die definierten acht Merkmale eines idealen I/O-Benchmarks bewertet. Bewertungen können positiv ($+$), neutral ($\bigcirc$) oder negativ ($-$) sein, wobei eine

Bewertungsmaßstab

positive Bewertung einem Punkt, eine neutrale Bewertung keinem Punkt und eine negative Bewertung einem Minuspunkt entspricht. Die Bewertungen der einzelnen Kategorien erfolgen nach folgenden Kriterien, um eine Objektivität zu gewährleisten:

Vergleichbarkeit. Vergleichbarkeit ist bei den meisten Benchmarks gegeben. Entweder verwenden Benchmarks feste Workloads für eine einfache Vergleichbarkeit oder sie sehen Referenzworkloads vor. Ist keine der beiden Möglichkeiten vorgesehen, wird eine negative Bewertung vergeben, sonst eine positive. Ebenso gibt es eine negative Bewertung, wenn der Benchmark nur für wenige spezifische Architekturen implementiert wurde.

Repräsentativität. Benchmarks, die nur einen Workload unterstützen, erhalten in diesem Punkt eine negative Bewertung. Werden mehrere Lastverhalten unterstützt, wird hier eine neutrale Bewertung vergeben. Nur Benchmarks, die eine freie Definition von I/O-Lasten in ihrem Fokus verarbeiten können, erhalten eine positive Bewertung.

Skalierbarkeit. Ist eine automatische oder einfache manuelle Skalierung der Lasten des Benchmarks an neue Systeme möglich, erhält ein Benchmark eine positive Bewertung in der Kategorie „Skalierung". Ist eine Skalierung nur durch Änderung und Neuübersetzung der Quellen möglich, wird eine neutrale Bewertung gegeben. Hingegen gibt es eine negative Bewertung, wenn keine Skalierung ermöglicht wird.

Systemverständnis. Ein Benchmark hilft dann beim Systemverständnis, wenn einzelne Parameter variiert werden können, um deren Einfluss auf das System zu erkennen. Wenn dies einfach möglich ist oder automatisch vom Benchmark gemacht wird, gibt es eine positive Bewertung. Kann der Nutzer Parameter variieren, die aber einen neuen Benchmarklauf benötigen, ist die Bewertung neutral. Negative Bewertungen gibt es, wenn eine derartige Variation gar nicht möglich ist.

Reproduzierbarkeit. Reproduzierbarkeit kann prinzipiell bei jedem Benchmark realisiert werden. Wichtig dafür ist eine gute Dokumentation des Entwicklers, in der genaue Hinweise zu erlaubten Optimierungen, zur Kompilation und Ausführungsumgebung gegeben werden. Viele Benchmark-Entwickler erlauben beliebige Compiler-Optimierungen. In solchen Fällen kann der Benchmark leicht zu einem Compiler-Test werden. Dennoch ist dies eine klare Aussage zur Kompilation und wird an dieser Stelle positiv bewertet. Fehlen Aussagen zur Kompilation und Ausführungsumgebung wird an dieser Stelle eine neutrale Bewertung gegeben.

I/O-Bezug. Ein Benchmark erhält die positive Bewertung $+$, wenn ausschließlich I/O vermessen wurde, sonst gibt es eine neutrale Bewertung.

Laufzeit. Diese Eigenschaft erfüllt jeder verbreitete Benchmark, da er von Nutzern nicht eingesetzt werden kann, wenn seine Laufzeit eine nicht tolerierbare Grenze

überschreitet. Deshalb bekommen alle Benchmarks in dieser Kategorie eine positive Bewertung. Dennoch ist das Laufzeit-Kriterium als Anforderung an einen idealen Benchmark wichtig, denn sonst wäre die Entwicklung eines Benchmarks möglich, der durch Vermessung aller möglichen I/O-Parameterkombinationen ein genaues und komplettes Leistungsabbild eines I/O-Systems schafft.

Nutzerunterstützung. Eine positive Bewertung erfolgt hier, wenn der Benchmark den Nutzer in Konfiguration und Ergebnisanalyse unterstützt. Dies kann dadurch geschehen, dass der Benutzer entweder nur wenige Möglichkeiten zur Konfiguration hat oder der Benchmark nur wenige leicht verständliche Ausgaben produziert. Wenn er komplexe Konfiguration ermöglicht bzw. viele Ausgaben produziert, muss er aber auch Möglichkeiten anbieten, komplexe Konfigurationen einfach zu erstellen und komplexe Ausgaben zu analysieren.

Tabelle 2.1 zeigt das Ergebnis dieses Vergleiches. Die Bewertungspunkte der letzten Spalte in Tabelle 2.1 geben einen Hinweis auf Benchmarks, die in besonders vielen Kategorien positive Ergebnisse erzielt haben. Zusätzlich wurden alle Felder mit negativer oder neutraler Bewertung in der Tabelle grau hinterlegt, um schneller wahrnehmen zu können, wo die zentralen Schwachstellen der verglichenen Benchmarks liegen. Demnach sind *IOzone*, *b_eff_io*, *HPIO* und *IOR* mit jeweils sieben Punkten empfehlenswerte Benchmarks. Aber auch sie haben Schwächen. Drei dieser vier Benchmarks besitzen eine positive Bewertung im Punkt Repräsentativität, dafür aber eine neutrale Bewertung im Punkt Nutzerunterstützung, während dies beim vierten Benchmark umgekehrt ist. Die Repräsentativität und Nutzerunterstützung sind offenbar Probleme aktueller Benchmarks, wobei die Repräsentativität insgesamt ein größeres Problem darstellt, wie die Gesamtpunktzahlen der letzten Zeile zeigen. Selbst die drei Benchmarks mit einem positiven Repräsentationspunkt erlangten diese Bewertung nur mit Einschränkungen, wie die Fußnoten *a*, *b* und *c* deutlich machen.

In den Gesamtpunkten der letzten Zeile von Tabelle 2.1 ist neben der Repräsentativität und der Nutzerunterstützung auch die Kategorie Systemverständnis mit wenigen Punkten vertreten. Diese drei Kategorien bilden also Probleme aktueller Benchmarks und werden im Folgenden näher beleuchtet.

2.4.2 Probleme: Repräsentativität, Systemverständnis und Nutzerunterstützung

Es konnte gezeigt werden, dass es drei konkrete Anforderungen an einen idealen I/O-Benchmark gibt, die von aktuell verfügbaren Benchmarks im Allgemeinen nicht oder nicht vollständig erfüllt werden. Diese drei Anforderungen und die Probleme, die aktuelle Benchmarks mit der Erfüllung dieser Anforderungen haben, werden im Folgenden erörtert.

	Vergleichbarkeit	Repräsentativität	Skalierbarkeit	Systemverständnis	Reproduzierbarkeit	I/O-Bezug	Laufzeit	Nutzerunterstützung	**Benchmark-Bewertung**
Bonnie	+	−	+	−	◯	+	+	+	3
IOzone	+	+[a]	+	+	+	+	+	◯	7
b_eff_io	+	◯	+	+	+	+	+	+	7
FLASH I/O	+	−	◯	−	+	+	+	+	3
HPIO	+	+[b]	+	+	+	+	+	◯	7
IOR	+	+[c]	+	+	+	+	+	◯	7
LANL MPI-IO	+	◯	+	◯	+	+	+	+	6
mpi-tile-io	+	−	+	◯	+	+	+	+	5
NPBIO-2.4-MPI	+	−	+	−	+	◯	+	+	3
Parallel Input/Output Test	+	◯	+	+	◯	+	+	◯	5
PIO-Bench	+	◯	+	+	+	+	+	−	5
Punkte	11	-1	10	3	9	10	11	5	

[a] POSIX-I/O-Lasten mit Einschränkungen
[b] nur Charakteristik des noncontiguous I/O variierbar
[c] MPI-IO-Lasten mit Einschränkungen

Tabelle 2.1: Vergleich vorhandener I/O-Benchmarks

Repräsentativität

Ein Benchmark, der dieses Kriterium erfüllt, produziert Ergebnisse, die für möglichst alle Applikationen repräsentativ sind. Da bei jedem Lastverhalten eine für diesen Workload spezifische Leistung erreicht wird, sind Benchmarks, die nur eine Last verwenden, im Allgemeinen nicht repräsentativ. In der Benchmark-Gegenüberstellung verwenden immerhin 4 der 11 Benchmarks nur einen Workload, der kaum an eigene Bedürfnisse anpassbar ist. Weitere 4 Benchmarks unterstützen mehrere Lastverhalten

und können damit einen größeren Umfang an Applikationen abdecken. Aber auch diese Applikationen bieten nicht die Möglichkeit, die Workloads so zu konfigurieren, dass der Nutzer sie an eigene Anforderungen anpassen kann. Eine derart flexible Anpassung des Lastverhaltens erlauben nur drei der 11 Benchmarks. Wenngleich die Entwickler dieser drei Benchmarks, die sich der Klasse der applikationsbasierten Benchmarks zuordnen lassen, eine Konfiguration und flexible Anpassung der Last vorgesehen haben, sind auch bei diesen Softwaresystemen die Workloads nicht beliebig konfigurierbar. *HPIO* bietet die Anpassbarkeit ausschließlich bei dem Aufbau der zu transferierenden Daten. Damit können beliebige Arten des noncontiguous I/O realisiert werden. Jedoch sind alle weiteren Parameter wie beispielsweise, ob die Daten lesend oder schreibend, blockierend oder asynchron transferiert werden sollen, nicht konfigurierbar. Blockierende Zugriffe sind auch bei den Zugriffsmustern der Benchmarks *IOzone* und *IOR* nicht einstellbar. Beide können auch keine spezifische Mischung zwischen lesenden und schreibenden Zugriffen realisieren. Dies ist eine erhebliche Einschränkung, da es ein wesentliches Kriterium des I/O-Workloads von Applikationen ist, wie die lesenden und schreibenden Zugriffe miteinander verflochten sind. Das zentrale Problem dieser applikationsbasierten Benchmarks ist, das die Entwickler in keinem der Fälle ein hinreichend gutes Modell entwickelt haben, um die Last mit ihren möglichen Facetten zu beschreiben. Vor der Entwicklung eines applikationsbasierten Benchmarks muss die Entwicklung eines Lastmodells zur adäquaten Beschreibung von I/O-Workloads stehen.

HPIO

IOZone und IOR

Systemverständnis

Benchmarks sollen helfen, das Verständnis für die Zusammenhänge im System zu steigern. Insbesondere sollen Benchmarks helfen zu verstehen, wie die Systemleistung von einzelnen Systemparametern abhängt. Ein Benchmark kann also das Systemverständnis steigern, wenn er zulässt, einzelne Parameter der Last zu variieren. Dies ist eigentlich ein einfach zu erfüllendes Kriterium, weshalb es sehr erstaunt, dass viele I/O-Benchmarks dieses Kriterium nicht erfüllen. Vorbildlich ist hier der *IOzone*, der eine Ausgabe seiner Messergebnisse in Excel-Dateien erlaubt und den Nutzer so befähigt, durch die Erzeugung mehrdimensionaler Plots die Zusammenhänge zwischen Parametern und Leistung zu visualisieren. Der eingesetzte Bewertungsmaßstab sieht vor, dass Benchmarks eine neutrale Bewertung bekommen, wenn sie eine Variation von Parametern nur manuell mit anschließendem Neustart des Benchmarks ermöglichen. Da derartige Benchmarks grundsätzlich auch das Systemverständnis steigern, wird diese Anforderung nur von den drei der elf Benchmarks mit einer negativen Bewertung gar nicht unterstützt. Zwei dieser drei Benchmarks sind Applikationsbenchmarks, die ein spezifisches Anwendungsverhalten als Grundlage der Last verwenden. Es ist ein Kennzeichen von Applikationsbenchmarks, dass sie in dieser Bewertungskategorie Schwächen haben, da sie häufig so spezifisch sind, dass eine Änderung einzelner Parameter die Implementierung eines neuen Benchmarks zur Folge hätte. Aus diesem Grund wird die Möglichkeit der Variation von Parametern von Entwicklern der

Anwendungsbenchmarks nur in wenigen Fällen (bspw. bei der Variation der Problemgröße) vorgesehen. Zusammenfassend ist zu sagen, dass es grundsätzlich kein Problem für einen Benchmark darstellt, das Systemverständnis zu steigern. Benchmarks mit zahlreichen Konfigurationsmöglichkeiten (wie applikationsbasierte Benchmarks) tun dies von Haus aus. Allerdings sollten im Zuge einer hohen Nutzerfreundlichkeit einzelne Parameter automatisch vom Benchmark variiert werden.

Nutzerunterstützung

Eine gute Nutzerunterstützung wurde den Benchmarks zugesprochen, die den Nutzer sowohl bei Konfiguration als auch bei der Ergebnisauswertung unterstützen. Unterstützung kann dabei durch verschiedene Möglichkeiten erfolgen: Einerseits können Werkzeuge zur Unterstützung zur Verfügung gestellt werden, andererseits können die Ein- und Ausgaben so einfach gestaltet werden, dass keine weitere Unterstützung notwendig ist. Zusätzliche Werkzeuge werden nur von den wenigsten I/O-Benchmarks bereit gestellt. Zwei der untersuchten Softwaresysteme unterstützen eine Exportfunktion ihrer Ergebnisdaten in das Excel-Format, um die Daten dort visualisieren und weiterverarbeiten zu können. Eine Unterstützung bei der Konfiguration wird außer durch mehr oder weniger gute Dokumentation von keinem der Werkzeuge durchgeführt. Aus diesem Grund sind gerade die Benchmarks, die eine komplexe Konfiguration ermöglichen (und damit viele unterschiedliche Lasten erlauben – also eine hohe Repräsentativität aufweisen), in dieser Kategorie mit einer schwachen Bewertung vertreten. Benchmarks hingegen, die sehr einfache Lastverhalten ohne viele Variationsmöglichkeiten besitzen, erhielten in dieser Kategorie eine positive Bewertung, da umfangreiche Nutzerunterstützung bei diesen nicht notwendig ist. Die Konfigurationsmöglichkeiten eines Benchmarks (damit die Repräsentativität seiner Ergebnisse) und die Nutzerunterstützung sind bei den meisten in dieser Arbeit untersuchten Benchmark-Systemen gegenläufig. Dies muss aber nicht der Fall sein. Es ist durchaus möglich, einen I/O-Benchmark zu entwickeln, der in beiden Kategorien eine positive Bewertung erhält.

2.4.3 Kernfragen der vorliegenden Arbeit

Dieses Kapitel zeigte bislang den aktuellen Stand der Forschung im Bereich des I/O-Benchmarkings. Es wurde dargelegt, wo aktuelle Benchmarking-Systeme Schwächen aufweisen. Dazu konnte gezeigt werden, dass es drei zentrale Probleme bei aktuellen I/O-Benchmarks gibt: die Repräsentativität der Ergebnisse, die Steigerung des Systemverständnisses und die Unterstützung des Nutzers bei der Konfiguration des Benchmarks und Ergebnisauswertung. Die Kriterien Repräsentativität und Nutzerunterstützung sind bei den meisten untersuchten Benchmarks gegenläufig, obwohl dies nicht so sein muss. Keiner der untersuchten I/O-Benchmarks erlangte in beiden Bereichen gemeinsam eine positive Bewertung.

Insbesondere im Bereich der Repräsentativität haben alle untersuchten Benchmarks Probleme, da kein Benchmark ein I/O-Last-Modell verwendet, das Lasten einfach und dennoch mit umfassenden Möglichkeiten darstellen kann.

Ziel dieser Arbeit ist die Definition einer Benchmark-Architektur, die diese Probleme behebt. Es wird gezeigt, wie ein I/O-Benchmark aufgebaut sein muss, der alle vorgestellten Kriterien eines optimalen I/O-Benchmarks erfüllt. Besonderer Fokus liegt dabei auf der Lösung des Problems der Repräsentativität. Außerdem wird eine prototypische Implementierung eines Benchmarks vorgestellt, der diese Benchmark-Architektur nutzt.

Zur Definition einer derartigen Benchmark-Architektur ergeben sich die folgenden Kernfragen, die den Fokus der vorliegenden Arbeit bilden: *Kernfragen*

1. Wie ist die Architektur einer Benchmark-Suite gestaltet, die alle Anforderungen eines idealen I/O-Benchmarks und insbesondere die Kriterien Repräsentativität und Nutzerunterstützung erfüllt?

2. Um repräsentative Ergebnisse mit einem Benchmark erzielen zu können, muss die vom Benchmark erzeugte I/O-Last möglichst genau und vollständig beschrieben werden. Genau ist die Beschreibung einer I/O-Last, wenn alle Möglichkeiten von I/O-Anforderungen exakt abgebildet werden können und vollständig ist sie, wenn keine Informationen durch die Lastbeschreibung (bspw. durch statistische Mittelwertbildung) verloren gehen. Diese Beschreibung ist einerseits zur Dokumentation und andererseits als Lastdefinition für den Benchmark von Bedeutung. Wie kann eine Lastbeschreibung aussehen, die für den spezifischen Anwendungsfall ausreichend vollständig und genau ist, ohne angesichts der Komplexität des Speichersystems zu umfangreich zu werden?

3. Wie können Lastbeschreibungen erzeugt werden, die reellen Lasten entsprechen und damit repräsentativ und nutzerrelevant sind? Es stellt sich insbesondere die Frage, wie dies erreicht werden kann, ohne den Nutzer zu überfordern. Wie können solche Lastbeschreibungen also einfach und leicht verständlich erstellt und präsentiert werden?

Die vorliegende Arbeit wird diese Kernfragen beantworten und damit einen Weg zu einer neuen Art des Sekundärspeicher-Benchmarkings weisen. Dazu werden im Kapitel 3 die theoretischen Grundlagen gelegt, die der Erstellung einer neuen Benchmark-Architektur zu Grunde liegen.

Kapitel 3

Methoden der Leistungsanalyse von I/O-Systemen

Contrary to common belief, performance evaluation is an art.

– RAJ JAIN [1]

Nachdem im Kapitel 2 die Zielstellung dieser Arbeit in Form von drei zentralen Fragestellungen formuliert wurde, wird dieses Kapitel die theoretischen Grundlagen zur Beantwortung der drei Kernfragen erarbeiten. Das Kapitel bildet damit die Basis der praktischen Betrachtungen, die in den anschließenden Kapiteln folgen.

Ausgehend von den Teilschritten, die für eine fehlerfreie und effiziente Leistungsanalyse durchgeführt werden müssen, wird eine Softwarearchitektur skizziert, die den Aufbau einer Benchmarking-Toolchain ermöglicht und die besonderen Anforderungen des Benchmarkings und insbesondere des Benchmarkings von I/O-Systemen verteilter Rechensysteme berücksichtigt. Zur Präzisierung der Architektur sind theoretische Betrachtungen notwendig, die eine genaue Beschreibung des I/O-Lastverhaltens von Applikationen erlauben.

Grundlage der theoretischen Betrachtungen wird der I/O-Workload-Raum sein, der in diesem Kapitel definiert wird und eine Modellierung des Zugriffsverhaltens verschiedener Applikationen ermöglicht. Mit Hilfe des I/O-Workload-Raumes wird eine adäquate Darstellungsmethode gefunden, die das Zugriffsverhalten von Applikationen formal beschreibt, um damit die Grundlage für den Vergleich unterschiedlicher I/O-Benchmarking-Methoden zu legen. Es wird gezeigt, dass vorhandene I/O-Benchmarking-Methoden zahlreiche Probleme besitzen, und es wird der Aufbau eines optimalen I/O-Benchmarks im Hinblick auf die in Abschnitt 2.3.2 präsentierten Zielanforderungen vorgestellt.

3.1 Schritte der Leistungsanalyse

Mit der Leistungsanalyse von Rechensystemen sind aufgrund der hohen Komplexität moderner Computer zahlreiche Fallstricke verbunden, die dazu führen können, dass selbst von erfahrenen Anwendern Ergebnisse erzeugt werden, die hinsichtlich des geplanten Einsatzzweckes bedenklich und teilweise sogar falsch sind. Dies kann nur verhindert werden, wenn der gesamte Vorgang der Leistungsanalyse mit höchster Sorgfalt durchgeführt wird. Um die Anzahl möglicher Fehler zu verringern, bietet es sich an, den Nutzer bei dem Analysevorgang mit Hilfe von automatischen Werkzeugen zu unterstützen.

In [1] werden die häufigsten Fehler beschrieben, die bei der Leistungsanalyse von Rechnersystemen entstehen. Zur Vermeidung dieser Fehler wird ein systematisches Vorgehen bestehend aus 10 Teilschritten vorgeschlagen.

Diese Teilschritte werden im Folgenden kurz wiedergegeben:

1. Definition der Ziele der Leistungsanalyse, des zu evaluierenden Systems und der Systemgrenzen
2. Definition der zu vermessenden Dienste des Systems
3. Auswahl einer der drei Leistungsanalysemethoden: Simulation, analytische Modellierung oder Messung
4. Festlegung der zu verwendenden Metriken, die definieren, wie die ermittelte Leistung dargestellt wird
5. Bestimmung der Parameter zur Definition der Last
6. Definition der Variablen, in deren Abhängigkeit die Leistung ermittelt wird (sogenannte Faktoren)
7. Festlegung der Last (des Workloads) für die Experimente
8. Durchführung der Experimente
9. Analyse und Interpretation der Daten
10. Präsentation der Ergebnisse in der Form, dass Interessenten die wesentlichen Erkenntnisse sofort sehen

Zur Vermeidung von Fehlern müssen alle zehn Schritte durch den Nutzer gewissenhaft ausgeführt werden. Um sicherzustellen, dass der Nutzer keinen Schritt vergisst oder fehlerhaft ausführt, muss eine weitestgehende Unterstützung durch Werkzeuge erfolgen, die einen automatisierten Arbeitsablauf steuern. Damit werden Fehler vermieden, die ohne diese Hilfestellung bei der Leistungsanalyse oftmals auftreten. Es wird sich im Folgenden zeigen, dass eine automatisierte Unterstützung durch Softwarewerkzeuge nicht in allen Schritten erfolgen kann. Beispielsweise können die Ziele

einer Leistungsanalyse nur durch den Nutzer selbst definiert werden. Die beschriebenen zehn Teilschritte werden in dieser Arbeit vier Phasen zugeordnet, von denen der Nutzer in drei der vier Phasen durch Werkzeuge unterstützt werden kann. Diese Zuordnung vereinfacht die Handhabung der zahlreichen notwendigen Benchmarking-Teilschritte in den folgenden Betrachtungen. Die Vorstellung der einzelnen Phasen geschieht in Hinblick auf eine Unterstützung des Nutzers durch Software-Werkzeuge in jeder der Phasen.

3.2 Phasen des Benchmarkings

Abbildung 3.1 veranschaulicht die Teilschritte der Leistungsanalyse, die zugehörigen Phasen und Werkzeuge grafisch. Jede der vier Phasen wird im Folgenden mit dem zur Phase gehörenden Werkzeug vorgestellt.

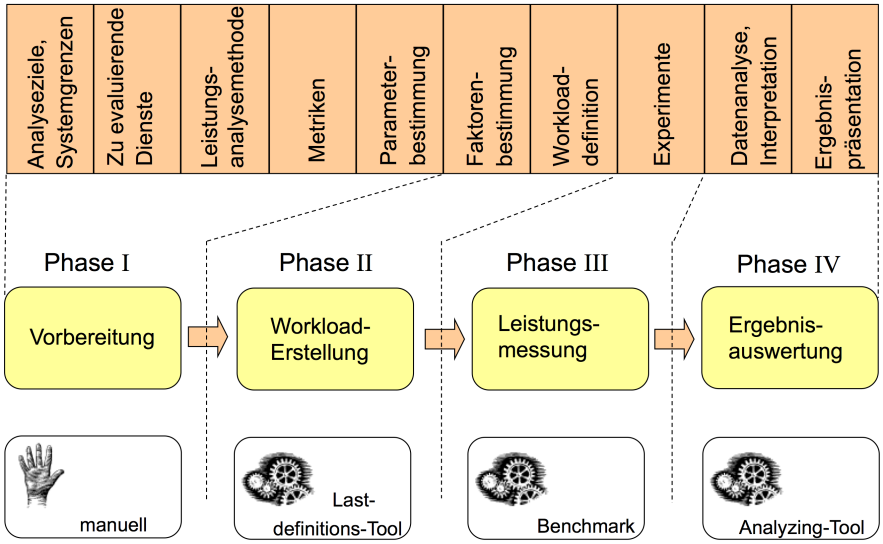

Abbildung 3.1: Vier Phasen der Leistungsanalyse und deren unterstützende Werkzeuge

3.2.1 Phase I: Vorbereitung

Phase I, die den Teilschritten 1 bis 5 des Benchmarkings entspricht, beinhaltet die Vorbereitung, die durch den Nutzer durchgeführt werden muss. Nur der Nutzer kennt die *Ziele der Leistungsanalyse* (Schritt 1) und weiß, welche *Dienste* (Schritt 2) des Systems untersucht werden sollen. Abhängig von diesen beiden Schritten, muss sich der Nutzer für eine *Analysemethode* entscheiden (Schritt 3). Im Detail hängt diese Entscheidung von zahlreichen Kriterien ab, die in Tabelle 3.1 dargestellt sind. Die Kriterien sind nach ihrer Wichtigkeit sortiert, wobei auch diese von Projekt zu Projekt

Kriterium	Analytische Modellierung	Simulation	Messung
Zugang zum System	nicht notwendig	nicht notwendig	notwendig
zeitliche Verfügbarkeit des Ergebnisses	schnell	mittel	schwankend
Werkzeuge	math. Werkzeuge, bspw. Computeralgebra-Tools	Simulations-Tools	Benchmarks, Instrumentations-werkzeuge
Genauigkeit	niedrige Genauigkeit	mittlere Genauigkeit	variierend
Alternativen-Untersuchung	gut möglich	möglich	schwierig
Kosten	niedrig	mittel	hoch
Überzeugungskraft der Ergebnisse	wenig	mittel	hoch

Tabelle 3.1: Kriterien zur Auswahl eines Leistungsanalyseverfahrens [1]

schwanken kann. In dieser Form entspricht der Inhalt der Tabelle den Erfahrungen des Autors von [1].

Zugang zum System

Grundsätzlich ist das wichtigste Kriterium, ob für die Leistungsanalyse Zugang zum zu vermessenden System besteht. Nur in diesem Fall ist es überhaupt möglich, Messungen durchzuführen.

zeitliche Verfügbarkeit des Ergebnisses

Häufig ist ebenfalls wichtig, wann ein Ergebnis zur Verfügung stehen muss, da die Analysemethoden unterschiedliche Zeitanforderungen haben. Oftmals erzielt man die schnellsten Ergebnisse mit einer analytischen Modellierung, die aber im Allgemeinen nicht mehr als eine ungenaue Abschätzung darstellt. Simulationen dauern oftmals länger, insbesondere dann, wenn das Simulationsmodell erst erstellt werden muss. Steht eines zur Verfügung, kann auch eine Simulation verhältnismäßig zeitnah Ergebnisse liefern. Die erforderliche Zeit für Messungen ist hingegen am schwierigsten einzuschätzen. Wenn der Messvorgang sofort gut funktioniert, stehen Ergebnisse sehr schnell zur Verfügung. Allerdings gibt es gerade bei der Messung sehr häufig unvorhergesehene Probleme, da das System in seiner gesamten Komplexität als Analysegrundlage verwendet wird, während alle anderen Analysemethoden vereinfachte Modelle einsetzen.

Werkzeuge

Die Werkzeuge der Analyse sind je nach Analyseverfahren mathematische Werkzeuge, Simulations-Tools oder Mess-Software, wie Benchmarks. Wenn spezifische Werkzeuge oder Know-How zur Erstellung der Analyse bzw. zur Nutzung der Analysewerkzeuge nicht vorhanden sind, kann dieser Punkt bereits die Entscheidungsgrundlage für die Auswahl des Analyse-Verfahrens darstellen.

Genauigkeit

Die Genauigkeit des Ergebnisses sollte in jedem Fall so hoch wie möglich sein. Allerdings hängt sie ebenfalls vom verwendeten Verfahren ab. Im Allgemeinen gilt: Je abstrakter des Modell, desto ungenauer das Ergebnis. Allerdings kann eine sehr hohe Komplexität des Systems wiederum die Genauigkeit des Ergebnisses in Frage stellen, da die Anzahl der konfigurierbaren Parameter bei komplexeren Modellen steigt. Bei Messungen kann die Komplexität so hoch werden, dass die Messergebnisse abhängig von der verwendeten Last sehr stark vom gesuchten Ergebnis abweichen.

„Alternativen-Untersuchung"

Sehr häufig ist das Ziel einer Leistungsanalyse die Untersuchung verschiedener Alternativen, um die beste herauszufinden. Eine derartige „Alternativen-Untersuchung" ist im Allgemeinen bei einfachen Modellen leicht möglich, da sie schnell verstanden und modifiziert werden können. Entsprechend ist eine derartige Untersuchung oftmals bei analytischen Modellen leichter als bei Simulationsmodellen oder gar bei wirklich existierenden Systemen. Dennoch ist die Variation einzelner Parameter bei allen Analysevarianten möglich und gebräuchlich, weshalb gerade dieses Kriterium im Einzelfall gesondert untersucht werden muss.

Kosten

Der Kostenfaktor ist sehr häufig ein entscheidender Grund zur Auswahl einer Vorgehensweise. Auch bei der Leistungsanalyse ist dies oft der Fall. Kosten hängen direkt mit der Analysezeit zusammen, weshalb die in der Tabelle angegebenen Kosten für die analytische Modellierung am geringsten sind. Im Einzelfall kann aber auch eine hier als kostenintensiv angegebene Analysemethode günstig sein. Insbesondere ist die Messung nur dann die teuerste Analysemethode, wenn die Systeme erst beschafft werden müssen oder hohe Kosten für den Zugang zum System entstehen.

Überzeugungskraft der Ergebnisse

Die Überzeugungskraft der Analyseergebnisse ist dann von Bedeutung, wenn diese wichtigen Entscheidern als Entscheidungshilfe präsentiert werden müssen. In solchen Fällen ist die Überzeugungskraft von analytisch ermittelten Ergebnissen oft geringer als die von Simulationsergebnissen, da die Modelle, die hinter der analytischen Modellierung stehen, oftmals in der Kürze der Präsentation nicht übermittelt und verstanden werden. Am leichtesten überzeugen jedoch Messergebnisse, da sofort einsichtig ist, dass diese am realen System durchgeführt wurden.

Die hier angegebenen Prioritäten der einzelnen Kriterien sind im Einzelfall eines spezifischen Projektes genau zu überdenken. Sie dienen an dieser Stelle lediglich als erster Anhaltspunkt. Auch die angegebenen Relationen innerhalb der Tabelle 3.1 können sich von Projekt zu Projekt unterscheiden. In besonderen Projekten kommen außerdem weitere projektspezifische Entscheidungskriterien hinzu. Es ist aus diesem Grund nur schwer eine automatische Entscheidungshilfe realisierbar. Am ehesten kann ein Expertensystem Unterstützung leisten, dessen Erstellung aber den Fokus dieser Arbeit verlässt.

Ein vollständiger Entscheidungsbaum, der durch Abfrage aller Kriterien bei der Erstellung eines eindeutigen Ergebnisses hilft, ist nur definierbar, wenn die Prioritäen der Kriterien eindeutig projektunabhängig definiert werden können. Da dies nicht der Fall

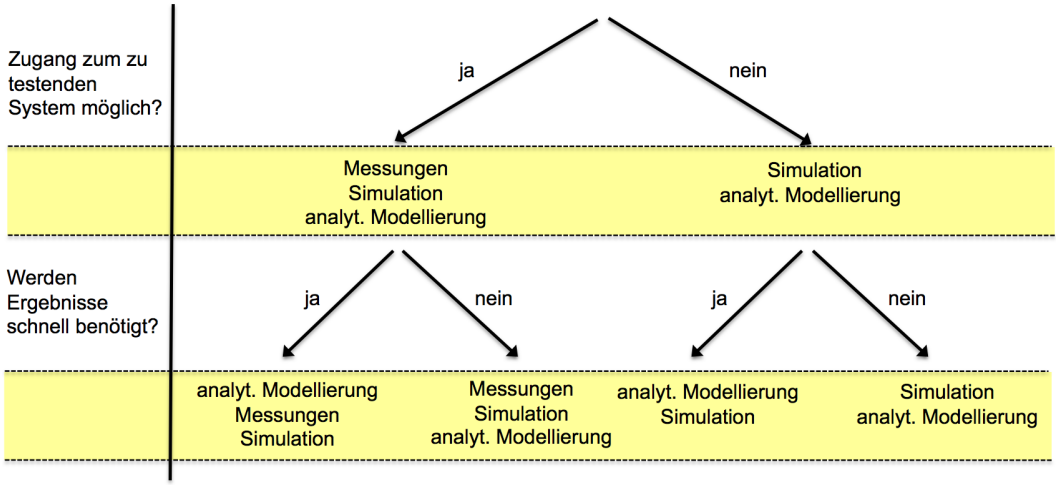

Abbildung 3.2: Entscheidungsbaum zur Auswahl eines Leistungsanalyseverfahrens

ist, wird an dieser Stelle ein einfacher Entscheidungsbaum als erste Entscheidungshilfe definiert.

Entscheidungsbaum

Der Entscheidungsbaum ist in Abbildung 3.2 dargestellt und bezieht nur die ersten beiden angegebenen Kriterien der Tabelle 3.1 ein, da das Kriterium „Systemzugang" notwendigerweise beachtet werden muss und die Analysezeit in vielen Projekten eine hohe Relevanz besitzt. In den Knoten des Entscheidungsbaums stehen die, jeweils in diesem Entscheidungszweig, einsetzbaren Leistungsanalyseverfahren. Wenn dort mehrere Verfahren aufgeführt sind, gibt deren Reihenfolge die Priorität des Einsatzes an. Die mittels des Entscheidungsbaums ermittelten Verfahren müssen aber in jedem Fall gegen weitere im Projekt relevante Kriterien abgeglichen werden. Oftmals ist es auch sinnvoll, mehrere Analyseverfahren einzusetzen, um Ergebnisse miteinander zu vergleichen.

Hat sich der Nutzer für eine spezifische Analysemethode und Analyseziele entschieden, sind die Schritte 4 und 5, also die verwendete *Metrik* und die unterstützten *Parameter* durch die Werkzeuge, für die der Nutzer sich entschieden hat, festgelegt. Lediglich bei einer Neuentwicklung, wie beispielsweise einem neuen analytischen Modell, hat der Nutzer an dieser Stelle detaillierter über die zu verwendenden Metriken und Parameter nachzudenken. Da der Schwerpunkt dieser Arbeit in der Vermessung von I/O-Systemen durch Benchmarkprogramme liegt, wird davon ausgegangen, dass der Nutzer sich für die Verwendung eines I/O-Benchmarks zur Leistungsmessung eines I/O-Systems entschieden hat. Es stehen dann die Metriken und Parameter fest, da diese durch das Benchmarkprogramm bestimmt werden. Typische I/O-Benchmarks haben die Bandbreite (angegeben in MB/s) und Latenz (angegeben in ms) beim Zugriff auf den Sekundärspeicher als Metriken. Zwischen den Benchmarks gibt es bezüglich der verwendeten und konfigurierbaren Parameter große Unterschiede. Dieses Problem wurde als ein zentrales Problem aktueller Benchmarks im letzten Kapitel bereits erläutert. Der Benchmarknutzer muss sich dieser Unterschiede bewusst sein und sollte

nur Benchmarks verwenden, die eine für seine Anwendung ausreichende Anzahl an Parametern in die Betrachtung mit einbeziehen, um hinreichend genaue Ergebnisse zu erzielen.

3.2.2 Phase II: Workload-Erstellung

Phase II der Leistungsanalyse beinhaltet die Workload-Erstellung respektive die Teilschritte 6 und 7 der oben genannten Teilschritte der Leistungsanalyse. Zur Workload-Erstellung gehört die *Definition von Faktoren*, also von Parametern deren Einfluss auf die Ergebnisse durch Variation im Rahmen der Analyse untersucht werden soll (Schritt 6). Das Analysewerkzeug legt durch seine Möglichkeiten grundsätzlich fest, welche Parameter als Faktoren verwendet werden können. Am Beispiel der I/O-Benchmarks wurde im letzten Kapitel gezeigt, dass es Benchmarks gibt, die gar keine Variation von Parametern (und damit keine Faktoren-Definition) zulassen, während einige Benchmarks eine Variation zahlreicher Parameter vorsehen. Da der Nutzer die Zielvorstellungen der Untersuchung kennt, ist es seine Aufgabe, zu definieren, welche vom Werkzeug zur Verfügung gestellten Faktoren diese Zielvorstellungen am besten umsetzen. Vom Werkzeug kann an dieser Stelle dadurch Unterstützung geleistet werden, dass alle möglichen Parameter gut und für den Anwender leicht verständlich beschrieben werden. Die Auswahl der Faktoren liegt letztendlich aber in der Hand des Nutzers.

Faktoren-bestimmung

Neben der Faktorenbestimmung gehört die Definition der festen, nicht-variierbaren *Parameter* zur Workload-Erstellung, weil diese signifikant den Charakter der Last bestimmen (Schritt 7). Workloads können manuell durch den Nutzer definiert werden.

Lastparameter

Lastdefinitionen für ein komplexes System, wie das I/O-System, werden oftmals sehr umfangreich. Um dem Nutzer in Phase II zu befähigen, nachvollziehbare und realistische Lasten zu erzeugen, ist eine Werkzeugunterstützung unumgänglich. In der Praxis der I/O-Performance-Messung wird diese Anforderung bislang nur sehr selten berücksichtigt.

Lastdefinitions-Tool

Lastdefinitionswerkzeuge unterstützen den Nutzer beispielsweise durch visuelle Möglichkeiten bei der Erstellung der Lasten. So ist es möglich, die zahlreichen Parameter, die die Last beeinflussen, als mehrdimensionalen Parameterraum grafisch zu visualisieren, um dem Nutzer die Möglichkeit zu geben, Punkte innerhalb des Raumes zu definieren. Denkbar ist auch die Vordefinition verschiedener Lastszenarien, die durch den Nutzer an seine Bedürfnisse angepasst werden.

Andererseits können Lastdefinitionswerkzeuge Lasten von vorhandenen Applikationen analysieren und automatisiert mittels der Lastparameter darstellen. Diese Art der Lastendefinition erzeugt so ein getreues Abbild der Applikationslast, um dieses während des Analysevorganges zu verwenden.

Zentrales Merkmal eines jeden Lastdefinitionswerkzeugs ist die Möglichkeit einer Lastdarstellung, die einfach ist, aber exakt genug, um den Nutzeranforderungen zu entsprechen. Wie eine derartige Lastdarstellung aussieht, ist eine zentrale Fragestellung, die innerhalb dieser Arbeit beantwortet wird.

3.2.3 Phase III: Leistungsmessung

Phase III der Leistungsanalyse entspricht dem eigentlichen *Experimentiervorgang*. Sollte der Nutzer eine Leistungsmessung mittels Benchmark durchführen, entspricht sie der Applikationsausführung des Benchmarkprogrammes. Im Falle einer Leistungsanalyse durch Simulation entspricht diese Phase der Simulationsdurchführung. Häufig wird Phase III der kompletten Leistungsanalyse gleichgesetzt, obgleich sie in den dargestellten zehn Teilschritten der Leistungsanalyse nur dem Schritt 8 entspricht. Phase III ist also wichtig – alle anderen Phasen dürfen aber nicht unterschätzt werden und sind für ein aussagekräftiges Ergebnis ebenfalls von großer Bedeutung.

Benchmark

Da in dieser Arbeit der Fokus auf I/O-Benchmarks liegt, wird angenommen, dass das Werkzeug zur Leistungsanalyse dieser Phase ein Benchmark ist. Der Benchmark, als Messwerkzeug, das die Nutzerunterstützung in dieser Phase durchführt, verwendet die im Lastdefinitions-Tool erzeugte Lastbeschreibung und erzeugt eine Last, die dieser entspricht, um Leistungsdaten zu sammeln. Entsprechend besteht der Benchmark aus zwei Komponenten: Einem Lastgenerator und einem Messwerkzeug.

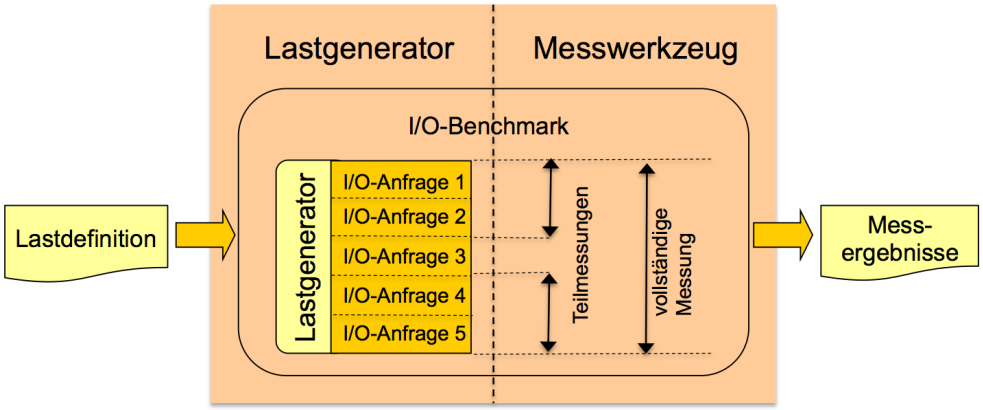

Abbildung 3.3: Lastgenerator und Messwerkzeug eines I/O-Benchmarks

Abbildung 3.3 zeigt dies am Beispiel eines I/O-Benchmarks. Der Lastgenerator des Benchmarks erzeugt I/O-Anfragen entsprechend der Lastdefinition, während die Messkomponente zeitgleich Laufzeitmessungen einzelner Anfragen oder Anfragenblöcke durchführt. Es kann eine Messung über die gesamte Laufzeit des Benchmarks erfolgen, um eine Gesamtleistung des Benchmarks zu ermitteln. Allerdings kann durch die so ermittelte Gesamtleistung kaum auf spezifische Eigenschaften des I/O-Systems (wie beispielsweise das Verhältnis des Schreib- und Lesedurchsatzes) geschlossen werden.

Für derartige Ergebnisse müssen zusätzlich Messungen zu bestimmten Zeitpunkten während der Lastgeneration durchgeführt werden (z. B. vor und nach allen Lese- bzw. Schreibaufrufen). Dies ermöglicht während der Ergebnisauswertung ein feinaufgelöstes Bild der I/O-Leistung.

Um zwischen I/O-Anfragen eine Zeitmessung durchführen zu können, muss die Lastkomponente des Benchmarks sehr feingranular instrumentarisiert werden, d. h. es müssen zwischen den betreffenden I/O-Anforderungen Funktionsaufrufe eingefügt werden, die eine Protokollierung des zeitlichen Ablaufs durchführen. Eine komplette Trennung von Last- und Messkomponente ist deshalb im Allgemeinen nicht möglich. Es findet eine implementierungsseitige Vermischung beider eigentlich logisch getrennter Komponenten statt. Beide Anteile werden aus diesem Grund in einem Werkzeug, dem Benchmark, zusammengefasst.

Es spricht bei dieser Implementierung nichts gegen eine Nutzung des Benchmarks als reines Lastgenerations-Werkzeug in entsprechenden Szenarien.

3.2.4 Phase IV: Ergebnisauswertung

Eine der häufig vernachlässigten Phasen der Leistungsanalyse ist *Phase IV*, die Ergebnisauswertung. Sie entspricht den Teilschritten 9 und 10, also sowohl der *Analyse und Interpretation* der Daten, als auch deren *Präsentation*. Insbesondere dieser letzte Punkt wird häufig vernachlässigt. Leistungsanalysen werden im Allgemeinen durch einen Auftraggeber veranlasst, der Entscheidungen anhand der Ergebnisse fällen muss. Dies kann ein Wissenschaftler sein, der für eine Simulation den optimalen Rechner unter den im Rechnerpool vorhandenen Systemen herausfinden möchte. Dies kann aber auch der Rechenzentrumsleiter sein, der im Rahmen neuer Beschaffungen den leistungsfähigsten Clustercomputer für spezifische in der Universität verwendete Applikationen herausfinden muss. Damit er schnell und richtig entscheiden kann, ist es wichtig, dass die bedeutenden (und nicht alle) Zusammenhänge schnell erkennbar dargestellt werden. Zu einer erfolgreichen Präsentation gehört neben der Entscheidung für einen Präsentationstyp (Tortendiagramm, Balkendiagramm, . . .) auch die Entscheidung über relevante und weniger relevante Ergebnisse. An dieser Stelle können Werkzeuge eine Unterstützung dadurch bieten, dass sie Daten in verschiedenen Formen darstellen und exportieren. Auch können sie eine Filterung der Daten durchführen, um zu extrahieren, welche Faktoren einen relevanten Einfluss auf die Leistung haben. Eine endgültige Entscheidung über die beste Darstellungsvariante sollte aber immer der Nutzer durchführen, da eine gute Präsentation an den Adressaten der Vorstellung angepasst sein muss.

Analyzing-Tool

Nachdem die vier Phasen der Leistungsanalyse vorgestellt wurden, dienen diese nun als Grundlage zur Implementierung einer Werkzeugkette, die ein fehlerfreies, exaktes und nutzerfreundliches I/O-Benchmarking ermöglicht. Dazu werden im Folgenden

theoretische Aspekte des I/O-Benchmarkings in der Reihenfolge der Phasen, denen diese Aspekte zugehörig sind, beleuchtet.

3.3 Definition einer Lastbeschreibung

Zentrales Werkzeug der Phase II der Leistungsanalyse ist das Lastgenerationswerkzeug. Es wurde bereits erläutert, dass dieses Werkzeug die formale Darstellung einer I/O-Last so ermöglichen muss, dass die Darstellung eine hinreichend genaue Rekonstruktion des gewünschten Lastverhaltens erlaubt. Es wird sich im Folgenden zeigen, dass für eine exakte Abbildung des I/O-Verhaltens einer Applikation eine sehr komplexe Repräsentation erforderlich ist. Ziel des Abschnitts ist es deshalb, eine Darstellung zu entwickeln, die eine hohe Genauigkeit bei der Beschreibung des Lastverhaltens mit skalierbarer Komplexität ermöglicht.

Dazu wird zuerst der I/O-Workload-Raum als theoretische Grundlage der Lastbeschreibung eingeführt, bevor im Anschluss ein Vorschlag für eine Applikations-Lastbeschreibung vorgestellt wird.

3.3.1 Lastparameter des I/O-Subsystems

I/O-Workload

Die Leistungswerte des Sekundärspeichersystems bei spezifischen Anwendungen resultieren wesentlich aus deren Zugriffsverhalten und der damit verbundenen Last, auch I/O-Workload genannt. Der Workload ist eine somit Folge des Zugriffsverhaltens der Applikation. Dennoch werden in dieser Arbeit die Begriffe *Workload* und *Zugriffsverhalten* gleich gesetzt, da sie unmittelbar miteinander verknüpft sind. Bei vielen Speichersystemen hat beispielsweise ein ausschließlich lesender Zugriff wesentlich andere Leistungs-Charakteristika als ein schreibender Zugriff.

Im Rahmen dieser Arbeit werden neben dem Lese- und Schreibanteil von Zugriffen weitere Parameter bestimmt, die die I/O-Leistung einer Applikation beeinflussen.

Der erste Schritt zur Ermittlung einer I/O-Lastbeschreibung ist die Erkennung von Parametern, die die von einer Applikation erzeugte I/O-Last beeinflussen. Ein I/O-System hat aufgrund seiner hohen Komplexität eine ganze Reihe derartiger Parameter.

System- und Lastparameter

Parameter werden in Systemparameter und Lastparameter unterteilt, wobei erstgenannte durch das System, also Hardware und Systemsoftware bestimmt werden und die anderen durch die konkreten Anfragen an das System, also durch die Last, die die Applikation erzeugt [1].

Tabelle 3.2 zeigt die wichtigsten Parameter eines Rechensystems, die die I/O-Leistung beeinflussen. Zur Ermittlung der Parameter wurden die Komponenten der in Abschnitt 2.1.1 beschriebenen Architektur eines I/O-Subsystems als Grundlage verwendet. Jede einzelne Komponente hat aufgrund ihrer Realisierung einen I/O-Einfluss,

Komponente	#	Parameter	Typ
Applikation	1	Anordnung, Frequenz und Art der I/O-Anforderungen	Last
	2	Anzahl der Prozesse	Last
I/O-Bibliothek/ C-Bibliothek	3	Umsetzung der I/O-Schnittstelle (damit verbundene Optimierungen)	System
	4	Realisierung des Betriebssystemkern-Einsprungs	System
Virtual File System	5	Realisierung der Virtualisierung der Dateisysteme	System
Dateisystem	6	Dateisystemorganisation	System
	7	lokale/entfernte Speicherung	System
	8	umgesetzte Optimierungen	System
Cache-System	9	Größe des Hauptspeichers (Sek.-Speicher-Cache) bzw. des Hauptspeicher-Caches	System
	10	Zugriffsgeschwindigkeit	System
	11	Cache-Organisation	System
Treiber	12	Umsetzung der Zugriffsstrategie (z. B. Fahrstuhlalgorithmus)	System
	13	Umsetzung von Optimierungen	System
Disk/Netzwerk	14	Bandbreite	System
	15	Latenz	System
	16	Umsetzung von Fehlerkorrekturmaßnahmen	System
	17	Optimierungen (bspw. Festplattencache)	System

Tabelle 3.2: Wichtige I/O-beeinflussende Parameter eines Rechensystems

der durch mehrere Parameter spezifiziert werden kann. Die so ermittelten Lastparameter werden in der Tabelle recht abstrakt beschrieben. Letztendlich kann jeder dieser abstrakten Parameter selbst durch mehrere funktionsspezifische Parameter abgebildet werden, wenn eine genauere Beschreibung erwünscht ist. An dieser Stelle würde eine komplette Aufsplittung der angegebenen Parameter allerdings die Betrachtung erheblich erschweren.

In Tabelle 3.2 werden wesentliche Parameter der einzelnen Komponenten eines I/O-Subsystems angegeben und nach System- oder Lastparameter klassifiziert. Dabei werden, beginnend in der höchsten Schicht eines I/O-Systems, Parameter bestimmt, die die Applikation beeinflussen kann. Jeder Parameter erhält zur späteren Referenzierung eine Nummer.

Applikation

Die Applikation erzeugt selbst die Last auf das I/O-System, also die Reihenfolge, Frequenz und Art der Anforderungen (1). Die möglichen Arten werden durch die I/O-Schnittstelle definiert. So sind je nach Unterstützung durch die Schnittstelle blockierende oder nicht-blockierende Lese- oder Schreibzugriffe bzw. eine ganze Reihe weite-

rer Zugriffsarten möglich. Offensichtlich ist allein dieser Lastparameter so vielschichtig, dass er selbst durch zahlreiche Werte charakterisiert werden muss. Die Applikation bestimmt die Last auf das I/O-System auch durch die Anzahl paralleler Prozesse (2), da jeder Prozess Komponenten des Gesamtsystems (z. B. Prozessor, Speicher) verwendet und die Anzahl der Prozesse direkt eine höhere Belastung der Komponenten zur Folge hat.

C-Bibliothek/ I/O-Bibliothek

Die C-Bibliothek besitzt geringen Einfluss auf die I/O-Performance, da sie nur eine Weitergabe von POSIX-I/O-Aufrufen der Applikation an das Betriebssystem durchführt. Der dabei durchgeführte Einsprung in den Betriebssystemkern kann je nach Realisierung Leistung kosten (4), da typischerweise ein Kopiervorgang der Daten in den Betriebssystemadressraum notwendig ist. Allerdings ist dieser Vorgang durch die Implementierung des Betriebssystems festgelegt und kann durch Applikationen nicht beeinflusst werden. I/O-Bibliotheken hingegen implementieren zahlreiche Optimierungen (3), wie beispielsweise eine lokale Zwischenspeicherung von Daten beim Zugriff auf entfernte Dateisysteme. Diese haben natürlich einen erheblichen Leistungseinfluss, sind aber aufgrund der Unterschiedlichkeit der I/O-Bibliotheken kaum feiner definierbar. In jedem Fall sind diese Systemparameter nur zu einem geringen Grad durch die Applikation beeinflussbar. Einige I/O-Bibliotheken bieten die Möglichkeit, einzelne Parameter durch die Applikation anzupassen. Erschwerend kommt aber hinzu, dass die Einstellungsmöglichkeiten spezifisch für die I/O-Bibliothek sind und sich von Bibliothek zu Bibliothek unterscheiden können. Gewichtig ist an dieser Stelle das Problem, dass I/O-Bibliotheken ihre I/O-Zugriffe letztendlich auch über die C-Bibliothek durchführen müssen. Bestimmte leistungsoptimierende Möglichkeiten, die die I/O-Schnittstelle den Applikationen anbietet, sind u. U. aber nicht auf die POSIX-Schnittstelle der C-Bibliothek abbildbar. An dieser Stelle kommt es dann zu Leistungsverlusten durch Abbildungsprobleme.

virtuelles Dateisystem

Das virtuelle Dateisystem, das eine Abbildung der I/O-Aufrufe auf die im besonderen Fall gewünschte Dateisystemimplementierung durchführt, besitzt keinen besonderen Leistungseinfluss. Als zusätzliche Softwareschicht erzeugt sie eine Leistungseinbuße, die abhängig von der verwendeten Implementierung ist (5).

Dateisystem

Das Dateisystem wiederum hat einen sehr großen Einfluss auf die I/O-Leistung. Mitentscheidend kann sein, wie das Dateisystem Daten organisiert (tabellenbasiert, Inodebasiert) (6), wo es sie ablegt (lokal oder entfernt) (7) und welche Optimierungsverfahren es dabei implementiert (8). Normalerweise ist ein Einfluss von Applikationen auf diese Dateisystem-Interna ausgeschlossen. In Einzelfällen können bestimmte Konfigurationsparameter eines Dateisystems durch den Systemadministrator angepasst werden, um eine manuelle Optimierung des Systems durchzuführen [77].

Cache-System

Caches, deren Aufgabe die Beschleunigung von Zugriffen auf langsame Ebenen der Speicherhierarchie ist, haben naturgemäß einen Leistungseinfluss, der von vielen Faktoren abhängt. I/O-Zugriffe werden im Sekundärspeicher-Cache, für den typischerweise Hauptspeicher verwendet wird, zwischengespeichert. Der Hauptspeicherzugriff

selbst wird mit Hilfe des Hauptspeicher-Caches beschleunigt. Einen leistungsbetreffenden Einfluss haben an dieser Stelle die Größen (9) und Zugriffsgeschwindigkeiten (10) der Caches. Weder diese Parameter, noch die Organisation des Caches, d. h. Assoziativität bzw. Organisationsstruktur im Betriebssystem (11) sind durch Applikationen variierbar.

Treiber und Hardware

Der Treiber und die Hardware, die mit einem I/O-Zugriff angesprochen werden, haben letztlich einen weiteren Leistungseinfluss. Der Treiber kann verschiedene Optimierungen (13) und Strategien des Zugriffs (12) implementieren. Beispielsweise verringert der Fahrstuhlalgorithmus [22] die Anzahl der Neupositionierungen des Schreib-/Lesekopfes einer Festplatte, indem mehrere Anforderungen gesammelt werden. Diese werden nicht in der Reihenfolge abgearbeitet, in der sie im System eintreffen, sondern in der Reihenfolge, in der ihre Zielpositionen auf der Platte angeordnet sind. Die Platte selbst bzw. im Falle von entfernten Speichern das Netzwerk bestimmen mit ihrer Leistung letztlich ebenfalls die I/O-Gesamtleistung. Allerdings sind auch diese Systemparameter, (14 bis 17) nicht durch Applikationen variierbar.

Die angegebenen Parameter sind längst nicht alle, die einen Einfluss auf die I/O-Leistung haben. Es wurden in diesem Fall nur die Parameter der Komponenten angegeben, die direkt dem I/O-Subsystem angehören. Ein Rechensystem besteht aus wesentlich mehr Komponenten, die in bestimmten Umfang ebenfalls einen Leistungseinfluss besitzen. Zum Beispiel hat die Geschwindigkeit des Prozessors, der I/O durchführt, natürlich einen Einfluss, da seine Geschwindigkeit die Abarbeitungszeit von I/O-Anforderungen im I/O-Subsystem mitbestimmt. Allerdings wird an dieser Stelle davon ausgegangen, dass Parameter, die das I/O-System nicht direkt beeinflussen, einen vergleichsweise geringen Einfluss besitzen. Sie werden deshalb im Folgenden ignoriert.

Für die Definition eines I/O-Benchmarking-Softwaresystems sind nur die Parameter von Bedeutung, a) die einen nennenswerten Einfluss auf die I/O-Leistung haben, und b) auf die die Benchmarking-Applikation Einfluss nehmen kann. Alle übrigen Parameter sind zwar auch leistungsbeeinflussend, können aber vom Benchmark nicht variiert werden. Es besteht aus Sicht des Benchmarks also keine Möglichkeit der Vermessung des Leistungseinflusses und auch keine Möglichkeit der Optimierung. Im Folgenden werden Parameter, die nicht durch die Applikation beeinflussbar sind, als Teil des zu vermessenden Systems verstanden. Wenn der Benutzer eine Abhängigkeit der Leistung von einem dieser Parameter ermitteln möchte (bspw. verschiedene Dateisysteme miteinander vergleichen), muss er selbstständig den entsprechenden Parameter im System variieren (also ein weiteres Dateisystem installieren und konfigurieren) – so ein „neues" zu vermessendes System schaffen und den Benchmark erneut ablaufen lassen.

Offensichtlich sind von den gerade angegebenen Parametern nur ein kleiner Teil durch die Applikation manipulierbar. Naturgemäß sind dies die Lastparameter 1 und 2, da die Last von der Applikation erzeugt wird. Von den genannten Systemparametern sind nur Parameter variierbar, die I/O-Bibliotheken dem Anwendungsprogramm zur Bibliotheks-Konfiguration zur Verfügung stellen (3). Letztere sind, wie beschrieben, spezi-

fisch für einzelne Implementierungen von I/O-Bibliotheken und deshalb sehr divergierend. Eine einheitliche Modellierung dieser unterschiedlichen Parameter ist nur bis zu einem gewissen Grad möglich und wird der Idee dieser Konfigurationsparameter auch nicht gerecht. Es ist gerade die Aufgabe dieser Parameter implementierungsspezifische Einstellungen, die in dieser Form in anderen Implementierungen nicht vorkommen, zu ermöglichen. Die genannten Parameter werden deshalb in dieser Arbeit ähnlich wie alle anderen Systemparameter behandelt und nicht als Teil der durch eine Applikation variierbaren Lastparameter behandelt. In der Praxis ist dies keine große Einschränkung darstellen, da existierende Applikationen die Bibliothekskonfigurationen nur selten verändern. Applikationsentwickler würden sich sonst auf einzelne Bibliotheksimplementierungen festlegen.

Es verbleiben die folgenden Parameter aus Tabelle 3.2, die durch eine Applikation variierbar sind und damit als Grundlage zur Beschreibung der I/O-Last dieser Applikation eingesetzt werden können:

1. Anordnung, Frequenz und Art der I/O-Anforderungen
2. Anzahl der Prozesse

Die beiden Parameter werden im Folgenden detaillierter betrachtet und bilden die Grundlage der Erstellung einer genauen und skalierbaren I/O-Lastbeschreibung.

3.3.2 Merkmale von I/O-Anforderungen

Die Anordnung, Frequenz und Art von I/O-Anforderungen bestimmen ganz maßgeblich die Last einer Applikation auf das I/O-System. Im Folgenden wird untersucht, wie die so erzeugte Last genau beschrieben und quantifiziert werden kann. Dazu ist eine Analyse des Aufbaus und der Möglichkeiten von I/O-Anforderungen, auch I/O-Requests genannt, notwendig. In der Literatur wird ein I/O-Request wie folgt definiert:

> *I/O-Request: Die Anforderung einer Anwendung, eine bestimmte Menge an Daten zu lesen oder zu schreiben. Im Kontext realer oder virtueller Disks spezifizieren I/O-Anforderungen den Transfer einer Anzahl an Datenblöcken zwischen aufeinander folgenden Disk-Block-Adressen und aufeinander folgenden Speicherstellen [78].*

Eigenschaften eines I/O-Requests

Bereits aus dieser Definition sind eine Reihe von Eigenschaften ableitbar, die einen I/O-Request kennzeichnen. Jeder I/O-Request wird durch einen Prozess initiiert, der eine Startadresse im Hauptspeicher, eine Startadresse auf dem Sekundärspeicher, die Datengröße, sowie deren Richtung (lesend oder schreibend) spezifizieren muss. Ist die Richtung lesend, werden Daten von der Sekundärspeicheradresse zur Hauptspeicheradresse transferiert, sonst umgekehrt. Der I/O-Request in dieser Definition entspricht einer Anforderung, wie sie direkt vom Sekundärspeicher verarbeitet wird. Höhere Ebenen des I/O-Subsystems fügen dieser Anforderung weitere Eigenschaften hinzu, beispielsweise, ob eine I/O-Anforderung den Prozess während der Bearbeitung blockieren

soll oder nicht. Diese weiteren Eigenschaften werden im Einzelnen in den Abschnitten 3.3.3 und 3.3.4 erläutert.

Definition 3.1: I/O-Request

Ein I/O-Request r ist ein 4-Tupel bestehend aus den 4 Parametern Hauptspeicherstartadresse (m), Sekundärspeicherstartadresse (s), Länge der Daten (n) und Richtung ($d \in \{r, w\}$):

$$r = (m, s, n, d) \text{ mit } m, s \in \mathbb{N} \wedge n \in \mathbb{N}^+ \wedge d \in \{r, w\}$$

Die Richtungen bedeuten dabei:

- r – lesende Anforderung (Sekundärspeicher → Hauptspeicher)
- w – schreibende Anforderung (Hauptspeicher → Sekundärspeicher)

Die Menge aller I/O-Requests als karthesisches Produkt über den Wertebereichen der Elemente eines Requests werde $\mathcal{R}_{IO}$ genannt:

$$\mathcal{R}_{IO} = \mathbb{N} \times \mathbb{N} \times \mathbb{N}^+ \times \{r, w\}$$

Für alle I/O-Requests $r_e = (m_e, s_e, n_e, d_e)$; $r_e \in \mathcal{R}_{IO}$ werden die folgenden Zugriffsfunktionen definiert:

$$\begin{aligned} m &: \mathcal{R}_{IO} \to \mathbb{N} \quad \text{mit } m(r_e) = m_e \\ s &: \mathcal{R}_{IO} \to \mathbb{N} \quad \text{mit } s(r_e) = s_e \\ n &: \mathcal{R}_{IO} \to \mathbb{N}^+ \quad \text{mit } n(r_e) = n_e \\ d &: \mathcal{R}_{IO} \to \{r, w\} \quad \text{mit } d(r_e) = d_e \end{aligned}$$

I/O-Last während eines Prozesslaufes

Da während eines Prozesslaufes zahlreiche I/O-Requests in einer bestimmten Reihenfolge durchgeführt werden, ist die I/O-Last während eines Prozesslaufes eine zeitlich angeordnete Folge von I/O-Requests.

Definition 3.2: I/O-Last eines Prozesslaufes

Die Folge aller I/O-Anforderungen eines Prozesslaufs wird als I/O-Last des Prozesslaufs bezeichnet und mit p abgekürzt:

$$p = (r_0, r_1, r_2, \ldots, r_n) \text{ mit } r_i \in \mathcal{R}_{IO} \wedge n \in \mathbb{N} \wedge i \leqslant n$$

Die I/O-Anforderungen werden entsprechend ihrer Auftrittsreihenfolge während des Prozesslaufs indiziert. $|p| = n + 1$ gibt die Anzahl aller I/O-Anforderungen der Last p an, wobei eine Last mindestens einen I/O-Request enthalten muss.
Es werde die Zugriffsfunktion $r_i(p)$ auf eine Last p definiert, die den I/O-Request mit dem Index i der Last p zurück gibt.

I/O-Profil

Die I/O-Last p beschreibt das gesamte Prozess-I/O-Verhalten eines Prozesslaufes. Im Folgenden wird angenommen, dass unterschiedliche Prozessläufe des gleichen Programms ein näherungsweise gleiches I/O-Verhalten aufweisen. In diesem Fall kann die I/O-Last p als Stellvertreter für das I/O-Verhalten des Programmes dienen. Sie wird dann als I/O-Profil bezeichnet und ist formal identisch zur I/O-Last definiert. Werden I/O-Profile aller Programme erstellt, die an einer Applikation beteiligt sind, kann die Zusammenfassung dieser I/O-Profile auch als Applikations-I/O-Profil bezeichnet werden. Dennoch ist leicht erkennbar, dass ein derartiges I/O-Profil unter Umständen sehr umfangreich werden kann. Da jeder I/O-Request durch die Parameter Quelle, Ziel, Größe und Richtung spezifiziert werden muss, kann das Profil bereits bei kleinen Applikationen schnell einen Umfang von mehreren tausend Werten überschreiten.

I/O-Profil bei IOzone

Ein I/O-Profil im beschriebenen Sinn stellt ein Abbild für das I/O-Verhalten einer Applikation auf Ebene der I/O-Anforderungen dar. Der Benchmark *IOzone* kann ein I/O-Profil in ähnlicher Form als Lastbeschreibung verwenden, um das I/O-Verhalten eines Prozesses anhand des Profils nachzubilden und die Leistung zu vermessen. Das I/O-Profil einer zu vermessenden Applikation wird bei *IOzone* in zwei Teilprofile aufgeteilt, die jeweils nur aus den lesenden bzw. schreibenden I/O-Requests bestehen. Jedes der Teilprofile speichert pro Anforderung die Anforderungsgröße, die Sekundärspeicherstartadresse und zusätzlich eine Wartezeit, die nach der Ausführung einer Anforderung gewartet wird, um die zeitliche Funktion abzubilden. Mit diesem Vorgehen verliert der *IOzone* alle Informationen über die Verknüpfung von lesenden und schreibenden Zugriffen, was durchaus eine erhebliche Einschränkung ist [64]. Die Hauptspeicherstartadresse als weiterer Parameter eines I/O-Requests wird ebenfalls nicht vermerkt. Der Benchmark erzeugt zur Laufzeit Speicherbereiche, deren Adressen als Hauptspeicherstartadressen für die Anforderungen verwendet werden. Diese Vorgehensweise ist bei allen aktuellen I/O-Benchmarks üblich. Es wird davon ausgegangen, dass die Zugriffszeit auf den Hauptspeicher eines Systems von dessen Speicherposition unabhängig ist. Tatsächlich ist dies zwar bei einer großen Klasse von Rechensystemen der Fall, nicht jedoch bei allen.

Uniform Memory Access

Non-Uniform-Memory-Access

Bei UMA-Architekturen (Uniform Memory Access) ist die Geschwindigkeit zum Zugriff auf den Hauptspeicher von der Speicherposition unabhängig, da alle Prozessoren des Systems den gleichen Hauptspeicher verwenden und damit die gleiche „Nähe" zum Speicher haben. Die Hauptspeicheradresse als Teil eines I/O-Requests kann bei derartigen Systemen ohne Informationsverlust ignoriert werden, wie alle vorhandenen I/O-Benchmarks es tun. Bei NUMA-Architekturen (Non-Uniform-Memory-Access) hingegen sind nicht alle Speicherpositionen des Hauptspeichers mit der gleichen Geschwindigkeit zugreifbar, da einige Speicherbereiche zu anderen Prozessoren gehören (entfernter Speicher) und die Speicheranfrage über andere Verbindungsstrukturen zu erfolgen hat, als der lokale Hauptspeicher [79]. Im Einzelnen sind die Einflüsse der Speicherarchitektur auf die I/O-Leistung noch zu untersuchen, wobei sie auch im Rahmen dieser Arbeit aus folgenden Gründen ignoriert werden:

- Die Wahl, ob angeforderter Speicher eines Prozesses lokal oder entfernt realisiert wird, liegt beim Betriebssystem, welches so die Leistung des Prozesses zur Laufzeit maximieren kann. Im Idealfall funktioniert die Auswahl des Speichers also völlig transparent für den Prozess und ohne dessen Einfluss, aber trotzdem in einer Form, dass eine maximale Leistung erreicht wird. Die Festlegung bestimmter Speicher im Lastprofil würde also die wirklich durch die Applikation in einem bestimmten Prozesslauf erzeugte Last eher verfälschen als realistisch wiedergeben.

- Die Speicherzugriffszeit auf eine aktuelle Festplatte ist mit 5 bis 10 ms ca. 1.000.000 mal so hoch wie die Zugriffszeit auf einen lokalen Hauptspeicher (Zugriffszeit zwischen 5 und 30 ns). Bei einem Datentransfer von Hauptspeicher zum Sekundärspeicher ist der zeitliche Anteil des Hauptspeicherzugriffs, selbst wenn mehrere Zugriffe aufgrund verschiedener Kopiervorgänge innerhalb des Betriebssystems durchgeführt werden müssen, noch immer der Bruchteil von einem Promille (1 / 1.000.000 = 0,001 Promille). In [79] wird angegeben, dass der Zugriff auf entfernten Hauptspeicher der NUMA-Architektur einer SGI Altix-Hardware maximal um den Faktor 3.5 länger dauert, als dies bei lokalem Speicher der Fall ist. Selbst in einem solchen Fall liegt der Zeitanteil des Zugriffs auf (entfernten) Hauptspeicher an der Gesamtzugriffszeit bei einem I/O-Transfer noch weit unterhalb eines Promille. In beiden Fällen ist der Anteil des Hauptspeicherzugriffs an der Gesamtzugriffszeit sehr gering.

Die Hauptspeicheradresse wird in dieser Arbeit, wie bei aktuellen I/O-Benchmarks üblich, aus den genannten Gründen ignoriert. Eine genauere Untersuchung des Einflusses der Speicherarchitektur auf die I/O-Leistung sollte dennoch Thema von Untersuchungen sein, die dieser Arbeit folgen.

3.3.3 Lastparameter nicht-verteilter Applikationen

Bereits hervorgehoben wurde das Problem, dass ein I/O-Profil schnell sehr umfangreich wird. Eine manuelle Erstellung eines Profils zur Nachbildung einer Applikationslast gestaltet sich allein wegen des Umfangs als sehr schwierig. Es muss aus diesem Grund eine Zusammenfassung der Werte des I/O-Profils in einer Form erfolgen, in der trotz des damit verbundenen Informationsverlusts eine hinreichend genaue Rekonstruktion des I/O-Verhaltens einer Applikation möglich ist. In [80] werden von P. Chen und D. Patterson fünf Parameter vorgeschlagen, die durch Zusammenfassung aller I/O-Anforderungen einer Applikation ermittelt werden. Vier dieser Parameter werden direkt aus den Merkmalen der I/O-Requests berechnet, während der fünfte Parameter der Anzahl von Prozessen entspricht, die zeitgleich I/O durchführen (er entspricht also dem Parameter 2 aus Tabelle 3.2). Die fünf von Chen und Patterson definierten Parameter dienen auch dieser Arbeit als Grundlage und werden deshalb im Anschluss beschrieben:

I/O-Parameter nach Chen/ Patterson

uniqueBytes – die Gesamtmenge gelesener oder geschriebener Bytes einer Folge von I/O-Requests

sizeMean – die durchschnittliche Größe einer I/O-Anforderung

readFrac – der Anteil gelesener Bytes an der Gesamtmenge transferierter Bytes

seqFrac – der Anteil von I/O-Anforderungen, die sequentiell auf den vorherigen Request folgen

processNum – die Anzahl zeitgleich arbeitender Prozesse, die I/O-Zugriffe durchführen

Diese Parameter der I/O-Last bilden wichtige Eigenschaften ab, die für die I/O-Leistung einer Applikation von Bedeutung sind. Im Folgenden werden diese Lastparameter formal definiert, da die Autoren in ihren Arbeiten nur verbale Beschreibungen angeben, die an einigen Stellen nicht in aller Klarheit erläutert sind und zu unterschiedlichen Auslegungen führen können. Neben der formalen Definition wird zu jedem Parameter der Wertebereich angegeben.

uniqueBytes

Der Parameter *uniqueBytes* gibt die Gesamtmenge aller transferierten Bytes einer Folge von I/O-Anforderungen p eines Prozesses an, kann also formal wie in Definition 3.3 aus den I/O-Anforderungen (s. Definition 3.1) berechnet werden. Neben der formalen Berechnung des Parameters *uniqueBytes* ist der Definitionsbereich angegeben, also die Menge von Werten, in die die Funktion zur Berechnung von *uniqueBytes* abbildet:

Definition 3.3: uniqueBytes

Der Parameter *uniqueBytes* wird aus der Größe der I/O-Anforderungen einer I/O-Last p nach Definition 3.2 wie folgt ermittelt:

$$\text{uniqueBytes} \quad : \quad \nu_{\mathtt{unique}}(p) = \sum_{i=0}^{|p|-1} n(r_i(p))$$

$$\text{Wertebereich} \quad : \quad D_{\mathtt{unique}} = \{x \,:\, x \in \mathbb{N}^+\}$$

sizeMean

Der Parameter *sizeMean* gibt die durchschnittliche Größe eines I/O-Requests an und wird damit als Quotient aus der Gesamtmenge an transferierten Daten *uniqueBytes* und der Anzahl der I/O-Requests berechnet:

Definition 3.4: sizeMean

Der Parameter *sizeMean* wird aus einer I/O-Last p nach Definition 3.2 wie folgt ermittelt:

$$\begin{aligned} \text{sizeMean} &: \nu_{size}(p) = \frac{\sum_{i=0}^{|p|-1} n(r_i(p))}{|p|} = \frac{\nu_{unique}(p)}{|p|} \\ \text{Wertebereich} &: D_{size} = \{x : x \in \mathbb{Q}^+\} \end{aligned}$$

Der Anteil gelesener Bytes wird durch *readFrac* spezifiziert und als Quotient aus dem Anteil der insgesamt gelesenen Daten und der gesamten Datenmenge *uniqueBytes* berechnet. Dazu wird zuerst die Menge gelesener Bytes $\text{read}(r)$ eines I/O-Requests bestimmt: *readFrac*

Definition 3.5: readFrac

Für jedes $r \in \mathcal{R}_{IO}$ sei die Funktion $\text{read}(r)$ wie folgt definiert:

$$\text{read}(r) = \begin{cases} 0 & \text{für } d(r) = w \\ n(r) & \text{sonst} \end{cases}$$

Mit dieser Funktion wird *readFrac* aus einer I/O-Last p nach Definition 3.2 wie folgt ermittelt:

$$\begin{aligned} \text{readFrac} &: \nu_{read}(p) = \frac{\sum_{i=0}^{|p|-1} \text{read}(r_i(p))}{\sum_{i=0}^{|p|-1} n(r_i(p))} = \frac{\sum_{i=0}^{|p|-1} \text{read}(r_i(p))}{\nu_{unique}(p)} \\ \text{Wertebereich} &: D_{read} = \{x : x \in \mathbb{Q}; 0 \leqslant x \leqslant 1\} \end{aligned}$$

Der Parameter *seqFrac* spezifiziert den Anteil der I/O-Aufrufe, die sequentiell einem vorhergehenden I/O-Request folgen. *seqFrac*

Definition 3.6: Sequentialität

Ein I/O-Request r_i heißt dann sequentiell folgend in der Last p nach Definition 3.2, wenn der r_i ausführende Prozess mit der Last p unmittelbar vor der Ausführung von r_i nicht explizit die Zielposition des I/O-Requests verändert hat.
In einem solchen Fall ist das Ergebnis der Funktion $seq(i,p)$ eins, in allen anderen Fällen, gibt die Funktion null zurück:

$$seq(i,p) = \begin{cases} 1 & \text{falls } r_i \text{ sequentiell folgend heißt} \\ 0 & \text{sonst} \end{cases}$$

Sequentialität bei POSIX-I/O

Die eingeführte Definition der Sequentialität von Requests kann schwer detaillierter ausgeführt werden, da die formale Beschreibung je nach der verwendeten I/O-Schnittstelle anders definiert werden muss. Beispielhaft wird Sequentialität im Folgenden anhand der POSIX-I/O-Schnittstelle definiert. Bei ihr ist der I/O-Request $r_i(p)$ in der Last p dann sequentiell auf den vorherigen Request folgend, wenn die Sekundärspeicheradresse $s(r_i(p))$ der Byteposition entspricht, die der letzten Byteposition der vorherigen I/O-Anforderung $r_{i-1}(p)$ folgt. Die Funktion $seq(i,p)$ lautet in diesem Fall wie folgt:

$$seq(i,p) = \begin{cases} 1 & \text{falls } 0 < i \leqslant |p| - 1 \wedge s(r_{i-1}(p)) + n(r_{i-1}(p)) = s(r_i(p)) \\ 0 & \text{sonst} \end{cases}$$

Definition 3.7: seqFrac

Der Anteil der sequentiellen Aufrufe *seqFrac* einer Last p nach Definition 3.2 wird mittels der Funktion $seq(i,p)$ berechnet:

$$\begin{aligned} \text{seqFrac} \quad &: \quad \nu_{seq}(p) = \frac{\sum_{i=0}^{|p|-1} seq(i,p)}{|p|} \\ \text{Wertebereich} \quad &: \quad D_{seq} = \{x \,:\, x \in \mathbb{Q}; 0 \leqslant x \leqslant 1\} \end{aligned}$$

Sequentialität bei MPI-IO

Bei Verwendung von MPI-IO ist die formale Definition der Sequentialität schwieriger, da die Zieladresse eines I/O-Requests auch ohne explizite Veränderung des Dateizeigers nicht direkt dem Ende des vorherigen I/O-Transfers folgen muss. Dies wird zum Beispiel bei der Verwendung von sogenannten gemeinsamen Dateizeigern (*shared file pointer*) deutlich. Ein gemeinsamer Dateizeiger gibt die aktuelle, für alle Prozesse einer Applikation gültige Schreib-/Leseposition innerhalb einer Datei an. Schreibt in so einem Fall ein Prozess der Applikation Daten, wird der Dateizeiger für alle Prozesse auf die neue Position verschoben. Die Position des I/O-Requests eines Prozesses folgt in einem solchen Fall nicht der Position des letzten Requests des Prozesses, ist aber

dennoch auf dem Datenspeicher sequentiell folgend auf den vorherigen Request der Applikation.

Deutlicher wird das Problem der Sequentialität bei MPI-IO, wenn Sichten auf Dateien verwendet werden. MPI-IO unterstützt das Konzept unterschiedlicher Sichten mehrerer Prozesse auf die gleiche Datei. Dazu definiert jeder Prozess die für ihn zugreifbaren Bytes innerhalb einer für alle Prozesse einer Applikation gleich großen Dateieinheit. Die Definition erfolgt mittels einer Maske, die *file type* genannt wird. In einer Dateieinheit darf jedes Datenbyte nur durch maximal einen Prozess verwendet werden. Abbildung 3.4 zeigt diesen Zusammenhang anhand von drei Prozessen. Es ist zu sehen, dass jeder Prozess innerhalb der Datei verteilt Daten speichert, die sich für ihn aber als ein zusammenhängender Datenstrom darstellen, da er keine Möglichkeit hat, auf die Daten der anderen Prozesse zuzugreifen. Sequentielle Zugriffe aus Sicht des Prozesses

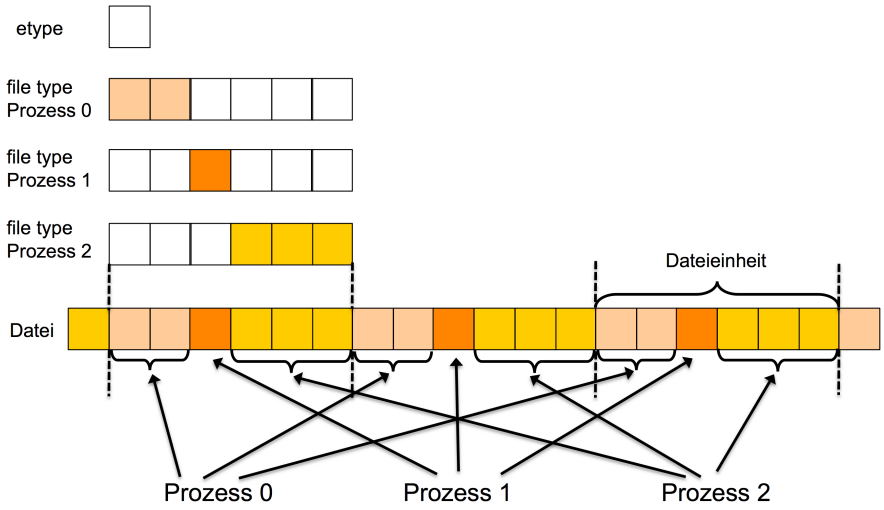

Abbildung 3.4: Unterschiedliche Sicht von 3 Prozessen auf eine Datei bei MPI-IO

werden also verteilt auf dem Datenspeicher durchgeführt, so dass eine Sequentialität aus Sicht des Datenspeichers nicht gegeben ist. Die oben angegebene formale Definition für Sequentialität, die auf den Zieladressen des Datenspeichers basiert, kann hier nicht funktionieren. Da auch das MPI-System versuchen wird, die Zugriffe aller Prozesse so zu optimieren und zu vereinigen, dass letztlich auf dem Datenspeicher eine sequentielle Speicherung vorgenommen wird, werden auch derartige Zugriffe, die aus Sicht des Prozesses sequentiell durchgeführt werden, als sequentiell folgend interpretiert. Die angegebene textuelle Definition der Sequentialität gilt also bei der POSIX-I/O-Schnittstelle ebenso wie beim MPI-IO-Interface.

Neben den vorgestellten vier Lastparametern definieren Chen und Patterson einen weiteren I/O-Parameter, der ebenfalls durch eine Benchmarkapplikation variiert werden kann.

processNum bei Chen/ Patterson

processNum ist nicht aus den Charakteristiken der I/O-Anforderungen einer Applikation berechenbar, gibt laut den Autoren aber die Anzahl der Prozesse an, die parallel I/O auf dem Rechensystem durchführen. Dieser Parameter entspricht damit dem Parameter 2 aus Tabelle 3.2. In [81] wird definiert, dass alle durch *processNum* angegebenen Prozesse die gleiche I/O-Last produzieren, die mittels der vier beschriebenen Lastparameter *uniqueBytes*, *sizeMean*, *readFrac* und *seqFrac* definiert wird. In den Ausführungen wird nicht deutlich, warum die Parallelität innerhalb einer Applikation auf nur einen Parameter – die Prozessanzahl – reduziert werden kann. Es ist offensichtlich, dass in einer Applikation alle Prozesse unterschiedliche I/O-Charakteristiken haben können und oftmals auch haben. Demnach besitzen die I/O-Charakteristiken der Einzelprozesse unterschiedliche I/O-Parameter-Werte. Chen/Patterson ignorieren diesen Fakt, zeigen aber anhand von Messungen, dass ihr Lastmodell in den dargestellten Beispielen genau genug ist, die I/O-Last einer Applikation zu beschreiben. Zur I/O-Lastbeschreibung einer parallelen Applikation wird im Rahmen dieser Arbeit die Bezeichnung des Parameters *processNum* von Chen und Patterson übernommen, wenngleich seine Semantik abgeändert wird. Es wird später gezeigt, dass mit der neuen Semantik eine sehr gute Darstellung der typischen I/O-Charakteristiken paralleler Applikationen ermöglicht wird.

Die bisher genannten fünf Parameter reichen laut Chen und Patterson aus, um I/O-Workloads für nicht-verteilte Rechensysteme genau zu beschreiben. In [62] wird dies anhand von Messungen an einer SPARCStation 1+ und einer DECstation 5000/200 nachgewiesen. Für verteilte Rechensysteme, die einen parallelen Zugriff auf Massenspeicher durchführen, sind weitere Anforderungen notwendig. Die Prozesse verwenden besondere I/O-Zugriffsmethoden paralleler I/O-Schnittstellen, die mit den genannten fünf Parametern nicht vollständig beschrieben werden können.

3.3.4 Lastparameter verteilter Applikationen

Parallele Sekundärspeichersysteme bedienen besondere Anforderungen paralleler Applikationen durch spezielle Eigenschaften der I/O-Schnittstelle, die durch Chen/Patterson nicht betrachtet wurden. Neben der bereits erwähnten Änderung der Semantik des Parameters *processNum* werden in dieser Arbeit drei neue Parameter vorgeschlagen, die den spezifischen Eigenschaften der parallelen MPI-IO-Schnittstelle Rechnung tragen, und entsprechend eine genaue Abbildung dieser Schnittstelle ermöglichen [82].

Änderung der Semantik für *processNum*

Während *processNum* in [81] als die Anzahl von Prozessen einer lokalen Applikation verstanden wird, die auf einen Sekundärspeicher zugreifen, d. h. die auf einem Rechenknoten laufenden Prozesse, wird das Verständnis des Begriffes in dieser Arbeit derart geändert, dass darunter die Anzahl der Prozesse *einer parallelen Applikation*

verstanden wird, die auf einen gemeinsamen Datenspeicher zugreifen. Diese Prozesse können deshalb durchaus auf unterschiedlichen Rechenknoten eines Rechensystems ausgeführt werden. Einzige Voraussetzungen sind, dass sie zu einer Applikation gehören und I/O-Operationen auf einem gemeinsamen verteilten Speicher, der beispielsweise per NFS zur Verfügung gestellt wird, durchführen. Der Parameter *processNum* kann nicht formal aus den I/O-Requests der Applikation ermittelt werden.

processNum

Definition 3.8: processNum

Der Parameter *processNum* gibt die Anzahl von Prozessen einer Applikation an, die parallele I/O-Requests durchführen.

$$\text{Wertebereich} \quad : \quad \mathrm{D}_{\mathrm{process}} = \{\mathrm{x} \,:\, \mathrm{x} \in \mathbb{N}^{+}\}$$

Häufig entspricht der Wert der Gesamtanzahl von Prozessen, aus denen eine Applikation besteht – dies muss aber nicht so sein, wie im Verlauf der Arbeit gezeigt wird.

Weitere Parameter

Um ein I/O-Lastmodell zu entwickeln, das alle Möglichkeiten einer I/O-Schnittstelle modellieren kann, ist eine genaue Untersuchung der I/O-Schnittstelle erforderlich. Die ersten vier der vorgestellten Lastparameter entstanden durch die Analyse der möglichen Parameter eines I/O-Requests, wie er durch die I/O-Schicht des Betriebssystems generiert und vom Sekundärspeicher bearbeitet wird. Der Parameter *processNum* entstand durch weitere Analyse des I/O-Subsystems.

MPI ist die wichtigste Schnittstelle zur Implementierung paralleler Applikationen. MPI-IO, als Teil der MPI-Spezifikation, stellt die I/O-Schnittstelle von MPI dar und wird im Folgenden genau untersucht, um weitere Parameter zu identifizieren, die die I/O-Lastcharakteristik einer MPI-Applikation beeinflussen. Da die MPI-IO-Schnittstelle auf Applikationsebene Möglichkeiten zur Verfügung stellt, die bislang noch nicht erfasst wurden, muss eine Analyse der I/O-Datenzugriffsmöglichkeiten von MPI-IO erfolgen.

MPI-IO

In [83] wird eine Übersicht präsentiert, die sämtliche I/O-Datenzugriffsfunktionen von MPI-IO tabellarisch darstellt und die als Grundlage zur Definition von weiteren Parametern verwendet wird. Zum besseren Verständnis wird diese Übersicht in Tabelle 3.3 wiedergegeben.

Datenzugriffsfunktionen der MPI-IO-Schnittstelle

Es werden in der Tabelle sämtliche MPI-IO-Datenzugriffsfunktionen bezüglich ihrer Eigenschaften klassifiziert. Die dargestellten Eigenschaften sind:

- die Positionierung – Verwendet jeder Prozess einen eigenen Dateizeiger (*individual file pointer*), wird ein gemeinsamer Dateizeiger für alle Prozesse eingesetzt

positioning	**synchronism**	**coordination**	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking & split collective	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
individual file pointers	*blocking*	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking & split collective	MPI_File_iread MPI_File_iwrite	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
shared file pointer	*blocking*	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking & split collective	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Tabelle 3.3: Datenzugriffsfunktionen von MPI-IO im Überblick [83]

(*shared file pointer*), oder wird die Zielposition explizit angegeben (*explicit offset*)?

- die Synchronizität – Ist die I/O-Anforderung blockierend (*blocking*) oder asynchron (*nonblocking*)?

- und die Koordination – Werden mehrere I/O-Requests verschiedener Prozesse zusammengefasst und so optimiert durch das MPI-IO-System ausgeführt (*collective*) oder sind die Anforderungen unabhängig voneinander (*noncollective*)?

Unabhängigkeit der Parameter

Tabelle 3.3 zeigt in dieser Form, dass die dargestellten drei Eigenschaften unabhängig voneinander verwendbar sind, da es für jede beliebige Kombination der drei Eigenschaften jeweils Zugriffsfunktionen gibt. Da für jede dieser möglichen Kombinationen jeweils lesende und schreibende Zugriffsfunktionen existieren, ist der Parameter *readFrac* also ebenfalls unabhängig von den drei Parametern einsetzbar. Ebenso sind die bereits eingeführten Parameter *uniqueBytes*, *sizeMean* und *processNum* voneinander und von den drei zusätzlichen Eigenschaften von MPI-IO unabhängige Eigenschaften, da jede I/O-Anforderung durch eine beliebige Anzahl von Prozessen mit jeder Größe durchgeführt werden kann. Der Parameter *seqFrac* stellt eine Ausnahme dar. Er ist zwar unabhängig von den Eigenschaften *Synchronizität* und *Koordination*, da

beide Formen von I/O-Requests jeweils sequentiell oder an zufälligen Dateipositionen durchgeführt werden können – er ist jedoch nicht unabhängig von der Eigenschaft *Positionierung*, da die Verwendung expliziter Offsets zu einem nicht sequentiellen Zugriffsmuster führt. Werden hingegen Zugriffe mit expliziten Offsets nicht als eigene Zugriffsart der Eigenschaft *Positionierung*, sondern als zufällige Zugriffe verstanden, gibt es für die Eigenschaft der Positionierung nur noch die beiden möglichen Dateizeigerarten (*eigener Dateizeiger* und *gemeinsamer Dateizeiger*). Zugriffe beider Dateizeigerarten können jeweils an zufälligen Dateipositionen als auch sequentiell ausgeführt werden und sind damit unabhängig mit dem Parameter *seqFrac* kombinierbar. Da die Dateizeigerarten weiterhin, wie Tabelle 3.3 deutlich macht, mit den anderen beiden MPI-IO-Eigenschaften sowie allen bisher eingeführten Parametern kombinierbar sind, wird für die Unterscheidung zwischen den beiden Dateizeigerarten der weitere Parameter *sharedFrac* für das I/O-Lastmodell eingeführt. *sharedFrac*

Definition 3.9: sharedFrac

SharedFrac gibt den Anteil der Zugriffe an, der mit einem gemeinsamen Dateizeiger durchgeführt wird. Es hat den folgenden Wertebereich:

$$\text{Wertebereich} \quad : \quad D_{\texttt{shared}} = \{x \,:\, x \in \mathbb{Q}; 0 \leqslant x \leqslant 1\}$$

Neben *sharedFrac* werden für die bereits kurz eingeführte Synchronizität der Parameter *syncFrac* und für die Koordination der Parameter *collFrac* eingeführt, deren Bedeutungen im Folgenden näher erläutert werden.

Der Parameter *syncFrac* gibt den Anteil der I/O-Zugriffe einer Applikation an, die den aufrufenden Prozess der Applikation auf das Ende des Requests warten lassen, also den Anteil sogenannter blockierender Zugriffe. *SyncFrac* ist ein leistungsentscheidender Last-Parameter, da moderne Applikationen mehr und mehr versuchen ihre Applikationsleistung durch die Verwendung von asynchronem I/O (also nicht blockierenden Zugriffen) zu erhöhen. Nur so können parallel zur Ein- und Ausgabe andere Aufgaben, wie zum Beispiel die Berechnung neuer Ergebnisse durchgeführt werden. *syncFrac*

Definition 3.10: syncFrac

SyncFrac gibt den prozentualen Anteil der blockierenden Zugriffe an der Gesamtanzahl an I/O-Zugriffen an. Es hat den folgenden Wertebereich:

$$\text{Wertebereich} \quad : \quad D_{\texttt{sync}} = \{x \,:\, x \in \mathbb{Q}; 0 \leqslant x \leqslant 1\}$$

Der Parameter *collFrac* gibt den Anteil der I/O-Anforderungen an, die kollektiv stattfinden. Kollektive Zugriffe werden durch das I/O-System gesammelt und werden erst dann auf den Sekundärspeicher geschrieben, wenn der letzte kollektive I/O-Zugriff stattgefunden hat. Durch das Sammeln der I/O-Anforderungen ist ein hohes Optimierungspotential durch das I/O-System gegeben. Bei Verwendung eines lokalen Dateisystems kann die Anzahl der Neupositionierungen des Schreib-Lese-Kopfes der Fest- *collFrac*

platte reduziert werden, während bei parallelen Systemen die Anzahl der Netzwerkübertragungen verringert wird.

Definition 3.11: collFrac

CollFrac gibt den prozentualen Anteil der kollektiven Zugriffe an der Gesamtanzahl an I/O-Zugriffen an. Es hat es den folgenden Wertebereich:

$$\text{Wertebereich} \quad : \quad D_{\text{coll}} = \{x \,:\, x \in \mathbb{Q}; 0 \leqslant x \leqslant 1\}$$

Noncontiguous I/O

Wissenschaftliche Applikationen haben besondere Anforderungen beim Zugriff auf den Massenspeicher. Einerseits verarbeiten sie im Vergleich zu klassischen Workstation-Anwendungen sehr große Datenmengen. Andererseits verwenden sie Zugriffsmuster, die für Workstations ungewöhnlich sind. In [84] ergibt die Analyse des Workloads verschiedener paralleler Applikationen, dass der größte Teil der Zugriffe *noncontiguous* ist, d. h. die Zugriffe folgen nicht zusammenhängend auf dem Massenspeicher. Eine genauere Untersuchung zeigt, dass sie trotz des nicht zusammenhängenden Charakters vorwiegend starke Regularitäten aufweisen.

noncontiguous

In [84] wird eine Untersuchung präsentiert, die zeigt, dass auf einem Parallelrechnersystem von insgesamt 17.312 untersuchten Dateien 16.843 Dateien, also 97.3 % überlappende (*strided*) Strukturen aufwiesen, die entstehen, wenn jeder Prozess einer Applikation regulär auf eine gleich große Datenmenge zugreift, wobei diese einzelnen Zugriffe jeweils in gleichem Abstand voneinander auf dem Sekundärspeicher angeordnet sind. Dies ist ein Zugriffsmuster, wie es in Abbildung 3.4 auf Seite 67 vorgestellt wurde. Dateisichten, wie sie von MPI-IO unterstützt werden, sind ein Hilfsmittel, um noncontiguous I/O zu realisieren. Ein typischer Anwendungsfall für ein derartiges Zugriffsmuster ist eine zeilenweise geschriebene Matrix, bei der jede Spalte von einem Prozess bearbeitet wird. Derartige Zugriffsmuster treten häufig auch ineinander verschachtelt auf, was auf höherdimensionale Felder hinweist. Auf einem zweiten untersuchten Parallelrechnersystem hatten zwar nur ca. 94 % der Dateien überlappende Strukturen, was aber dennoch die Wichtigkeit dieses Zugriffsmusters bei Applikationen aus dem Hochleistungsrechnen unterstreicht. Derartige Lasten unterscheiden sich signifikant von klassischen Workstation-Lasten, da sie zwar ein reguläres Muster aufweisen, allerdings mit herkömmlichen I/O-Schnittstellen (wie POSIX-I/O) nur durch ständiges Neupositionieren des Dateizeigers in den einzelnen Prozessen und damit als nicht zusammenhängende, zufällige Zugriffe realisiert werden können [85].

strided

Auch andere Untersuchungen kommen zu dem Ergebnis, dass Zugriffe von Prozessen, die sich gegenseitig überlappen, ein sehr häufiges Lastszenario in parallelen Applikationen sind [86, 87, 85]. In [87] weisen die Autoren bei drei von fünf untersuchten I/O-Lasten paralleler Applikationen überlappende Zugriffe nach. Die anderen beiden

Applikationen verwenden rein sequentielle Zugriffe, bei denen jeder Prozess einen zusammenhängenden Teil einer großen Datei oder aber jeweils eine eigene Datei verwendet, um darin zu schreiben.

Da *noncontiguous I/O* einen großen Leistungs-Einfluss hat und der besondere Fall der überlappenden I/O-Zugriffe bei parallelen Applikationen eine hohe Wichtigkeit besitzt, wird an dieser Stelle gesondert betrachtet, ob eine adäquate Abbildung dieser Zugriffsmuster mit dem vorgestellten I/O-Lastmodell möglich ist und welche Einschränkungen es diesbezüglich gibt.

Genaue Abbildung. MPI-IO als I/O-Schnittstelle für parallele Applikationen wurde so implementiert, dass es bei allen Datenzugriffsfunktionen *strided*-Zugriffe unterstützt, um Optimierungen bei den Zugriffen durchführen zu können. So können bspw. die einzelnen Zugriffe aller schreibenden Prozesse durch das MPI-IO-System im Hauptspeicher gesammelt werden, bis ein zusammenhängender Datenbereich entsteht. Dieser wird dann mit hoher Leistung als *contiguous I/O* auf den Datenspeicher geschrieben [84, 69, 30].

Überlappende Zugriffe werden bei MPI-IO mit den bereits eingeführten Dateisichten realisiert (vgl. Abbildung 3.4). Sie ermöglichen, dass sämtliche Zugriffe auf die Datei aus Sicht eines Prozesses wie sequentielle Zugriffe durchgeführt werden, wobei das MPI-IO-System die Daten auf die zum jeweiligen Prozess gehörenden Datenbereiche verteilt.

Für eine exakte Abbildung dieses Verhaltens mittels eines Lastmodells muss pro Prozess eine Maske definiert werden, die zeigt, welche Datenbereiche der Dateieinheit (die die für alle Prozesse gemeinsame übergeordnete Zugriffseinheit der Datei darstellt) von welchen Prozessen verwendet werden. Der als durchschnittliche Anfragegröße *sizeMean* definierte Parameter spiegelt dann die Größe der Dateieinheit wieder und nicht, wie bislang definiert, die durchschnittliche Größe der Daten, die ein Prozess verarbeitet. Pro Prozess käme bei einer solchen Realisierung ein Parameter im Lastmodell hinzu. Die Parameteranzahl des Lastmodells wäre also abhängig vom Wert des Parameters *processNum*. Da das Aussehen jeder Prozessmaske vom Wert des Parameters *sizeMean* abhängt, sind einige Werte für die neuen Parameter in Abhängigkeit von *sizeMean* nicht möglich, weshalb die Unabhängigkeit der Parameter teilweise verloren geht. Dennoch wäre ein solches Modell realisierbar, aber für den Nutzer schwer verständlich. Die nicht vorhandene Unabhängigkeit erschwert die Nutzung und die Darstellung der Maske bspw. als eine Bitmaske ist eine Inkonsistenz im Modell, das bislang alle Parameter als Zahlen definiert hat.

Annäherung. Eine erste Vereinfachung der vorgestellten Idee fügt pro Prozess einen Parameter hinzu, der den prozentualen Anteil der Bytes der Dateieinheit angibt, den der Prozess verwendet. Die Größe der Dateieinheit wird wie gerade beschrieben mit *sizeMean* dargestellt. Die genaue Anordnung der Bytes jedes Prozesses ist mit

einer solchen Vereinfachung nicht mehr rekonstruierbar. Das Modell wird leichter verständlich, hat aber nach wie vor das Problem, dass die Parameteranzahl vom Wert des Parameters *processNum* abhängig ist.

Eine weitere Vereinfachung kann erreicht werden, wenn definiert wird, dass im Lastmodell für alle Prozesse einer Applikation die Menge der Daten identisch ist, die ein Prozess innerhalb einer Dateieinheit mit der Größe *sizeMean* bearbeitet. Insbesondere bei dem wichtigsten Anwendungsfall der Matrizen, bei denen jedes Matrix-Element normalerweise den gleichen Typ besitzt, ist dies keine Einschränkung. In einem solchen Fall kann der Anteil, der von der Dateieinheit eines Prozesses zu bearbeitenden Daten als ein weiterer (neunter) Parameter definiert werden. Wenn dieser Wert nicht prozentual, sondern absolut als Anzahl von Bytes angegeben wird, kann er leicht auch größer als eine Dateieinheit werden. Dies hat einen weiteren wesentlichen Vorteil: Auch die Realisierung größerer (mehr als eine Dateieinheit umfassende) Bereiche, auf die durch einen Prozess sequentiell zugegriffen wird, ist möglich. Dieser Parameter wird im Folgenden *contMean* genannt und als neunter Parameter in das I/O-Lastmodell aufgenommen.

contMean

Mittels *contMean* kann *noncontiguous I/O* ebenso abgebildet werden, wie sequentielle Zugriffe einzelner Prozesse auf große unabhängige Dateiteile. In [87] werden derart sequentielle Zugriffe als zweite Art des Zugriffs beschrieben, die von parallelen Applikationen verwendet wird.

Es wird nun geklärt, wie *contMean* aus den MPI-IO-Anforderungen einer Applikation ermittelt wird. Für jede MPI-IO-Anforderung einer Applikation gilt, dass diese, neben der Größe der Dateieinheit, die Byteanzahl der vom Prozess ν in dieser Anforderung $\mathfrak{i}$ bearbeiteten Daten $\mathfrak{l}_\mathfrak{i}^\nu$ spezifiziert. $\mathfrak{l}_\mathfrak{i}^\nu$ ist ein zusätzlicher Parameter, der in der Definition 3.1 eines I/O-Requests nicht aufgeführt wird, da er MPI-IO-spezifisch ist.

Unter der Voraussetzung, dass alle Prozesse einer MPI-Applikation die gleiche Anzahl von MPI-IO-Anforderungen $|\mathfrak{p}|$ durchführen, kann nun *contMean* als Durchschnitt der Längen $\mathfrak{l}_\mathfrak{i}^\nu$ aller $|\mathfrak{p}|$ MPI-IO-Anforderungen aller ν_{proc} Prozesse der Applikation berechnet werden. Tatsächlich können mehrere Prozesse eine unterschiedliche Anzahl von I/O-Anforderungen ausführen, so dass die an dieser Stelle gemachte Voraussetzung eine starke Einschränkung zu sein scheint. In Abschnitt 3.3.9 wird auf dieses Problem näher eingegangen und gezeigt, dass die gemachte Voraussetzung für typische Applikationen aus dem Bereich des Hochleistungsrechnens keine Beschränkung darstellt.

Definition 3.12: contMean

ContMean gibt die durchschnittliche Anzahl an zusammenhängenden Bytes an, die von den I/O-Anforderungen eines Prozesses einer Applikation bearbeitet werden:

$$\text{contMean} \quad : \quad \nu_{\text{cont}} = \frac{\sum_{\nu=0}^{\nu_{\text{proc}}-1} \sum_{i=0}^{|p|-1} l_i^{\nu}}{\nu_{\text{proc}} \cdot |p|}$$

$\nu_{\text{proc}} \in \mathbb{N}^+$: Prozessanzahl

$|p|$: Anzahl der I/O-Requests der Prozesse (Def. 3.1)

$l_i^{\nu} \in \mathbb{N}^+$: Byteanzahl der von Prozess ν in Anforderung i bearbeiteten Daten

Der Parameter hat folgenden Wertebereich:

$$\text{Wertebereich} : \quad D_{\text{cont}} = \{x : x \in \mathbb{Q}^+\}$$

In dem vereinfachten Fall, dass alle l_i^p gleich groß sind und dem Wert l entsprechen (was bei dem wichtigen Einsatzszenario von Matrizen mit gleich großen Elementen der Fall ist), vereinfacht sich die Formel zu:

$$\text{contMean} \quad : \quad \nu_{\text{cont}} = \frac{\nu_{\text{proc}} \cdot |p| \cdot l}{\nu_{\text{proc}} \cdot |p|} = l$$

In diesem Fall ist *contMean* also immer ganzzahlig.

3.3.5 Parameterübersicht

Es wurden neun Parameter definiert, die die I/O-Last paralleler Applikationen basierend auf der MPI-IO-Schnittstelle genau beschreiben. Die neun Parameter lauten:

- uniqueBytes
- sizeMean
- readFrac
- seqFrac
- processNum
- sharedFrac
- syncFrac
- collFrac
- contMean

Es wurde auch gezeigt, dass alle diese neun Parameter unabhängig voneinander verwendet werden können, d. h. jeder Parameterwert kann mit jedem anderen Parameterwert innerhalb der festgelegten Definitionsbereiche kombiniert werden. Diese Eigenschaft bildet die Grundlage der Definition eines metrischen Raumes, der im Folgenden als I/O-Workload-Raum bezeichnet wird.

3.3.6 Definition des I/O-Workload-Raumes

Jeder der vorgestellten neun Parameter kann unabhängig von den übrigen acht Parametern beliebige Werte innerhalb seines Wertebereiches annehmen. Es kann also eine Menge $\mathcal{S}_{IO}$ als karthesisches Produkt über den Wertebereichen der 9 eingeführten Parameter wie folgt definiert werden:

$$\begin{aligned}\mathcal{S}_{IO} &= D_{unique} \times D_{size} \times D_{read} \times D_{seq} \times \\ &\quad D_{process} \times D_{shared} \times D_{sync} \times D_{coll} \times D_{cont}\end{aligned}$$

Wenn es eine Funktion $p : \mathcal{S}_{IO} \times \mathcal{S}_{IO} \rightarrow \mathbb{R}$ gibt, für die die drei folgenden Axiome gelten, so ist p eine Abstandsmetrik für $\mathcal{S}_{IO}$ und $(\mathcal{S}_{IO}, p)$ ein metrischer Raum [88]:

1. $p(x,y) \geqslant 0 \quad \forall x,y \in \mathcal{S}_{IO}$ (Nichtnegativität) und
   $p(x,y) = 0 \Leftrightarrow x = y \quad \forall x,y \in \mathcal{S}_{IO}$ (Definitheit)

2. $p(x,y) = p(y,x) \quad \forall x,y \in \mathcal{S}_{IO}$ (Symmetrie)

3. $p(x,y) \leqslant p(x,z) + p(z,y) \quad \forall x,y,z \in \mathcal{S}_{IO}$ (Dreiecksungleichung)

Da die Wertebereiche jedes einzelnen Parameters der 9-Tupel von $\mathcal{S}_{IO}$ Teilmengen der reellen Zahlen sind, ist auch $\mathcal{S}_{IO} \subseteq \mathbb{R}^9$. Alle Teilmengen von $\mathbb{R}^9$, sowie $\mathbb{R}^9$ selbst sind in Verbindung mit der Euklidischen Metrik

$$p_{euk}(x,y) = \sqrt{\sum_{k=1}^{9} |x_k - y_k|^2} \qquad \text{mit } x = (x_1, x_2, \ldots, x_9), y = (y_1, y_2, \ldots, y_9)$$

metrische Räume [88], d. h. die oben genannten 3 Axiome sind mit dieser Metrik auf $\mathbb{R}^9$ und dessen Teilmengen erfüllt. Die euklidische Metrik berechnet ein Abstandsmaß, wie es in der 2-dimensionalen Ebene als Satz des Pythagoras bekannt ist.

Im Folgenden wird $\mathcal{S}_{IO}$ in Verbindung mit der Euklidischen Metrik p_{euk} als metrischer Raum mit der Bezeichnung I/O-Workload-Raum definiert:

Definition 3.13: I/O-Workload-Raum

Die Menge $\mathcal{S}_{IO}$, die als karthesisches Produkt über den Wertebereichen der neun Parameter *uniqueBytes*, *sizeMean*, *readFrac*, *seqFrac*, *processNum*, *sharedFrac*, *syncFrac*, *collFrac* und *contMean* gebildet wird, ergibt in Verbindung mit der Euklidischen Metrik p_{euk} einen metrischen Raum, den I/O-Workload-Raum.
Die neun Parameter werden auch als Dimensionen des I/O-Workload-Raumes bezeichnet.

Zur Vereinfachung wird im Folgenden auch die Menge $\mathcal{S}_{IO}$ als I/O-Workload-Raum bezeichnet. In diesem Fall wird immer die Euklidische Metrik als Teil des Raumes vorausgesetzt.

I/O-Arbeitspunkt

Jedes Element der Menge $\mathcal{S}_{IO}$ ist ein 9-Tupel und beschreibt eine mögliche I/O-Charakteristik, wie sie von einer Applikation verwendet werden kann. Solch ein Tupel, das auch einen Punkt innerhalb des I/O-Workload-Raumes darstellt, wird als I/O-Arbeitspunkt bezeichnet und wie folgt definiert:

Definition 3.14: I/O-Arbeitspunkt

Jedes $P_i \in \mathcal{S}_{IO}$ heißt ein I/O-Arbeitspunkt. I/O-Arbeitspunkte sind also 9-Tupel der Form:

$$\begin{aligned} P_i \quad &= \quad (u, m, r, s, p, h, y, c, t) \\ &\text{mit} \quad u \in D_{unique}, m \in D_{size}, r \in D_{read}, s \in D_{seq}, p \in D_{process}, \\ &\qquad h \in D_{shared}, y \in D_{sync}, c \in D_{coll} \text{ und } t \in D_{cont} \end{aligned}$$

Der I/O-Workload-Raum wird im Folgenden genutzt, um das I/O-Verhalten beliebiger Applikationen zu spezifizieren.

3.3.7 I/O-Workload von Applikationen

In Definition 3.13 wurde der 9-dimensionale I/O-Workload-Raum beschrieben, der die Abbildung einer I/O-Last als 9-Tupel (also I/O-Arbeitspunkt) in diesen Raum ermöglicht. In den verwandten Arbeiten [81] und [61] wird die Applikations-I/O-Last im fünf-dimensionalen Raum als ein Punkt definiert. Die Zusammenfassung des kompletten I/O-Verhaltens einer Applikation zu nur wenigen Parametern eines Punktes erzeugt einen großen Informationsverlust. Es stellt sich die Frage, ob es noch möglich ist, nach diesem Informationsverlust hinreichend genaue Aussagen über das I/O-Verhalten der gesamten Applikation zu treffen. In den genannten verwandten Arbeiten wird auf diese Frage nur eingegangen, indem Messungen präsentiert werden, die die Genauigkeit des

Ansatzes zeigen. Alternative Lösungen, die eine höhere Genauigkeit erreichen könnten, werden nicht diskutiert.

In dieser Arbeit wird der in den verwandten Arbeiten beschriebene Ansatz erweitert, um eine möglichst flexible und umfassende Spezifikation von I/O-Workload-Verhalten durchzuführen, ohne aber eine einfache Spezifikation zu verbieten. Dazu wird ein I/O-Workload-Modell definiert, dass eine beliebig feingranulare Spezifikation des Verhaltens ermöglicht. Die Granularität soll durch den Benutzer spezifizierbar sein und bei höchster Genauigkeit ein komplettes Wiederherstellen des I/O-Verhaltens einer Applikation ermöglichen.

I/O-Workload als Sequenz von I/O-Arbeitspunkten

Mögliche Zugriffsmuster einer Applikation reichen von einer einzigen I/O-Anforderung bis hin zu einer komplexen Folge von I/O-Anforderungen. Bislang wurden alle I/O-Anforderungen einer Applikation zu einem I/O-Arbeitspunkt zusammengefasst. Alternativ können auch feinere Granularitäten gewählt werden, bei denen nur eine bestimmte Anzahl der I/O-Anforderungen einer Applikation als ein I/O-Arbeitspunkt verstanden werden. Der gesamte Applikations-I/O-Workload wäre dann eine Sequenz von mehreren I/O-Arbeitspunkten. Im Extremfall der feinsten Granularität würde jede I/O-Anforderung auf einen I/O-Arbeitspunkt abgebildet werden, so dass die Applikationslast aus so vielen Arbeitspunkten besteht, wie sie I/O-Anforderungen besitzt.

I/O-Trajektorie

Im allgemeinen Fall kann ein Zugriffsmuster – der Workload einer Applikation – als eine (geordnete) Sequenz von Punkten des I/O-Workload-Raumes verstanden werden. Da diese Punkte in einer zeitlichen Relation zueinander stehen – der erste Arbeitspunkt wird zeitlich vor dem zweiten ausgeführt – ist die Bezeichnung der Trajektorie besser geeignet als Sequenz. Im Folgenden werden I/O-Applikationsverhalten also als Trajektorien im I/O-Workload-Raum oder I/O-Trajektorien bezeichnet. Die Menge aller I/O-Trajektorien $\mathcal{T}_{IO}$ wird in Definition 3.15 als die Menge aller I/O-Arbeitspunkte und aller möglichen Sequenzen von Arbeitspunkten mit beliebigen Längen definiert. $\mathcal{S}_{IO}^{k}$ bedeutet in dieser Definition das k-fache karthesische Produkt von $\mathcal{S}_{IO}$ und kann wie folgt rekursiv beschrieben werden:

$$\begin{aligned} \mathcal{S}_{IO}^{1} &= \mathcal{S}_{IO} \\ \mathcal{S}_{IO}^{k} &= \mathcal{S}_{IO} \times \mathcal{S}_{IO}^{k-1} \quad \text{für } k \geqslant 2 \end{aligned}$$

Definition 3.15: I/O-Trajektorie

Die Menge aller I/O-Trajektorien $\mathcal{T}_{IO}$ ergibt sich als

$$\mathcal{T}_{IO} = \bigcup_{k \geqslant 1} \mathcal{S}_{IO}^{k} = \mathcal{S}_{IO}^{1} \cup \mathcal{S}_{IO}^{2} \cup \mathcal{S}_{IO}^{3} \cup \ldots$$

I/O-Trajektorien grafisch

Graphisch veranschaulicht besteht eine I/O-Trajektorie aus Punkten im 9-dimensionalen I/O-Workload-Raum, die durch gerichtete Kanten miteinander verbunden sind. Abbildung 3.5 zeigt diesen Sachverhalt auf der linken Seite am Beispiel eines dreidimensionalen Workload-Raumes mit den Dimensionen A, B und C, in der zwei gerichtete Kanten die drei Punkte P1, P2 und P3 miteinander verbinden.

Alternativ können I/O-Trajektorien als gerichtete Graphen verstanden werden. Der gerichtete Graph $G_{IO} = (V, E)$ einer I/O-Trajektorie besteht aus einer Menge von Punkten $V : V \subset \mathcal{S}_{IO}$, die die Menge der I/O-Arbeitspunkte der Trajektorie darstellt und einer Menge von gerichteten Kanten zwischen diesen Punkten (also Tupeln $V \times V$), die die Reihenfolge der einzelnen Operationen verdeutlichen. Die Werte der einzelnen Parameter der I/O-Arbeitspunkte müssen bei einem gerichteten Graphen als Attribute der Knoten modelliert werden.

Abbildung 3.5 zeigt rechts eine Darstellung, die an die UML (Unified Modeling Language) [89] angelehnt ist und einem Zustandsdiagramm entspricht. Die einzelnen I/O-Arbeitspunkte werden als Zustände verstanden, deren Zustandsübergänge durch Pfeile gekennzeichnet sind. Der Startzustand wird durch einen ausgefüllten Kreis markiert, während der Endzustand anhand von zwei ineinander geschachtelten Kreise gekennzeichnet wird. Die Werte der Dimensionen A, B und C werden in der Statechart-Darstellung textuell in den Zuständen als Attribute angegeben. Dadurch wird eine Unabhängigkeit der Darstellung von der Anzahl der verwendeten Dimensionen erreicht, was insbesondere beim vorgestellten Lastmodell mit neun Dimensionen einen wesentlichen Vorteil bringt. Neun Dimensionen sind in einer Darstellungsart, wie sie links mit drei Dimensionen verwendet wird, nicht mehr nachvollziehbar darstellbar.

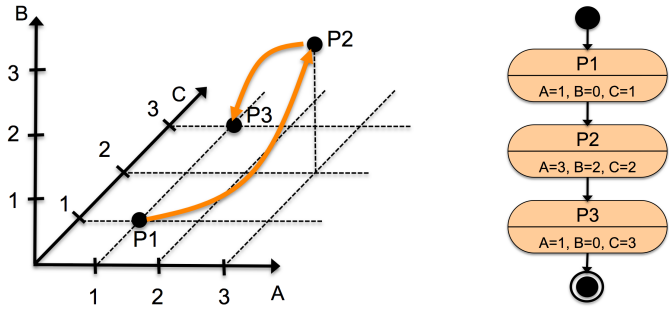

Abbildung 3.5: Grafische Darstellung eines Workloads als Trajektorie in einem dreidimensionalen Raum und äquivalente Darstellung als Statechart

Eine derartige sequentielle Aneinanderreihung von Punkten innerhalb des Workload-Raumes reicht für die Darstellung einer Applikationslast nahezu aus. Für eine allgemeine Darstellung muss zusätzlich die Möglichkeit von Schleifen existieren. Applikationen wie beispielsweise eine Webserver-Applikation haben eine quasi unendliche Laufzeit in der ständig zyklisch ähnliche I/O-Anforderungen durchgeführt werden. Um das I/O-Verhalten einer derartigen Applikation komplett zu beschreiben, ist die Möglichkeit eines Schleifenkonstruktes unerlässlich.

LOOP-Schleifen

In Definition 3.16 wird der I/O-Workload von Applikationen derart definiert, dass auch die Nutzung von LOOP-Schleifen möglich ist. LOOP-Schleifen in einer I/O-Lastbeschreibung werden auf Teile der Beschreibung angewendet und definieren eine Wiederholung des Lastbeschreibungsteils mit einer im Voraus definierten Anzahl an Wiederholungen. Damit sind LOOP-Schleifen eine vereinfachte Darstellung sehr langer I/O-Trajektorien, in denen sich einzelne Teile oftmals wiederholen.

Neben den LOOP-Schleifen mit fest definierter Schleifendurchlaufanzahl gibt es sogenannte WHILE-Schleifen, die eine Schleife abbrechen, sobald eine bestimmte Bedingung erfüllt ist. Deren Durchlaufanzahl wird also erst bei der Ausführung der Schleife bestimmt. WHILE-Schleifen bieten die Möglichkeit der Definition potentiell unendlicher Ausführungslängen, was im Beispiel eines Webservers notwendig wäre. Allerdings ist dies auch ein erheblicher Nachteil, da die Terminierung einer Lastbeschreibung sicher gestellt werden muss, wenn der die Lastbeschreibung verwendende Benchmark terminieren soll. Die Annäherung einer WHILE-Schleife einer Applikation kann jederzeit durch eine LOOP-Schleife erfolgen, deren Schleifenanzahl auf die durchschnittliche Anzahl von Schleifendurchläufen verschiedener Applikationsausführungen festgelegt wird (oder auf einen sehr hohen Wert bei potentiell unendlichen Ausführungslängen).

WHILE-Schleifen werden aus diesen Gründen im Folgenden bei der Definition eines I/O-Workloads nicht berücksichtigt.

Definition 3.16: I/O-Workload

$\mathcal{W}_{\mathrm{IO}}$ sei die Menge aller I/O-Workloads und werde wie folgt definiert:

$$\mathcal{W}_{\mathrm{IO}} \quad = \quad \mathcal{T}_{\mathrm{IO}} \cup \{\mathrm{LOOP}(w,x) \,:\, w \in \mathcal{W}_{\mathrm{IO}} \wedge x \in \mathbb{N}^{+}\}$$

Ein I/O-Workload kann also

- eine I/O-Trajektorie (also ein einzelner I/O-Arbeitspunkt oder eine Sequenz von ihnen) oder
- eine zyklische Wiederholung (LOOP) von I/O-Workloads (x gibt dabei die Anzahl der Schleifendurchläufe an)

sein.

grafische Lastdarstellung

Es wurde eine grafische Repräsentation der I/O-Lasten eingeführt, die Schleifendurchläufe nicht darstellen konnte. In Abbildung 3.6 wird die bereits in Abbildung 3.5 eingeführte Last um eine Schleife erweitert, die die letzten beiden I/O-Arbeitspunkte (P2 und P3) 10 Mal wiederholt. Arbeitspunkt P1 entspricht einer Initialisierung und die anderen beiden Arbeitspunkte der eigentlichen Berechnung. Dazu werden *Bedingungen* und *Aktionen* an Zustandsübergängen eingesetzt, wie sie in der UML spezifiziert sind. *Aktionen*, die am Übergangspfeil zwischen zwei Zuständen mit einem Schrägstrich (/) markiert sind, werden bei Durchführung des entsprechenden Übergangs ausgeführt.

Die zur Ausführung erforderliche Zeit muss vernachlässigbar klein sein, um einen undefinierten Zustand des Automaten während des Übergangs zu vermeiden. Dies kann bei den einfachen Aktionen, die bei diesem Last-Diagramm definiert sind, vorausgesetzt werden. Es gibt nur die Aktionen des Setzens der Iterationsvariablen auf einen Initialisierungswert (in diesem Fall 1) und des Inkrementierens bei jedem Durchlauf. Die Schleifenabfrage wird mit einer *Bedingung* realisiert, die im UML-Diagramm mit Hilfe von eckigen Klammern gekennzeichnet werden kann [89].

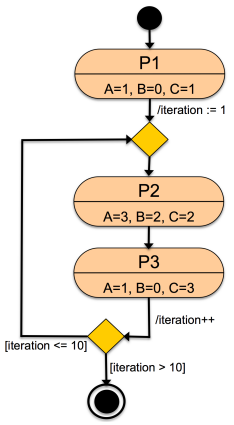

Abbildung 3.6: UML-basierte Darstellung einer I/O-Last als Zustandsdiagramm

Diese Darstellungsart von I/O-Lasten wird im Folgenden in dieser Arbeit eingesetzt, um schnell und übersichtlich I/O-Lasten zu beschreiben.

I/O-Workloads werden auf diese Weise recht genau beschrieben. Es ist möglich, I/O-Workloads feingranular, aber auch auf hohem Abstraktionsniveau darzustellen. Unterschiedliche Granularitätsebenen können auch problemlos miteinander vermischt werden. Beschreibungen auf feinster Granularitätsebene erlauben dabei eine komplette Rekonstruktion des Applikations-I/O-Workloads, sind allerdings weniger kompakt als Beschreibungen auf höherem Abstraktionsniveau.

3.3.8 I/O-Workloads als Ablaufbeschreibung

Lastengenerierung

Die in dieser Arbeit vorgestellte I/O-Workload-Beschreibung wurde zur Spezifikation von I/O-Workloads definiert. Sie kann ebenfalls als Ablaufbeschreibung für einen Lastgenerator verstanden werden, wie er in jedem I/O-Benchmark verwendet wird (s. Abbildung 3.3). Dieser Lastgenerator verwendet die Beschreibung eines Applikations-I/O-Workloads als Eingabe und erzeugt anschließend I/O-Lasten nach dem entsprechenden Muster, um die dabei entstehende I/O-Leistung zu analysieren.

Benchmarks haben eine wesentliche Eigenschaft: Sie müssen innerhalb einer endlichen Zeitspanne terminieren. Ein Benchmark, der aufgrund der fehlenden Terminierung kein Ergebnis liefert, ist sinnlos. Diese Eigenschaft kann sogar noch weiter eingeschränkt werden: Ein Benchmark muss innerhalb einer in Hinblick auf das Anwen-

dungsszenario sinnvollen Zeitspanne terminieren (vgl. Abschnitt 2.3.2). Die maximale Obergrenze für diese Zeitspanne wird bei den meisten Anwendungen einige Tage darstellen, da eine lange Messzeit die Rechenressourcen lange belegt und damit zu hohe Kosten verursacht. Bei Verwendung eines `WHILE`-Konstrukts in einer Benchmark-Lastbeschreibung kann die Terminierung des Benchmarks nicht garantiert werden, da unklar ist, wann die Abbruchbedingung der Schleife einen Benchmark terminieren würde. Somit ist die Verwendung eines `WHILE`-Konstrukts in einer Benchmark-Lastbeschreibung zweifelhaft und wurde nicht vorgesehen. Alle anderen verwendeten Konstrukte, inklusive `LOOP`-Schleife garantieren eine Terminierung des Benchmarks in endlicher Zeit.

3.3.9 Einschränkungen der Lastbeschreibung

Die in dieser Arbeit vorgestellte Lastbeschreibung für I/O-Workloads von Applikationen kennzeichnet I/O-Verhalten von Applikationen so genau, dass eine hinreichend exakte Wiedergabe des Verhaltens zur Performance-Bestimmung möglich ist. Jede Lastbeschreibung ist das Modell einer Last und hat je nach Exaktheit des Modells Ungenauigkeiten zur Folge. Ein Nachweis der Genauigkeit der vorgestellten Lastbeschreibung mittels Messungen folgt in Kapitel 5.

An dieser Stelle werden Punkte identifiziert, an denen Ungenauigkeiten in der Beschreibung existieren und es werden damit verbundene Bedenken diskutiert.

fehlendes `WHILE`

Da das `WHILE`-Konstrukt in der Lastbeschreibung für den Benchmark nicht erlaubt sein soll, entstehen damit verbundene Einschränkungen. Die Einschränkung, dass bei Verbot des `WHILE`-Konstruktes nicht beliebige I/O-Workloads definierbar sind, ist keine wesentliche Einschränkung. Nicht formulierbar sind unbestimmt lang laufende I/O-Charakteristiken, die durch eine I/O-Beschreibung mit endlicher Laufzeit (z. B. mit `LOOP`) angenähert werden müssen. Ein Vorteil eines fehlenden `WHILE`-Konstruktes hingegen ist, dass Parameter wie die Gesamtmenge von bearbeiteten Daten nicht von äußeren Einflüssen abhängen, die eventuell in der Abbruchbedingung der `WHILE`-Schleife definiert werden. Auf diese Weise sind Teile des Workloads leicht zu neuen I/O-Arbeitspunkten zusammenfassbar und es wird auch die Änderung der Granularität einer Lastbeschreibung durch Zusammenfassung oder Teilung von I/O-Arbeitspunkten ermöglicht.

Kumulation

I/O-Lastbeschreibungen nach der vorgestellten Form kumulieren die Parameter mehrerer I/O-Anforderungen und bilden sie auf einen I/O-Arbeitspunkt ab. Dies ist natürlicherweise mit einem Informationsverlust verbunden, der einen Einfluss auf die Genauigkeit der Abbildung hat. Diesbezüglich gibt es insbesondere zwei Abbildungsprobleme: die Reihenfolge und die Anforderungsgröße der I/O-Anforderungen.

Reihenfolge

Die Reihenfolge der I/O-Anforderungen, die einem I/O-Arbeitspunkt entsprechen, kann in der Lastbeschreibung nicht abgebildet werden. Bei Rekonstruktion einer Last aus der Lastbeschreibung muss eine willkürliche Reihenfolge gewählt werden. Durch Erhöhung der Granularität der Lastbeschreibung kann das Problem allerdings verkleinert werden, da die Reihenfolge von I/O-Arbeitspunkten in der Lastbeschreibung klar definiert ist. Im Extremfall, in dem ein I/O-Arbeitspunkt einer I/O-Anforderung entspricht (also höchste Granularität der Beschreibung) kann die Ausführungsreihenfolge komplett rekonstruiert werden.

Anforderungsgröße

Das beschriebene Lastmodell fässt die Anforderungsgrößen aller Anforderungen eines I/O-Arbeitspunktes zu einem Mittelwert (*sizeMean*) zusammen. Die genauen Größen der Einzeloperationen sind nicht rekonstruierbar. Ein Algorithmus, der eine Last aus der Lastbeschreibung zurück gewinnen möchte, kann entweder alle Anforderungen eines I/O-Arbeitspunktes mit der gleichen Größe *sizeMean* durchführen, oder stochastische Abweichungen von dieser Größe realisieren. Dazu ist aber eine Standardabweichung vom Mittelwert zu definieren, was eine weitere Erhöhung der Dimensionalität des Modells zur Folge hat und aufgrund der typischen Anfragegrößen, die oftmals ganze Blockgrößen sind (bspw. 4 kb, 8 kB, 16kB . . .), auch nicht sinnvoll ist. Auch hier kann das Problem verkleinert werden, indem die Granularität der Lastbeschreibung erhöht wird. Bei höchster Granularität gibt es keinen Informationsverlust mehr.

In diesem Kapitel wurde beschrieben, dass die vorgestellte I/O-Lastbeschreibung als Grundlage der Last eines I/O-Benchmarks verwendet wird. Es wird also anhand der Parameterwerte eines I/O-Arbeitspunktes eine I/O-Anforderungsfolge definiert, die der I/O-Requestfolge der Originalapplikation entspricht. In Kapitel 4 wird ein Algorithmus vorgestellt, der diese Rückgewinnung des I/O-Workloads ermöglicht. Es wird aber auch gezeigt, dass der Algorithmus nicht in allen Fällen exakt funktioniert. Neben den hier genannten Abbildungsproblemen der Lastbeschreibung kommen Probleme des Algorithmus hinzu, die Lastbeschreibung auf eine spezifische Last abzubilden.

Parallelität der Prozesse

Eine weitere Einschränkung bietet die Workload-Beschreibung hinsichtlich der parallelen Prozesse. Eine High-Performance-Applikation besteht aus zahlreichen Prozessen, die gemeinsam eine Aufgabe bearbeiten. Die Prozesse greifen dabei unabhängig voneinander auf Sekundärspeicher zu. Das in der Arbeit beschriebene Modell fasst diese parallelen Aktivitäten zusammen, so dass aus ursprünglich parallelen Pfaden ein einzelner sequentieller Pfad entsteht. Diese Vorgehensweise hat den Vorteil, dass I/O-Applikationsverhalten sehr einfach als sequentielle Folge von Operationen beschrieben werden kann. Sonst wäre eine Beschreibung als Netz notwendig, in dem unterschiedliche Operationen auf parallelen Pfaden zeitgleich ausgeführt werden können. Im Folgenden wird gezeigt, dass diese scheinbar große Beschränkung in der Praxis nur bei wenigen parallelen Applikationen einschränkend wirkt.

Parallelität mehrerer Prozesse

Parallele Programme bestehen aus zahlreichen Prozessen, die zeitgleich auf unterschiedlichen Rechenknoten eines Rechensystems ausgeführt werden. Die Programmierung paralleler Applikationen ist komplexer als die Programmierung sequentieller Programme, da Probleme, wie zum Beispiel Kommunikation und Synchronisation zwischen den Prozessen oder Datenaufteilung- und Verteilung zu lösen sind, die bei sequentieller Programmierung nicht existieren. Um diese hohe Komplexität zu verringern, gibt es parallele Programmiermodelle, die den Programmierer bei der Implementierung und Verteilung paralleler Applikationen unterstützen. Klassischerweise wird die parallele Programmierung in zwei Parallelitätsformen unterteilt [90]:

implizite Parallelität

- *Implizite Parallelität*– wird bei parallelen Programmiersprachen und parallelisierenden Compilern verwendet; der Programmierer hat keinen Einfluss und keine Möglichkeit zu spezifizieren, wie und wohin Code aufgeteilt bzw. verteilt wird,

explizite Parallelität

- *Explizite Parallelität*– hier ist der Programmierer für die Arbeitsaufteilung, der Abbildung von Aufgaben auf Prozessoren und die Kommunikationsmuster verantwortlich; unterstützt wird er durch ein System, das eine Abstraktion vom verwendeten Rechensystem und der Kommunikationsinfrastruktur erreicht.

Untersuchungen haben gezeigt, dass bei Nutzung expliziter Parallelität aufgrund des vorhandenen Zusatzwissens des Entwicklers eine höhere Effizienz erreicht werden kann [91]. In Spezialanwendungen wie z. B. bei Compilern für Vektorrechner wird zwar die *implizite Parallelität* verwendet, allerdings nutzen die meisten aktuellen Anwendungen *explizite Parallelität*, da moderne Message-Passing-Schnittstellen wie MPI zur Realisierung der letzteren Parallelitätsform verwendet werden. Für die Nutzung der *expliziten Parallelität* gibt es die folgenden Programmierparadigmen, die im Folgenden in Hinblick auf die Anzahl der Parallelitätsstränge mit unterschiedlichem Verhalten vorgestellt werden [91]:

- Master-Slave
- Single-Program Multiple-Data
- Data Pipelining
- Divide-and-Conquer
- Speculative Parallelism

Master-Slave. Applikationen, die nach dem Master-Slave-Programmierparadigma implementiert wurden, bestehen aus einem Master und mehreren Slaves, welche die durch den Master zugeteilten Aufgaben bearbeiten, um die Daten an den Master zurückzusenden. Alle Slaves führen dabei gleiche Programme aus, die unterschiedliche Daten bearbeiten. Master und Slave unterscheiden sich unter Umständen stark in ihrer Ausführung. Da normalerweise der Master die Daten für die Slaves liefert, ist er häufig

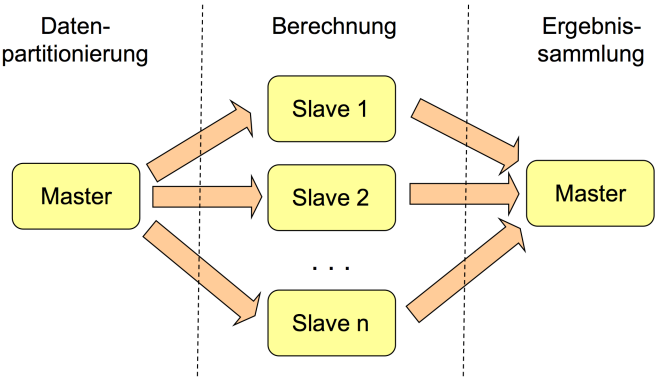

Abbildung 3.7: Gleiches Verhalten aller Slaves beim Master-Slave-Programmierparadigma

der Prozess, der das Lesen und Schreiben der Daten, also die I/O-Zugriffe übernimmt. Eine Abbildung des parallelen I/O-Verhaltens der Applikation mit dem eingeführten I/O-Modell ist in einem solchen Fall problemlos möglich, da kein paralleler I/O-Zugriff stattfindet. Selbst wenn, nach der Zuteilung von Daten durch den Master, I/O-Zugriffe durch Slaves durchgeführt werden, ist eine Abbildung durch das eingeführte Modell möglich, da alle Slaves grundsätzliches das gleiche Programm ausführen, also ein gleiches I/O-Verhalten zeigen. Der I/O-Workload einer derartigen Applikation bestände aus zwei Phasen, in deren erster Phase ein Prozess (nämlich der Master) mit einem bestimmten Zugriffsmuster I/O durchführt, während $n-1$ Prozesse in der zweiten Phase auf den Sekundärspeicher mit einem anderen Zugriffsmuster zugreifen. Allgemein ist die Parallelität des I/O-Lastverhaltens von Applikationen, die das Master-Slave-Programmierparadigma verwenden, mit dem eingeführten I/O-Lastmodell gut abbildbar, da alle Slaves aufgrund ihres ähnlichen Verhaltens gleiche Dimensionswerte im I/O-Workload-Raum besitzen.

Single-Program Multiple-Data. Das SPMD-Paradigma ist laut [91] das häufigst verwendete Programmierparadigma, bei dem jeder Prozess gleichen Code mit unterschiedlichen Daten ausführt. Das vorgestellte I/O-Lastmodell ist insbesondere für Applikationen dieser Paradigmen-Art optimiert und kann sie aufgrund der gleichen I/O-Charakteristik aller Prozesse sehr gut abbilden.

Divide-and-Conquer. Die Idee des Divide-and-Conquer-Ansatzes ist aus der sequenziellen Programmierung bekannt. Ein Problem wird in zwei Teilprobleme mit geringerer Problemgröße geteilt, die mittels des gleichen Ansatzes rekursiv weiterbearbeitet werden. Dies geschieht solange, bis die Problemgröße so klein ist, dass die Berechnung leicht erfolgen kann. Im Anschluss werden die Ergebnisse der Berechnung an den Vater zurückgegeben. Damit lässt sich der Divide-and-Conquer-Ansatz als virtueller Baum verstehen, der einen Wurzelknoten besitzt, mehrere innere Knoten,

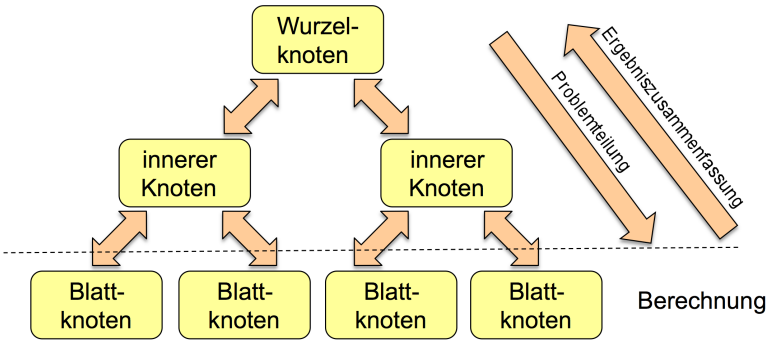

Abbildung 3.8: I/O beim Divide and Conquer-Ansatz wird durch Wurzelknoten oder Blattknoten durchgeführt

die die Problemteilung und auch die Ergebniszusammenfassung am Ende in mehreren Ebenen durchführen und den Blattknoten, die die Probleme lösen. Abbildung 3.8 verdeutlicht dies. Da I/O typischerweise entweder vom Wurzelknoten oder von den Blattknoten, die auch die Berechnungen ausführen, durchgeführt wird, lassen sich Applikationen, die nach dem Divide-and-Conquer-Ansatz implementiert wurden, sehr gut mit dem vorgestellten I/O-Lastmodell beschreiben. In jedem Fall ist das I/O-Verhalten aller an dem I/O beteiligten Knoten gleich und kann mit identischen Dimensionswerten charakterisiert werden.

Data Pipelining. Applikationen dieses Paradigmas teilen ihre komplexe Aufgabe in einfachere Teilaufgaben, die durch einzelne Prozessoren nacheinander durchgeführt werden. Der Parallelitätsgrad entsteht durch die parallele Bearbeitung der Teilaufgaben auf unterschiedlichen Datenanteilen. Die Verwendung des Data Pipelinings bringt insbesondere bei der Bearbeitung von Stromdaten Vorteile, hat aber in der einfachen geschilderten Form den Nachteil, dass keine Skalierung der Pipeline möglich ist. Wenn eine komplexe Aufgabe in 10 Teilaufgaben unterteilt wurde, bringt eine Aufteilung der Aufgabe auf bis zu 10 Ausführungseinheiten einen Geschwindigkeitsgewinn. Eine Nutzung von mehr Ausführungseinheiten ist nicht möglich. Aus diesem Grund

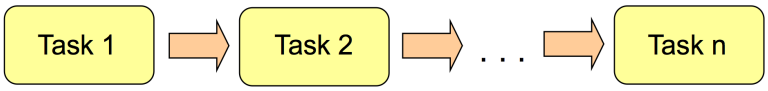

Abbildung 3.9: Beim Data Pipelining wird I/O typischerweise von Task 1 und Task n durchgeführt

wird das Data Pipelining häufig in Anwendungen eingesetzt, in denen die Anzahl der Teilaufgaben mit der Problemgröße skaliert, wobei die Teilaufgaben dann identisch sind. Das I/O-Verhalten einer typischen Data-Pipelining-Applikation lässt sich in Abbildung 3.9 erkennen. Daten werden von einem Ausführungselement an das nächste weitergereicht. Lediglich die erste Ausführungseinheit liest die zu verarbeitenden Daten ein, während die letzte Ausführungseinheit die Daten ausgibt. I/O findet also

nur in zwei parallelen Prozessen statt. Da beide Prozesse ein stark unterschiedliches I/O-Verhalten bei paralleler Ausführung besitzen können, ist eine Abbildung des I/O-Verhaltens von Data-Pipelining-Applikationen mit dem vorgestellten I/O-Modell nur mit Näherungen möglich. Um diesen Applikationstyp korrekt abzubilden, müsste das I/O-Modell um parallele Ausführungsstränge erweitert werden, d. h. die aktuell lineare Darstellung einer Last als I/O-Trajektorie müsste zu einem Netz erweitert werden. Um diese erhöhte Komplexität des I/O-Modells zu vermeiden, werden für Data-Pipelining-Applikationen zwei Näherungsmöglichkeiten vorgestellt, die im vorgestellten Lastmodell abbildbar sind: Eine Näherung ist die Sequenzialisierung der beiden eigentlich parallelen Operationen. Eine zweite Möglichkeit der Näherung ist die Zusammenfassung der Eingabe- und der Ausgabeoperationen in einen I/O-Arbeitspunkt mit zwei Prozessen. Beide Prozesse führen dann das gleiche I/O-Verhalten parallel aus.

Speculative Parallelism. Parallele Programme dieser Art finden bei komplexen Problemen Anwendung, die aufgrund starker Datenabhängigkeiten schlecht parallelisierbar sind. Beim spekulativen Parallelismus werden Annahmen über die Zukunft (bspw. über Bedingungen von Schleifen oder `if-then-else`-Strukturen) getroffen, deren Eintritt zum aktuellen Zeitpunkt noch nicht sicher ist, um sofort Rechnungen durchzuführen, die beim Abwarten des Eintritts dieser Annahmen erst in Zukunft anstehen würden. Es werden so ungenutzte Rechenressourcen verwendet – wenn auch mit der Gefahr, dass die berechneten Ergebnisse beim Nicht-Eintreten der Annahmen ignoriert werden müssen. Die Applikation muss in einem solchen Fall in den letzten richtigen Zustand zurückgesetzt werden. Waren die Zukunftsannahmen korrekt, werden die spekulativ gewonnenen Ergebnisse übernommen. Programme, die spekulativ agieren, ordnen sich architektonisch in eine der vorher genannten Klassen ein. Viele sind als Master-Slave-Programme oder SPMD-Programme implementiert, weshalb auch die dort getroffenen Aussagen auf diese Applikationen zutreffen. An dieser Stelle wird dieser Applikationstyp paralleler Programme nur der Vollständigkeit wegen aufgeführt.

Darstellung der Parallelität des I/O

Es zeigt sich also, dass Applikationen der meisten Klassen paralleler Programme zur Verringerung der Komplexität entweder

- einheitliche Programmstrukturen für alle Prozesse verwenden und damit auch gleiche I/O-Charakteristiken bei allen Prozessen besitzen, oder
- bestimmte Programmfunktionalität (so auch I/O) auf einzelne Prozesse auslagern.

Auch ohne die explizite Unterstützung von parallelem unterschiedlichen I/O-Verhalten werden beide Fälle mit dem vorgestellten I/O-Modell sehr gut abgebildet. Im ersten Fall wird die I/O-Charakteristik der Prozesse ermittelt und als Parameter *procNum* die Anzahl der parallelen Prozesse verwendet, während im zweiten Fall nur die I/O-Charakteristik des I/O-durchführenden Prozesses und dem *procNum*-Wert von 1 zur I/O-Beschreibung der Applikation verwendet wird. Lediglich einige Applikationen, die das

Data Pipelining verwenden, werden nicht immer erfasst, da bei diesen zwei separate Prozesse mit unterschiedlichen I/O-Charakteristiken existieren können. Aber auch bei Data-Pipelining-Applikationen gibt es mögliche Näherungen für die Darstellung im vorgestellten Lastmodell.

Auch wenn alle Prozesse einer Applikation ein prinzipiell gleiches I/O-Verhalten verwenden, können sie dennoch unterschiedliche Mengen an Daten transferieren, da jeder Prozess eine andere Sicht auf eine Datei haben kann. Dieser Zusammenhang wurde in Absatz 3.3.3 unter dem Punkt *Sequentialität bei MPI-IO* bereits erläutert. Datentransfers, die mittels unterschiedlicher Dateisichten von Prozessen verschränkt gelesen oder geschrieben werden, werden in der Literatur als *noncontiguous I/O* bezeichnet und sind ein grundlegendes Konzept [69] von MPI-IO. Der Parameter *contMean* wurde eingeführt, um *noncontiguous I/O* möglichst genau abzubilden. *ContMean* wurde als durchschnittliche Größe aller zusammenhängenden Dateiteile berechnet, die von einem Prozess bearbeitet werden. Naturgemäß ist diese Durchschnittsbildung mit einem Informationsverlust verbunden. Dennoch werden sowohl verschränkte als auch zusammenhängende Zugriffe, die die beiden wichtigsten Zugriffsarten paralleler Applikationen darstellen, mit dem vorgestellten Lastmodell genau charakterisiert, wenn die folgenden Punkte beachtet werden:

- *SizeMean* spezifiziert die Größe der Dateieinheit bei parallelen Zugriffen – dies ist keine Änderung sondern eine Präzisierung gegenüber der bislang eingeführten Definition im Falle paralleler Zugriffe.

- Ist die Dateieinheit größer oder gleich $\mathrm{processNum} \cdot \mathrm{contMean}$, bearbeitet jeder Prozess *contMean* Bytes innerhalb einer Dateieinheit. Bei Gleichheit gibt es innerhalb der Dateieinheit keine Lücken, sonst werden entsprechend Lücken eingefügt. Diese Modellierung realisiert verschränkte Zugriffe (*noncontiguous I/O*).

- Ist die Dateieinheit kleiner dem Wert $\mathrm{processNum} \cdot \mathrm{contMean}$, bearbeitet jeder Prozess *contMean* Bytes aufgeteilt auf $\lfloor \frac{\mathrm{contMean}}{\mathrm{sizeMean}} \rfloor$ Zugriffseinheiten. Für den bei der Division entstehenden Rest wird eine weitere Zugriffseinheit erstellt, die die fehlenden Bytes und eine entsprechend große Lücke enthält. Auf diese Weise werden große zusammenhängende Zugriffe realisiert (*contiguous I/O*).

Aus den Ausführungen folgt eine wesentliche Einschränkung bei der Modellierung von verschränkten Zugriffen. Applikationen, in denen verschiedene Prozesse unterschiedliche Anzahlen von Daten lesen oder schreiben, sind nicht korrekt abbildbar, da *contMean* für alle Prozesse einen identischen Wert definiert. Dies kann in einigen Applikationen eine Einschränkung sein. Allerdings wurde gezeigt, dass die Mehrheit der parallelen Applikationen verschränkte oder zusammenhängende Zugriffsmuster verwenden, bei denen dieses Verhalten nicht auftritt.

3.3.10 Zusammenfassung I/O-Lastmodell

Aufgrund bislang fehlender Möglichkeiten, einfach und genau Lasten für parallele I/O-Systeme zu spezifizieren, wurde in diesem Abschnitt ein neues Lastmodell bestehend aus 9 Parametern vorgestellt, das I/O-Lasten paralleler Applikationen, insbesondere paralleler Applikationen, die die MPI-IO-Schnittstelle verwenden, durch Phasen spezifiziert, von denen jede Phase mit Werten der 9 Parameter definiert werden kann. Es wurde eingehend erläutert, warum 9 Parameter erforderlich sind, um hinreichend genau die Möglichkeiten der MPI-IO-Schnittstelle abzubilden. Dennoch besitzt dieses Modell Einschränkungen in der Darstellung, die in den letzten Abschnitten genau beleuchtet wurden.

Das entwickelte I/O-Lastmodell kann nun als Lastbeschreibung für einen I/O-Benchmark-Lastgenerator dienen. Allerdings existieren für die eigentliche Messung verschiedene Möglichkeiten, die im Folgenden diskutiert werden. Eine dieser Möglichkeiten wird für die Verwendung in einem I/O-Benchmark favorisiert.

3.4 Klassifikation von I/O-Performance-Messmethoden

Nachdem im letzten Absatz eine Möglichkeit präsentiert wurde, wie I/O-Workloads von Applikationen spezifiziert werden können, wird in diesem Abschnitt gezeigt, welche prinzipiellen Möglichkeiten Benchmarks haben, um die I/O-Leistung eines spezifischen Rechensystems zu vermessen. Der eingeführte I/O-Workload-Raum dient als Konstrukt, um vorhandene I/O-Benchmark-Messmethoden zu klassifieren.

Grundsätzlich können I/O-Benchmarks eine Leistungsermittlung auf drei verschiedene Arten durchführen:

1. Eine vollständige Vermessung variiert sämtliche Dimensionen des Workload-Raumes in allen Kombinationen und ermöglicht so ein vollständiges Leistungsabbild des I/O-Systems.

2. Eine Messung mit Hilfe der Nutzung niederdimensionaler Räume variiert nur einige Dimensionen im Workload-Raum und hält die übrigen Parameter konstant. Dadurch kann der Messaufwand gegenüber einer vollständigen Vermessung deutlich verringert werden.

3. Die Messung anhand von Punkten innerhalb des Workload-Raumes sucht gezielt nutzerrelevante Punkte im Workload-Raum aus, um eine Leistungsermittlung anhand dieser durchzuführen. Dadurch wird der Messaufwand minimal gehalten.

Alle drei Methoden werden im Folgenden vorgestellt. Es wird gezeigt, welche Messmethoden für einen möglichst optimalen I/O-Benchmark zu verwenden sind. Ziel dieses Abschnitts ist es, herauszufinden, welche Messmethode ein I/O-Benchmarks verwenden muss, um die in Absatz 2.3.2 gestellten Zielanforderungen an einen optimalen I/O-Benchmark zu erfüllen.

3.4.1 Vollständige Vermessung des I/O-Workload-Raumes

Unterschiedliche Applikationen verwenden I/O-Workloads, deren I/O-Arbeitspunkte an verschiedenen Positionen innerhalb des I/O-Workload-Raumes liegen. Aufgrund der verschiedenen I/O-Charakteristiken besitzen die Arbeitspunkte voneinander abweichende Leistungswerte. Um somit die Leistung eines I/O-Systems für alle Applikationen repräsentativ zu erfassen, ist es erforderlich, den I/O-Workload-Raum vollständig zu vermessen. Das heißt, es müssen alle Dimensionen des Workload-Raumes variiert werden, um ein umfassendes Bild über das I/O-Verhalten zu erhalten.

Messaufwand

Eine vollständige Vermessung des I/O-Workload-Raumes bedeutet eine Vermessung zahlreicher I/O-Arbeitspunkte, bei denen alle neun Parameter, die den I/O-Workload-Raum aufspannen, variiert werden. Abbildung 3.10 verdeutlicht dies an einem 3-dimensionalen Beispiel, bei dem pro Dimension 3 Messwerte aufgenommen werden. Insgesamt handelt es sich also um $3^3 = 27$ Messwerte. In [62] wird angegeben, dass die Vermessung des I/O-Workload-Raumes bei 6 Messungen pro Dimension im dort beschriebenen nur 5-dimensionalen I/O-Workload-Raum auf einer Sprite DECstation fast 2 Monate dauern würde, da jede Messung ca. 10 Minuten dauert. Bei einem 9-dimensionalen Workload-Raum würde die Messzeit bei sonst gleichen Voraussetzungen auf fast 192 Jahre ansteigen! Allgemein kann die Komplexität der Messzeit in Abhängigkeit von der Anzahl der Dimensionen d und der Anzahl der Messpunkte n pro Dimension wie folgt angegeben werden:

$$t(n) = O(n^d)$$

Derartige exponentielle Zeiten sind auf keinen Fall akzeptabel, da n für ein genaues Abbild des Workload-Raumes möglichst groß gewählt werden sollte und damit die Messzeiten sehr schnell ansteigen würden.

Ergebnispräsentation

Weiterhin würde bei der Vermessung des 9-dimensionalen Workload-Raumes mit 6 Messwerten pro Dimension ein Datenvolumen von über 10 Mill. Messwerten anfallen. Dieses Datenvolumen muss dem Nutzer des Benchmarks ansprechend, verständlich und aussagekräftig präsentiert werden, wenn die Ergebnisse von Nutzen sein sollen. Auch dies gestaltet sich schwierig.

Diese zwei Argumente sprechen zumindest zum aktuellen Zeitpunkt gegen eine komplette Vermessung des I/O-Workload-Raumes. Es ist nicht auszuschließen, dass durch die Weiterentwicklung der Technologie zukünftig die Vermessung des gesamten I/O-Workload-Raumes in angemessener Zeit realisierbar ist. Allerdings bleibt auch dann

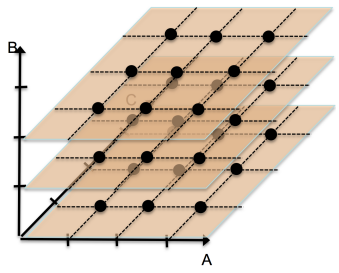

Abbildung 3.10: Die vollständige Vermessung eines n-dimensionalen Workload-Raumes besitzt einen exponentiellen Messaufwand.

das Problem der Auswertung der komplexen Ergebnisse. Im Folgenden werden deshalb zwei Ansätze beschrieben, die bereits gegenwärtig in der Praxis einsetzbar sind.

3.4.2 Leistungsmessung anhand von niederdimensionalen Räumen

Messaufwand

Bei diesem Verfahren werden Räume niedrigerer Dimension als der I/O-Workload-Raum selbst als Messgrundlage verwendet. Dazu werden einige Dimensionen innerhalb des Raumes variiert (Faktoren), während alle restlichen Dimensionen konstant gehalten werden. Abbildung 3.11 zeigt im dreidimensionalen Raum zwei Beispiele, in denen in der linken Abbildung nur der Parameter B konstant gehalten wird und durch die Variation der Parameter A und C eine Ebene als Messgrundlage verwendet wird, während rechts ein Parameter variabel ist und so eine Gerade vermessen wird.

Der Fall der Vermessung einer variablen Dimension ist der einfachste Fall, der in [61] als Messgrundlage vorgeschlagen wird. Der Autor empfiehlt die Messung einer Geraden pro Parameter, so dass pro Dimension des Workload-Raumes ein Graph entsteht, der dem Nutzer als Ergebnis der Messung präsentiert wird. Der Messaufwand reduziert sich bei dieser Methode enorm. Er hat nun nur noch eine lineare Abhängigkeit von der Anzahl der Messwerte pro Dimension n:

$$t(n) = O(n \cdot d)$$

In [61] wird diese Methode vorgeschlagen, um dem Benutzer einen möglichst genauen Eindruck von der Performance des I/O-Systems zu verschaffen und dennoch in angemessener Zeit zu verständlichen Ergebnissen zu gelangen.

Diese Messmethode ist nicht ohne Probleme: Ein Problem ist die Wahl der Parameterwerte für die nicht-variablen Dimensionen. Es kann dem Nutzer die Wahl der Parameter überlassen werden oder es wird ein heuristisches Verfahren verwendet. In [80] definieren die Autoren einen besonderen I/O-Arbeitspunkt, den *focal point*, an dem das I/O-System mindestens 75 % der Maximal-Leistung erreicht. Die Parameterwerte

focal point

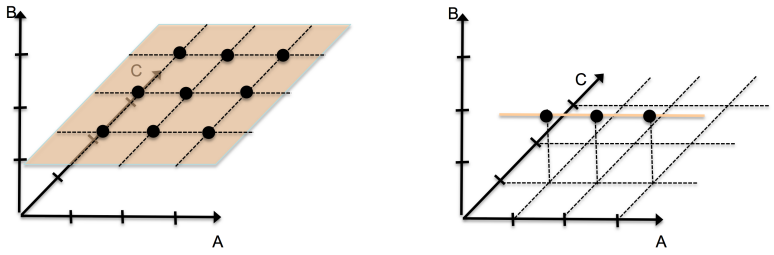

Abbildung 3.11: Vermessung einer Schnittebene und einer Gerade eines 3-dimensionalen Workload-Raumes

der Dimensionen werden entsprechend dem *focal point* gewählt und jeweils eine Dimension wird variiert. Mit diesem Weg soll ein Arbeitspunkt gewählt werden, bei dem sich das System gut, aber nicht optimal verhält. Für viele Anwender wird das Benchmarking aber eher im Falle von schlechter Systemleistung interessant sein, weil nur dann ein Optimierungspotential gegeben ist. Selbst wenn der verwendete *focal point* einer für bestimmte Anwender interessanten Last entspricht, wird lediglich eine Gerade, die parallel zu einer Dimension verläuft, vermessen. Bezüglich der Freiheitsgrade der Messung ist der Nutzer also extrem eingeschränkt.

Vergleichbarkeit

Fraglich ist ebenfalls, ob es möglich ist, I/O-Systeme anhand der mit diesem Verfahren ermittelten Ergebnisse zu vergleichen. Wird auf beiden Systemen der gleiche Arbeitspunkt verwendet, ist eine Vergleichbarkeit auf jeden Fall gegeben. Wird die Heuristik aber unterscheidliche Arbeitspunkte auf beiden Rechnersystemen ermitteln, ist eine Vergleichbarkeit nicht mehr gegeben, da die unterschiedlichen Rechnersysteme zur Vermessung grundsätzlich unterschiedliche Workloads verwenden. In [80] wird für diesen Fall die Methode der *predicted performance* vorgeschlagen. Anhand der Graphen für die Abhängigkeiten der I/O-Leistung von den einzelnen Dimensionsparametern kann die I/O-Performance an jedem Punkt des I/O-Workload-Raumes vorhergesagt werden. Es wird dafür angenommen, dass sich die Form eines Graphen unabhängig von allen anderen Parametern nicht verändert. Es verändern sich nur die absoluten Werte.

predicted performance

Die Autoren stellen in dem Beitrag fest, dass vorhergesagte Leistungswerte in 25 % der untersuchten Beispiel-Lasten um mehr als 15 % von den tatsächlichen Leistungswerten abweichen. Eine gesicherte Aussage zur Leistung eines I/O-Systems kann mit dieser Methode also nicht getroffen werden. Dennoch kann das Verfahren durchaus in Anwendungsfällen interessant sein, in denen eine ungesicherte Vorhersage der Leistung vieler Applikationen interessanter ist als eine gesicherte Aussage über die I/O-Leistung von wenigen Applikationen, da Leistungswerte der Workloads beliebiger Applikationen aus nur einer Messung bis zu einem gewissen Grad vorhergesagt werden können.

3.4.3 Leistungsmessung anhand einzelner Punkte

Messaufwand

Die bei I/O-Benchmarks am häufigsten verwendete Messmethode ist die Vermessung einzelner Arbeitspunkte des I/O-Workload-Raumes. Da bei dieser Messmethode eine feste Anzahl von I/O-Arbeitspunkten vermessen wird, ist die Messzeit in diesem Fall nur linear abhängig von der Anzahl der Messpunkte n:

$$t(n) = O(n)$$

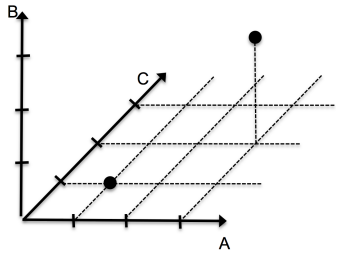

Abbildung 3.12: Vermessung eines 3-dimensionalen Workload-Raumes mit Punkten erzeugt linearen Aufwand

statische Ein-Punkt-Messung

Die punktbasierte Messung wird im Folgenden aufgrund verschiedener existierender Messmethoden, die in diese Kategorie fallen, weiter unterteilt. Die einfachste Messmethode, die statische Ein-Punkt-Messung, verwendet einen fest definierten I/O-Arbeitspunkt, also einen I/O-Workload. Da n in diesem, in der Praxis sehr häufig auftretenden Fall, den Wert 1 hat, verringert sich der Messaufwand weiter zu $t(n) = O(1)$. Es wurde bereits festgestellt, dass sich die Leistung eines I/O-Systems an unterschiedlichen I/O-Arbeitspunkten um Größenordnungen unterscheiden kann. Entsprechend ist es sehr wichtig, dass Benchmarks, die nur einen einzelnen I/O-Arbeitspunkt vermessen, eine Last verwenden, die für den Benchmark-Nutzer relevant ist, d. h. einer für ihn relevanten Applikation entspricht. Wie bereits gezeigt wurde, ist dies bei den meisten vorhandenen Benchmarks, die einzelne Arbeitspunkte vermessen, nicht der Fall. Häufig verwenden sie einen I/O-Workload, der durch den Benchmarkentwickler fix definiert ist und durch den Benchmark-Benutzer kaum verändert werden kann. Oft lässt sich die Gesamtgröße der transferierten Daten, also der Dimensionswert *uniqueBytes* verändern. Weitere Parameter sind bei vielen Benchmarks nicht einstellbar.

statische Mehr-Punkt-Messung

Neben der Ein-Punkt-Messung tritt in der Praxis häufig der Fall der statischen Mehr-Punkt-Messung auf. Bei dieser Messmethode werden mehrere festgelegte I/O-Arbeitspunkte als Grundlage der Messung verwendet. Damit können verschiedene häufig verwendete Applikationen bei der statischen Mehr-Punkt-Messung abgedeckt werden. Der *b_eff_io* mit seiner großen Anzahl verschiedener Zugriffmuster kann in diese Kategorie eingeordnet werden. Benchmarks, die die Mehr-Punkt-Messung verwenden,

sollten in der Präsentation der Ergebnisse die Einzelergebnisse der jeweiligen Arbeitspunkte separat darstellen. Ein kumuliertes Gesamtergebnis verwischt die Einzelleistungen und trägt nicht zum Verständnis der Systemleistung bei. Es bietet sich ein Diagramm an, das die Leistung in Abhängigkeit der unterschiedlichen Arbeitspunkte darstellt, ähnlich wie es später in dieser Arbeit im Abschnitt 5.3.1 getan wird.

dynamische Punkt-Messung

Eine dritte punktbasierte Messmethode ist die dynamische Punkt-Messung, bei der Arbeitspunkte durch den Benchmarkentwickler nicht zur Entwicklungszeit des Benchmarks definiert werden. Vielmehr werden Messpunkte erst während der Anwendung z. B. durch den Nutzer selbst ausgewählt. Benchmarks, die eine kleine Anzahl von Arbeitspunkten stochastisch auswählen und dann die Leistung anhand dieser Punkte vermessen, zählen in diese Kategorie. Ebenso die applikationsbasierten Benchmarks, da der Nutzer bei ihnen die Arbeitspunkte selbst definieren kann.

I/O-Workloads punktbasierter Performance-messungen

Im Abschnitt 3.3.7 wurde eine Workload-Beschreibung vorgestellt, die den I/O-Workload von Applikationen abbilden kann und phasenweise die Last einer Applikation beschreibt. Wenn eine derartige Lastbeschreibung von dem Lastgenerator eines I/O-Benchmarks als Grundlage der Lastgeneration verwendet wird, kann dies als Basis für eine dynamische Punkt-Messung anhand von mehreren Punkten im I/O-Workload-Raum dienen. Die dynamische Punkt-Messung in Verbindung mit dem vorgestellten I/O-Lastmodell kann die I/O-Last beliebiger Applikationen also hervorragend nachbilden und wird deshalb im Weiteren in dieser Arbeit verwendet.

Die dynamische Punkt-Messung hat gegenüber den vorher beschriebenen Methoden den Vorteil des geringen Messaufwandes bei gleichzeitig hoher Genauigkeit der Messergebnisse. Allerdings hat sie den Nachteil, dass die Messergebnisse nur für die Applikation gelten, der der verwendete I/O-Workload entspricht. Außerdem benötigt der Anwender gute Kenntnis vom I/O-Workload der Applikation. Ein I/O-Benchmark, der also diesen Ansatz der Vermessung verwendet, muss dem Nutzer Tools zur Verfügung stellen, die bei der Erstellung von gültigen I/O-Workloads helfen, um so die Nutzbarkeit zu erhöhen.

3.4.4 Nutz- und Vergleichbarkeit

Um nutzerrelevante Messergebnisse in einem I/O-Benchmark zu erzielen, wird in dieser Arbeit der Weg vorgeschlagen, dem Nutzer die volle Freiheit bei der Definition eines I/O-Workloads für den Benchmark zu geben. Um den Nutzer bei der komplexen Konfiguration zu unterstützen, werden Tools zur Verfügung gestellt, die ihm diesen Vorgang weitestgehend abnehmen. Wenn jeder Nutzer die Möglichkeit hat, I/O-Workloads leicht und umfassend zu verändern, kann es bei dem Ergebnisvergleich unterschiedlicher Nutzer leicht zum Vergleich zwischen den sprichwörtlichen „Äpfeln und Birnen” kommen. Die Vergleichbarkeit unterschiedlicher Systeme kann bei einem

solchen Benchmark-System gefährdet sein. Um dennoch die Vergleichbarkeit der Messergebnisse bei Verwendung unterschiedlicher Rechensysteme zu ermöglichen, werden Standardkonfigurationen des Benchmarks definiert, die in solchen Fällen durch den Nutzer eingesetzt werden.

3.5 Zusammenfassung

In diesem Kapitel der vorliegenden Arbeit wurden Anforderungen an einen idealen I/O-Benchmark erarbeitet, bevor einzelne theoretische Grundlagen, die für die Architektur des Benchmarksystems von Bedeutung sind, vorgestellt wurden. Dazu wurde ein Lastmodell eingeführt, mit dem I/O-Lasten paralleler I/O-Systeme adäquat, aber dennoch einfach dargestellt werden. Anhand dieses Lastmodells wurden eingehend Messmethoden von I/O-Benchmarks klassifiziert und hinsichtlich der Anforderungen an einen idealen I/O-Benchmark bewertet.

Nachdem die theoretischen Grundlagen zur Beantwortung der drei in Abschnitt 2.4.3 formulierten Kernfragen erarbeitet wurden, folgt im Kapitel 4 eine beispielhafte Implementierung einer I/O-Benchmarking-Architektur, die die Kriterien Repräsentativität und Nutzerunterstützung erfüllt. Das erarbeitete Lastmodell bildet die Grundlage des Softwaresystems *PRIOmark*. Der *PRIOmark Parallel I/O Benchmark* enthält, neben einer Benchmarkkomponente mit Lastgenerator, Werkzeuge zur Erzeugung reeller Lastszenarien, die entsprechend der Kernfrage 3 nutzerrelevant sind.

Kapitel 4

Realisierung der PRIOmark-Toolchain

> ***Benchmark:*** *A measurement of a computer's ability to do what no one will ever want it to do. Including you.*
>
> – THE DEVIL'S IT DICTIONARY

Nachdem im Kapitel 3 systematisch Voraussetzungen einer I/O-Benchmarking-Architektur entwickelt wurden, die die Anforderungen an einen idealen I/O-Benchmark erfüllen kann, wird in diesem Kapitel eine Werkzeugsammlung vorgestellt, die eine entsprechende Benchmark-Architektur beispielhaft implementiert. Sie kann damit als idealer I/O-Benchmark nach den genannten Anforderungen bezeichnet werden. Die Werkzeugsammlung wird als *PRIOmark-Toolchain* (*Parallel I/O-Benchmark*) bezeichnet.

Nach einer kurzen Einleitung, in der der Kontext der Entwicklung des PRIOmark beschrieben wird, folgt im Absatz 4.2 die Beschreibung der Architektur der Werkzeugsammlung, während anschließend die einzelnen Werkzeuge vorgestellt werden. Dabei werden sowohl die Implementierung als auch die Nutzung kurz beschrieben. Der PRIOmark ist ein Benchmark, der dem Nutzer vollständige Freiheit bei der Definition von I/O-Lasten gewährt. Wie bereits in Abschnitt 3.4.4 beschrieben wurde, haben derartige Benchmarks den Nachteil, dass Ihre Ergebnisse unter Umständen nicht miteinander vergleichbar sind. In Absatz 4.4 wird eine Lösung für das Problem der Vergleichbarkeit erarbeitet.

4.1 Das IPACS-Projekt

Die Entwicklung des PRIOmarks begann im Rahmen des vom Deutschen Bundesministerium für Bildung und Forschung geförderten Projektes IPACS (Integrated Performance Analysis of Computer Systems). IPACS war ein Verbundprojekt von Unterneh-

Projektpartner

men, Hochschulen und privaten Forschungseinrichtungen. Projektpartner waren das *Fraunhofer Institut für Techno-und Wirtschaftsmathematik* aus Kaiserslautern, das Rechenzentrum der *Universität Mannheim* sowie der Lehrstuhl für Rechnerarchitektur der *Universität Rostock*, *T-Systems Enterprise Services GmbH* sowie in der ersten Projektphase auch die *Pallas GmbH*.

Ziel

Ziel des IPACS-Projektes war die Entwicklung eines integrierten Benchmarking-Systems, das nicht nur alle Teile moderner Hochleistungsrechensysteme benutzerfreundlich vermisst, sondern auch einen Vergleich der vermessenen Systeme online durchführen kann. Dazu wurde ein Online-Portal entwickelt, das in der Installation und Benutzung der im Projekt entwickelten Benchmarks Unterstützung leistet und einen automatischen Upload von Messdaten mit Vergleichs- und Auswertungsmöglichkeiten anbietet.

Kontext der Entwicklung

Der PRIOmark entstand im Kontext des Aufgabenspektrums der Universität Rostock, dessen Inhalt darin bestand, neue Methoden zur I/O-Leistungsmessung bei Hochleistungscomputern zu entwickeln [58, 92]. In Zusammenarbeit mit den Projektpartnern, die ihre Kompetenz bereits national und international durch einschlägige wissenschaftliche Arbeiten unter Beweis gestellt haben, wurden viele Grundlagen und Gedanken, die Teil dieser Arbeit sind, diskutiert und validiert.

4.2 Architektur der PRIOmark-Toolchain

Dieser Absatz beschäftigt sich mit dem Aufbau und der Nutzung der PRIOmark-Werkzeuge als Ganzes. Es wird vorgestellt, wie die einzelnen Werkzeuge zusammenarbeiten und wie der Nutzer die besten Messergebnisse hinsichtlich seiner Messziele erreicht.

Der PRIOmark ist ein applikationsbasierter Benchmark (vgl. 2.3.1), der den Nutzer in möglichst allen Phasen des Benchmarkings unterstützt (vgl. 3.2). Die Unterstützung des Nutzers ist eine wesentliche Anforderung an einen Benchmark, wie sie als Punkt 8 der Anforderungen an einen idealen Benchmark formuliert wurde (vgl. 2.3.2). Entsprechend besteht die PRIOmark-Toolchain aus Werkzeugen, die in jeder der Phasen II bis IV des Benchmarkings Unterstützung leisten.

Abbildung 4.1 zeigt die 5 Werkzeuge der PRIOmark-Toolchain zugeordnet zu der jeweiligen Phase, in der das Werkzeug verwendet wird.

Werkzeuge

Die drei Werkzeuge *I/O-Profiler*, *Profile-Analyzer* und *Workload-Definition-Tool* dienen der Erstellung von I/O-Lastmustern, unterstützen also in der Phase II des Benchmarklaufes. Die erzeugten Lastmuster, oder Workload-Definition-Files werden in der dritten Phase vom *PRIOmark-I/O-Benchmark*, dem eigentlichen I/O-Benchmark verarbeitet. Dieser verwendet die erstellten I/O-Lastbeschreibungen als Messgrundlage und ermittelt die I/O-Leistung des zu vermessenden Systems bei Verwendung der entsprechenden I/O-Lastmuster. Die so entstehenden Ergebnisse werden vom Benutzer

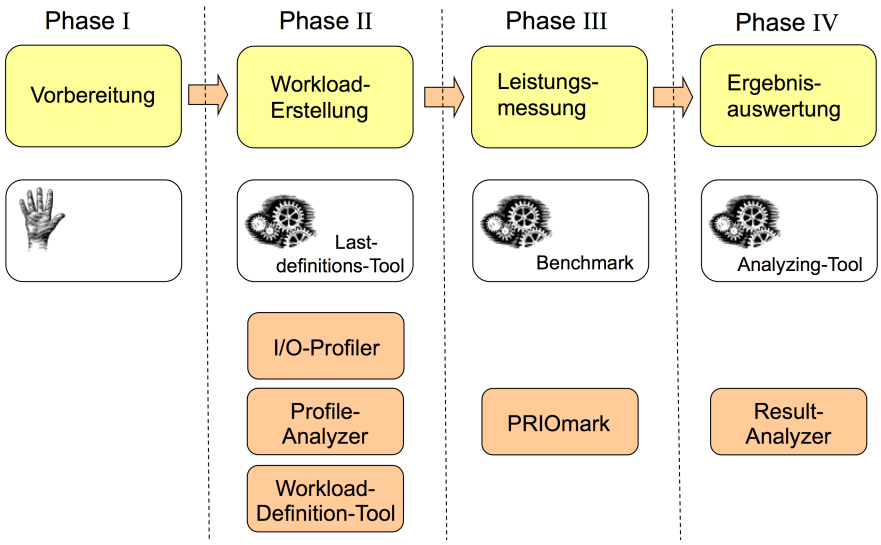

Abbildung 4.1: Zuordnung der Werkzeuge der PRIOmark-Werkzeugsammlung zu den Benchmarkingphasen

im *Result-Analyzer* betrachtet und ausgewertet. Etwas detaillierter wird die Architektur der PRIOmark-Werkzeugsammlung in Abbildung 4.2 dargestellt. Dort ist erkennbar, dass das *Workload-Definition-Tool* und der *I/O-Profiler* in Zusammenhang mit dem *Profile-Analyzer* zwei grundsätzlich unterschiedliche Möglichkeiten der Lastdefinition für den Benchmark zur Verfügung stellen. Während der *I/O-Profiler* die Last einer reellen Applikation ermittelt, indem ihr I/O-Verhalten während der Ausführung analysiert wird, steht mit dem *Workload-Definition-Tool* ein Werkzeug zur Verfügung, das dem Nutzer mittels grafischer Eingabemöglichkeiten die Definition von I/O-Lasten nach dem in Kapitel 3 eingeführten I/O-Workload-Modell erlaubt. Der *Profile-Analyzer* dient der Abbildung von feingranularen I/O-Profilen auf I/O-Workload-Beschreibungen mit durch den Nutzer definierbarer Granularität. Detaillierter werden die einzelnen Werkzeuge in den folgenden Abschnitten vorgestellt.

4.3 Vorstellung der Werkzeuge

Dieser Abschnitt stellt die einzelnen Werkzeuge der PRIOmark-Toolchain vor. Es wird dabei in chronologischer Reihenfolge der Benchmarkphasen sowohl auf die Architektur als auch auf die Nutzung der entsprechenden Werkzeuge eingegangen. Da Phase I eine manuelle Bearbeitung erfordert, wird mit den Werkzeugen der Phase II begonnen: I/O-Profiler, Profile-Analyzer und Workload-Definition-Tool.

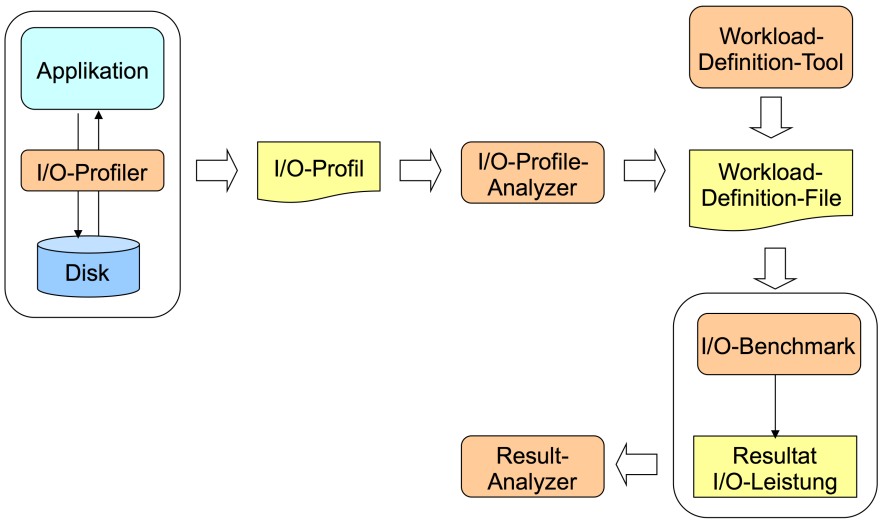

Abbildung 4.2: Architektur der PRIOmark-Toolchain

4.3.1 I/O-Profiler

Der I/O-Profiler ist ein Werkzeug zur Erstellung von I/O-Workloads, die aus einem beispielhaften Applikationslauf extrahiert werden. Beim Start einer Applikation wird der Profiler dynamisch an alle Prozesse der Applikation gebunden und protokolliert während des Applikationslaufes alle System- bzw. Bibliotheksaufrufe der einzelnen Prozesse, die Zugriffe auf den Sekundärspeicher betreffen. Das entstehende Protokoll, das I/O-Profil genannt wird, ist ein genaues Abbild der I/O-Tätigkeiten der Applikation während des protokollierten Applikationslaufes. Wenn angenommen wird, dass sich das I/O-Verhalten der Applikation zwischen einzelnen Applikationsläufen nur unwesentlich unterscheidet, spiegelt das Protokoll das I/O-Lastverhalten der Applikation wieder.

PMPI

SvPablo

Es gibt eine Reihe Profiling-Systeme für MPI auf dem Markt. Der MPI-Standard selbst definiert mittels der PMPI-Schnittstelle ein Profiling-Interface für MPI. Dabei gibt es für jede MPI-Funktion eine korrespondierende PMPI-Funktion, die ein Profiling durchführt und anschließend die Originalfunktionalität der MPI-Funktion aufruft [30]. Ein weiteres bekanntes Beispiel ist das *SvPablo Performance Analysis Tool* der University of Illinois [93]. Sowohl PMPI als auch SvPablo ermöglichen ein sehr detailliertes Profiling von MPI-Applikationen nach einer identischen Methode. Der Quellcode der Applikation wird instrumentarisiert, um ein Profiling zu unterstützen. Das heißt, es werden Funktionsaufrufe in den Quellcode der Applikation eingefügt, die das Profiling durchführen. Die Applikation wird schließlich gegen eine Profiling-Bibliothek gelinkt. Ein wesentlicher Nachteil dieser Methode wird sofort deutlich: Um ein Profiling zu ermöglichen, muss dem Anwender der Quellcode der Applikation zur Verfügung stehen. Bei einigen Tools muss er außerdem Programmiererfahrung haben, um den Vorgang der Instrumentarisierung durchführen zu können. Da nach der Instrumentarisierung eine erneute Übersetzung der Applikation und dem Linken mit der Profiling-Biblio-

thek erforderlich ist, wird auch die Existenz einer kompletten Entwicklungsumgebung durch die Werkzeuge vorausgesetzt. All dies geht über die Möglichkeiten vieler normaler Nutzer hinaus. Die Profiling-Werkzeuge haben offensichtlich Entwickler als Zielgruppe, die mit den Werkzeugen Leistungssteigerungen in ihren Applikationen erzielen sollen. Eine Leistungsermittlung und -optimierung der Laufzeitumgebung oder von Teilen derselben durch Systemadministratoren und andere Nutzer wird mit den vorgestellten Profiling-Systeme nicht fokussiert. Aber auch dies muss ermöglicht werden, um eine optimale Ausnutzung aller Rechnerressourcen zu gewährleisten.

Im Rahmen dieser Arbeit wurde deshalb ein I/O-Profiler entwickelt, der weniger auf die Performance-Optimierung von Applikationen zielt, sondern eher auf deren Leistungsermittlung und Performance-Optimierung ihrer Laufzeitumgebung. Die wesentlichen Anforderungen des Profilers sind also:

- Es soll auch ein Profiling von Applikationen ermöglicht werden, die nicht im Quellcode zur Verfügung stehen.
- Anwender sollen mit wenig Wissen und insbesondere ohne Einsatz spezieller Werkzeuge, wie Entwicklungstools (Compiler, Linker ...) mit dem Profiler arbeiten.
- Andererseits soll der Profiler ausschließlich das Protokollieren von I/O-Aufrufen ermöglichen, was die Komplexität gegenüber generischen Profilern, wie den beiden vorgestellten, verringert.

In Abschnitt 2.1.2 wurden bereits die beiden I/O-Schnittstellen POSIX-I/O und MPI-IO als wichtige I/O-Schnittstellen vorgestellt. Da MPI-IO die komplexere Schnittstelle darstellt, wurde im Rahmen dieser Arbeit ein Profiler für MPI-IO implementiert. Die verwendeten Konzepte sind bei beiden Schnittstellen identisch, weshalb die Implementierung eines POSIX-I/O-Profilers ähnlich (und an vielen Stellen einfacher) zu der hier vorgestellten MPI-IO-Implementierung ist. Aufgrund der hohen Komplexität der MPI-IO-Schnittstelle sind die verwendeten Konzepte durch Vereinfachung leicht auch auf beliebige weitere I/O-Schnittstellen übertragbar.

Funktionsweise des Profilers

Abbildung 4.3 zeigt die Funktionsweise des I/O-Profilers. Ähnlich wie bei den beschriebenen Profilern, wurde auch der im Rahmen dieser Arbeit entwickelte Profiler als Bibliothek entwickelt, welche Funktionswrapper für alle I/O-Funktionen der I/O-Schnittstelle enthält. Der wesentliche Unterschied ist die Implementierung als dynamisch ladbare Bibliothek (shared library). Während herkömmliche Profiling-Systeme beim Kompilationsvorgang statisch zum Programm hinzugelinkt werden und damit nicht transparent für den Nutzer zu verwenden sind, kann bei der hier verwendeten Architektur das bereits kompilierte Originalprogramm ohne jede Änderung verwendet werden, um ein I/O-Profil zu erstellen. Dem dynamischen Linker wird die dynamische

Profiling-Bibliothek als reguläre I/O-Bibliothek vorgetäuscht. Der Linker verbindet also das Programm zum Start nicht mit der regulären I/O-Bibliothek, sondern mit dem Profiler, so dass der Profiling-Vorgang beim anschließenden Applikationslauf durchgeführt werden kann. Dieser Mechanismus funktioniert allerdings nur dann, wenn die Applikation selbst den dynamischen Linker verwendet, um die I/O-Bibliothek hinzuzulinken. Wurde sie statisch während der Kompilation gelinkt, ist ein Profiling in der beschriebenen Form nicht möglich. In der Praxis sind insbesondere kommerzielle Applikationen gelegentlich statisch gegen sämtliche zur Ausführung benötigten Bibliotheken gelinkt, um eine möglichst große Unabhängigkeit von der Ausführungsumgebung zu erlangen. Die MPI-IO-Bibliothek (bzw. insgesamt die MPI-Bibliotheken) werden in solchen Fällen vom Nutzer vor der Programmausführung statisch hinzugelinkt, da MPI-Bibliotheken für die spezifische Ausführungsumgebung optimiert sind und nicht durch den Hersteller der Applikationssoftware mitgeliefert werden (bspw. sollte MPI-IO eine einkompilierte PVFS-Unterstützung bei Nutzung eines Parallel Virtual Filesystems auf der Plattform zur Verfügung stellen). In diesen Fällen kann der Nutzer die Applikation gegen die Profiling-Bibliothek statisch linken und muss zusätzlich eine dynamisch ladbare I/O-Bibliothek zur Verfügung stellen. Der Profiler ist also in beiden Fällen, die die häufigsten Einsatzfälle darstellen, problemlos einsetzbar.

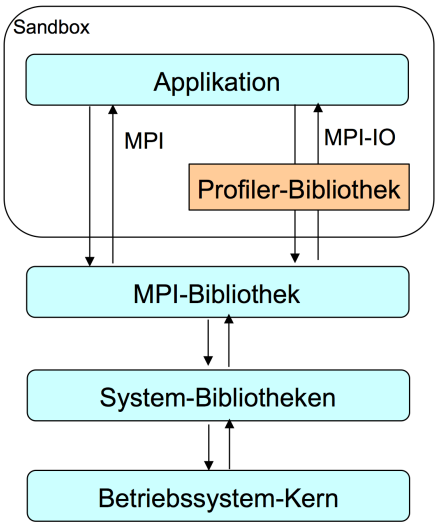

Abbildung 4.3: Funktionsweise des I/O-Profilers

Sandbox

Zur Ausführung des Profilers muss die Applikation in einer Umgebung gestartet werden, die ihr die Profiling-Bibliotheken als Standardbibliotheken zum Nachladen zur Verfügung stellt. Diese Umgebung, im Folgenden Sandbox genannt, wird automatisch beim Profiling-Vorgang durch das Profiling-Werkzeug erstellt. Neben der Eigenschaft, dass der dynamische Linker in der Sandbox geeignet umkonfiguriert wird, müssen Schnittstellen geschaffen werden, die der Profiling-Bibliothek innerhalb der Sandbox einen Zugriff auf die Standardfunktionalität der Bibliotheksfunktionen außerhalb der Sandbox ermöglichen. Nur so kann erreicht werden, dass die Profiling-Bibliothek neben dem Vorgang der Profilerstellung auch die eigentliche Funktionalität der aufgeru-

fenen Funktionen realisieren kann. Der implementierte Profiler ermittelt die Position der Standardbibliotheken im Dateisystem und übergibt diese der Profiling-Bibliothek. Mit dieser Information kann die Profiling-Bibliothek die Originalbibliothek manuell hinzulinken und trotz des umkonfigurierten dynamischen Linkers auf die Originalfunktionalität zugreifen, um der Applikation transparent die Funktionalität der aufgerufenen Funktionen zur Verfügung zu stellen [94]. Aufgrund der parallelen Ausführung mehrerer Prozesse bei Applikationen für Hochleistungsrechensysteme muss ein Profiler für MPI-IO sämtliche Prozesse einer Applikation überwachen. Das in dieser Arbeit entwickelte Werkzeug erstellt deshalb pro Prozess eine Sandbox, so dass eine Protokollierung sämtlicher I/O-Aufrufe aller Prozesse ermöglicht wird. Dieses Protokoll der I/O-Aufrufe wird I/O-Profil genannt und im folgenden Absatz vorgestellt.

I/O-Profil

Das I/O-Profil, das das gesamte Applikations-I/O-Verhalten beschreibt, wird durch den Profiler erzeugt und als Datei im Dateisystem abgelegt. Die Darstellungsform des Profils ist dabei feingranular, d. h. jeder I/O-Funktionsaufruf der Applikation wird vollständig mit sämtlichen Parameterwerten protokolliert. Außerdem werden Ausführungszeiten, also die Zeit des Aufrufes einer Funktion und die Zeit der Rückkehr aus einer Funktion vermerkt. Da jeder Prozess einer Applikation ein individuelles Profil des entsprechenden I/O-Verhaltens generiert, müssen diese Einzelprofile vom Profiler zu einer Profildatei kombiniert werden. Prinzipiell gibt es zwei Möglichkeiten, dies zu tun. I/O-Aufrufe können chronologisch sortiert mit der entsprechenden Prozessnummer gespeichert werden, oder die Einzelprofile werden hintereinander in einer Datei abgelegt. Im Rahmen dieser Arbeit wurde die zweite Möglichkeit verwendet, um bei der Weiterverarbeitung des Profils eine Aufteilung nach Prozessen zu ermöglichen.

XML zur Speicherung CSV-Datei

Es bieten sich verschiedene Möglichkeiten an, die Protokolldaten im Dateisystem abzulegen. XML (Extensible Markup Language) [95] als standardisierte Datenspeicherungsmöglichkeit käme in Frage, wurde aber in der aktuellen Implementierung nicht verwendet. Derzeit wird ein CSV-Format (Comma Separated Value) verwendet, bei dem die protokollierten Daten mittels Kommata getrennt in der Datei abgelegt werden [96]. Zukünftige Versionen des Profilers werden aufgrund der leichteren Verarbeitung und starken Verbreitung XML zur Speicherung der Daten verwenden [14].

Das I/O-Profil einer Applikation ist eine hochgenaue, feingranulare Darstellungsmöglichkeit des Applikations-I/O-Verhaltens. Allerdings ist es groß und schwer zu analysieren. Der Profile-Analyzer, der im Folgenden vorgestellt wird, konvertiert das I/O-Profil einer Applikation in eine sogenannte Workload-Definitions-Datei, die definierbare Teile des I/O-Profils übersichtlich zusammenfasst.

4.3.2 Profile-Analyzer

I/O-Profile, wie sie vom PRIOmark-I/O-Profiler erzeugt werden, haben die höchstmögliche Genauigkeit. Sie speichern sämtliche Parameter einer jeden I/O-Anfrage und können deshalb sehr groß und schwer verständlich werden. Im Abschnitt 3.3.7 wurde eine Lastbeschreibung für I/O vorgestellt, die leichter handhabbar ist und dennoch eine genaue Abbildung von I/O-Verhalten paralleler Applikationen ermöglicht. Es ist die Aufgabe des Profile-Analyzers, I/O-Profile in eine derartige Lastdefinition umzuwandeln. Dazu müssen mehrere I/O-Anfragen der Profile zu I/O-Arbeitspunkten zusammengefasst werden. Computerlesbar werden diese Lastdefinitionen in Workload-Definitions-Dateien gespeichert. Diese Dateien sind XML-basiert und werden vom I/O-Benchmark verarbeitet. Im Folgenden werden Workload-Definitions-Dateien genauer beschrieben [95].

Workload-Definitions-Datei

Workload-Definitions-Dateien bilden alle Merkmale der in Abschnitt 3.3.7 beschriebenen Applikationslasten ab. Eine Workload-Beschreibung besteht aus Operationen, Metaoperationen und Mengen von Operationen, die als `operation` bzw. Operationskollektionen (`operationCollection`) bezeichnet werden. Operationen bilden I/O-Arbeitspunkte ab und besitzen somit für jede der neun Dimensionen des Workload-Raumes jeweils ein gleichnamiges Element, das den Wert des entsprechenden Parameters beschreibt. Im Folgenden wird ein I/O-Arbeitspunkt deshalb in der Workload-Definitions-Datei als (I/O-)Operation bezeichnet. Neben den beschriebenen neun Dimensionen besitzt die Workload-Definitions-Datei die Möglichkeit, einen Speicherort, bezeichnet mit dem Parameter `storagePlace` zu definieren. Bei MPI-IO wird jeder Speicherort auf eine separate Datei abgebildet. Der Speicherort ist eine Erweiterung des Lastmodells um die Möglichkeit der Dateiverarbeitung. Der Parameter `storagePlace` wurde eingeführt, weil bei einigen Anwendern die Anforderung besteht, die Leistung sogenannter Metaoperationen des Dateisystems zu vermessen. Metaoperationen sind Operationen, die sich nicht auf den Inhalt der Datei, sondern auf ihre Metainformationen beziehen. So ist zum Beispiel das Ändern des Dateinamens ebenso eine Metaoperation wie das Öffnen, Schließen oder Löschen einer Datei. Metaoperationen lassen sich nicht im 9-dimensionalen Workload-Raum abbilden, da bei ihnen keiner der neun Dimensionsparameter einen definierten Wert besitzt. Sie werden deshalb neben den Operationen und Operationskollektionen als eigenes Element in der Workload-Definitions-Beschreibung eingeführt. Metaoperationen sind normalerweise nicht leistungsrelevant, da sie vergleichsweise selten ausgeführt werden (Dateien werden oftmals nur einmal geöffnet und geschlossen, während zwischen beiden Operationen zahlreiche Lese- und Schreiboperationen ausgeführt werden). Aus diesem Grund wurden sie im Lastmodell ignoriert. Tatsächlich kam die Anforderung der Unterstützung von Metaoperationen aus der Praxis, da ein Nutzer eines Rechensystems ein fehlerhaft konfiguriertes Dateisystem verwendete, bei dem jede Metaoperation mehrere

storagePlace

Metaoperationen

Sekunden benötigte und damit wirklich leistungsrelevant wurde. Die Unterstützung von Metaoperationen in der beschriebenen Art existiert in allen PRIOmark-Werkzeugen, wird aber in dieser Arbeit nicht weiter thematisiert.

Kollektionen von Operationen sind Mengen von Operationen, Metaoperationen oder eingeschlossenen Kollektionen und haben den Zweck, die Anzahl von Iterationen der eingeschlossenen Elemente mittels des Attributs `iteration` zu definieren. Auf diese Weise wird das `LOOP`-Konstrukt der formalen Lastbeschreibung realisiert.

Umwandlung von I/O-Profilen in Workload-Definitions-Dateien

Der Umwandlungsvorgang, der eine abstrakte und leichter verarbeitbare und verständliche Workload-Definition aus I/O-Profilen erzeugt, ist vollautomatisch nicht realisierbar, da für den Vorgang Anforderungen des Nutzers und bestimmtes Wissen über den Ablauf der Applikation erforderlich sind. Bei der Umwandlung werden mehrere I/O-Anforderungen zu einer I/O-Operation zusammengefasst. Es werden durch das Werkzeug die Werte der einzelnen Dimensionen der I/O-Operation berechnet und in einer XML-Datenstruktur gespeichert. Die Anzahl der I/O-Anforderungen, die zu einer Operation zusammengefasst werden, bestimmt die sogenannte Granularität der Beschreibung. Eine feingranulare Beschreibung wandelt jede I/O-Anforderung in eine I/O-Operation, während eine sehr abstrakte Lastbeschreibung alle I/O-Anforderungen zu einer I/O-Operation zusammenfasst. Diese Granularität der Zielbeschreibung ist von den Anforderungen des Nutzers abhängig. Entsprechend besitzt das Profile-Analyzer-Tool Möglichkeiten für den Nutzer, die Zielgranularität der Beschreibung zu definieren. Dazu legt der Nutzer Bedingungen fest, die die einzelnen Phasen des Applikationslaufes voneinander abgrenzen. Der Profile-Analyzer erlaubt die Definition von vier Bedingungsarten zur Abgrenzung der Applikationsphasen:

- zeitliche Bedingungen
- request-bezogene Bedingungen
- indexbezogene Bedingungen
- Verknüpfungen von Bedingungen

zeitliche Bedingungen

Zeitliche Bedingungen besagen, dass alle I/O-Anfragen innerhalb bestimmter Zeitspannen einer I/O-Operation zuzuweisen sind. So kann beispielsweise definiert werden, dass alle I/O-Anforderungen innerhalb der ersten Minute des Applikationslaufes einer I/O-Operation, die die Initialisierung der Applikation abbildet, zugeordnet werden. Da das I/O-Profil die Startzeiten der Ausführung einzelner I/O-Anforderungen abspeichert, ist eine derartige Realisierung ausschließlich mit Hilfe des I/O-Profils möglich.

request-bezogene Bedingungen

Request-bezogene Bedingungen verwenden das Vorkommen von bestimmten I/O-Anforderungen als Kriterium für die Zusammenfassung von I/O-Anfragen zu Operationen. Die Initialisierungsphase einer Applikation kann so genauer als mit einer zeitlichen Bedingung abgegrenzt werden, in dem zum Beispiel die Bedingung definiert wird, dass alle I/O-Anfragen vor dem Schließen der Konfigurationsdatei `config.cfg` der Initialisierungsphase zugeordnet werden.

indexbezogene Bedingungen

Indexbezogene Bedingungen sind die einfachste Möglichkeit, die Granularität eines I/O-Profils zu verringern. Mittels dieser Variante kann eine feste Anzahl von I/O-Anforderungen zu einer I/O-Operation zusammengefasst werden. Als Bedingung wird der Anzahl der zusammenzufassenden I/O-Anforderungen angegeben. Nach der Zusammenfassung der entsprechenden Anzahl von I/O-Anforderungen wird der Zählindex zurückgesetzt, um eine erneute Zusammenfassung von Requests nach der definierten Anzahl von Anforderungen durchzuführen.

Verknüpfungen von Bedingungen

Verknüpfungen ermöglichen auch bei komplexeren Bedingungen eine genaue Definition der Applikationsphasenabgrenzungen. So kann das Ende der Initialisierungsphase definiert werden, in dem entweder die Datei `config.cfg` geschlossen wird oder eine Minute Laufzeit vergangen ist (bspw. da sie aufgrund eines Fehlers nie gelesen werden konnte). Bei disjunktiven Verknüpfungen, wie der gerade beispielhaft beschriebenen, gilt die Bedingung als erfüllt, sobald eine der Teilbedingungen erfüllt wurde. Bei konjunktiven Bedingungen hingegen müssen beide Teilbedingungen erfüllt worden sein. Wenn die im Beispiel genannten beiden Teilbedingungen konjunktiv miteinander verknüpft werden, gilt die Bedingung also nur als erfüllt, wenn die Konfigurationsdatei geschlossen und die Applikation mehr als eine Minute gelaufen wäre. Neben disjunktiver und konjunktiver Verknüpfung muss die Negation einer Bedingung möglich sein, um beliebige Verknüpfungen von Bedingungen definieren zu können.

Die aktuelle Version des Profile-Analyzers implementiert nur die indexbezogenen Bedingungen, um die Erzeugung unterschiedlicher Granularitäten derselben Lastdefinition für weitere Untersuchungen zu ermöglichen.

Die genaue Definition von Phasen benötigt also durch den Nutzer einiges Wissen über die Applikation. Eine einfache Phasendefinition mittels der indexbasierten Bedingungen ist aber dennoch möglich, um schnell eine Zusammenfassung von I/O-Requests zu einem übersichtlichen I/O-Workload zu ermöglichen.

Wenn keine Applikation als Grundlage der Lastdefinition zur Verfügung steht, dennoch aber Kenntnis über die I/O-Last der Applikation existiert, kann das Workload-Definition-File über eine zweite Möglichkeit, dem Workload-Definition-Tool erstellt werden.

4.3.3 Workload-Definition-Tool

Während I/O-Profiler und Profile-Analyzer zur automatischen Erstellung von I/O-Workloads aus dem Applikationsverhalten verwendet werden, gibt es für den Nutzer eine weitere Möglichkeit, I/O-Workloads völlig nach eigenen Vorstellungen zu definieren. Dazu dient das Workload-Definition-Tool. Es handelt sich dabei um ein grafisches Werkzeug, das in Java entwickelt, dem Nutzer eine grafische Benutzeroberfläche anbietet, mit der komfortabel Lasten nach der in Absatz 3.3.7 beschriebenen I/O-Lastbeschreibung definiert werden. Gespeichert werden die Lastbeschreibung im gleichen XML-Format, wie die vom Profile-Analyzer erzeugten I/O-Lasten.

Das Workload-Definition-Tool verwendet die in Abschnitt 3.3.7 eingeführte grafische Darstellungsart von Lasten, die sich an der UML2 orientiert. I/O-Operationen werden als Zustände realisiert und Schleifen als Rückkopplungen der Zustandsübergänge mit entsprechend formulierten Übergangsbedingungen. Die Werte der einzelnen I/O-Dimensionen stehen textuell in den Zuständen.

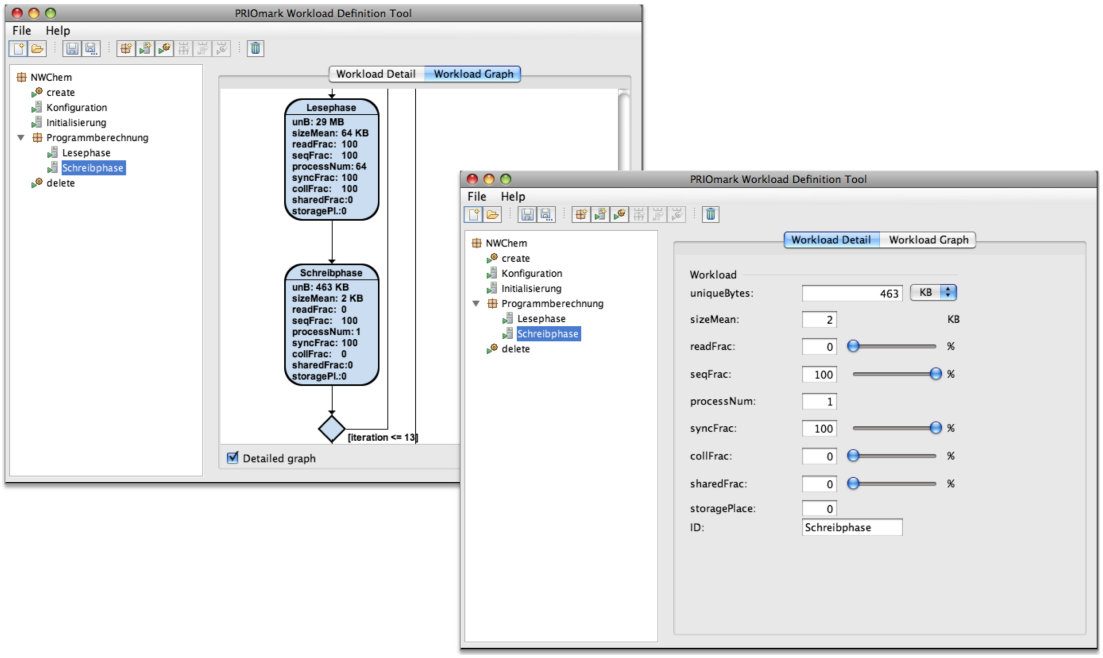

Abbildung 4.4: Screenshots des Workload-Definition-Tools

Abbildung 4.4 zeigt zwei Screenshots des Workload-Definition-Tools. Die linke Grafik stellt einen I/O-Workload in einer Baumansicht und in der beschriebenen an Zustandsdiagramme angelehnten Ansicht dar. In der Baumbeschreibung werden Operationen als Blätter dargestellt. Iterationen werden mittels der Zusammenfassung von Operationen zu *OperationCollections* realisiert, die als innere Knoten dargestellt werden. Zur übersichtlichen Manipulation der Dimensionswerte gibt es eine weitere Eingabemaske, die im rechten Screenshot abgebildet ist. Hier werden die definierten Lastdimensionen durch Schieberegler oder durch Zahlenwerte verändert [97].

Ziel der Phase II des I/O-Benchmarkings ist die Erstellung einer Lastbeschreibung, die als Grundlage der Vermessung in der dritten Phase der Leistungsanalyse eingesetzt wird. Das in der dritten Phase verwendete Werkzeug ist der I/O-Benchmark, der im Folgenden vorgestellt wird.

4.3.4 I/O-Benchmark

Zentrales Element der PRIOmark-Werkzeugsammlung ist der PRIOmark I/O-Benchmark. Seine Funktion ist die eigentliche Leistungsmessung. Er ist damit das Werkzeug zur Durchführung der Phase III des beschriebenen Benchmarklaufes. Wie in Abschnitt 3.2.3 dargelegt wurde, besteht jeder I/O-Benchmark aus einem Lastgenerator und einem Messwerkzeug. Die erweiterbare Architektur des im Rahmen der Arbeit entwickelten Benchmarksystems erlaubt die Implementierung verschiedener Lastgeneratoren, um die Messkomponente (die für einige Architekturen eine hochgenaue Zeitmessung auf der Basis von Prozessortakten ermöglicht) auch für andere Benchmarks als einem I/O-Benchmark nutzbar zu machen [98]. Der PRIOmark in der neuesten Version implementiert allerdings nur einen Lastgenerator, der Workloads in Form der beschriebenen XML-I/O-Lastbeschreibung verarbeitet. In früheren Versionen wurde verschiedene festdefinierte Lastszenarien erzeugt [99].

erweiterbare Architektur

Der Profile-Analyzer führt eine Abbildung von I/O-Anforderungen auf I/O-Operationen im I/O-Workload-Raum durch. Diese Abbildung ist mit einem Informationsverlust verbunden, wenn die Granularität der Zielbeschreibung sehr gering eingestellt wird. Der Lastgenerator des I/O-Benchmarks muss eine Abbildung der I/O-Workload-Beschreibung auf ein reales Lastverhalten durchführen, das dem Original so ähnlich wie möglich ist. Um die Ungenauigkeiten der Abbildung zu finden und zu diskutieren, wird an dieser Stelle der Algorithmus zur Erzeugung einer realen I/O-Last aus der I/O-Workload-Beschreibung vorgestellt.

Erzeugung der I/O-Last

Ausgangspunkt der Lasterzeugung im Lastgenerator ist eine Lastbeschreibung in Form einer Workload-Definitions-Datei. Diese beschreibt Operationen, Metaoperationen und Operationskollektionen in einer bestimmten Reihenfolge.

Meta-operationen

Metaoperationen werden dabei direkt auf entsprechende Dateisystemoperationen abgebildet.

Operations-kollektionen

Eine Operationskollektion ist eine Menge von Operationen, Metaoperationen und anderen Operationskollektionen. Die Kollektion schließt andere Elemente mittels der XML-Tags `<operationCollection>` und `</operationCollection>` ein,

besitzt selbst aber nur einen Parameter, die Iterationsanzahl. So werden alle Elemente in einer Operationskollektion entsprechend der Iterationsanzahl wiederholt, um das `LOOP`-Konstrukt der Lastbeschreibung zu realisieren.

Operationen

Operationen sind das zentrale Element der Lastbeschreibung. Jede Operation wird durch die im Lastmodell beschriebenen Parameter definiert und entspricht mehreren I/O-Anforderungen mit bestimmten Eigenschaften. Die Sequenz von I/O-Anforderun-

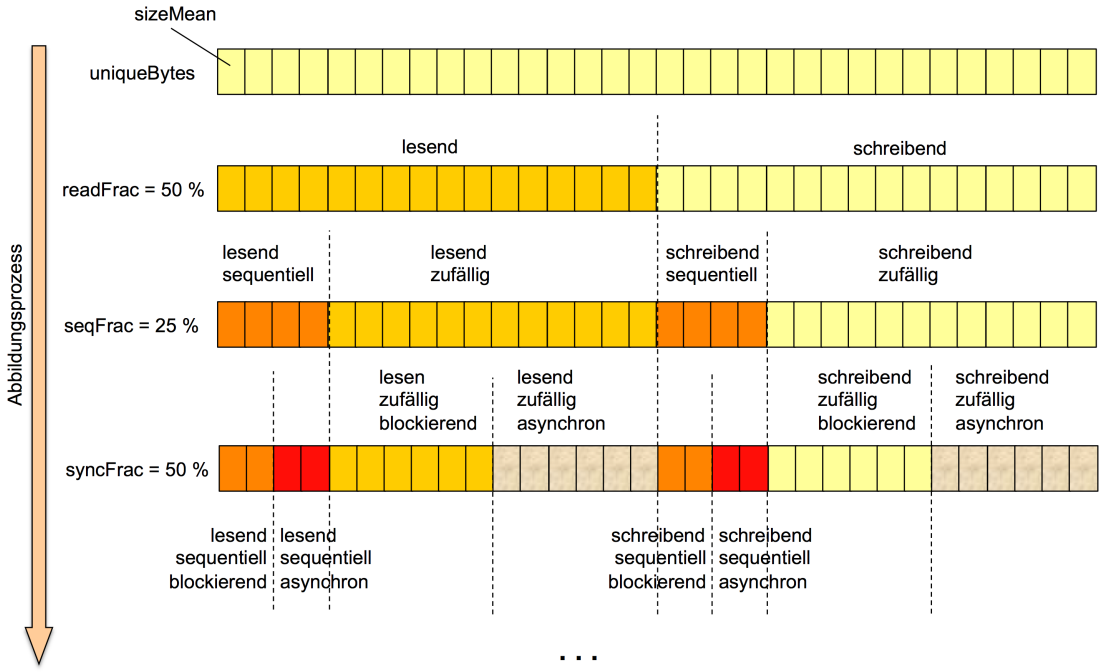

Abbildung 4.5: Vorgang der Lasterzeugung im PRIOmark Lastgenerator

gen, die einer I/O-Operation entspricht, wird nach dem in Abbildung 4.5 dargestellten Verfahren ermittelt. Die durch *uniqueBytes* angegebene Transfermenge der Operation wird in $\lfloor \frac{\text{uniqueBytes}}{\text{sizeMean}} \rfloor$ Anforderungen mit jeweils der Größe *sizeMean* aufgeteilt. In jedem der *processNum* Prozesse werden den einzelnen Anforderungen anschließend Anforderungstypen zugeordnet. Der Parameter *readFrac* gibt den Anteil aller lesenden Anforderungen an, *seqFrac* den Anteil der sequentiell durchgeführten Zugriffe sowohl bei den lesenden, als auch bei den schreibenden Zugriffen. Dieses Verfahren wird mit den Parametern *syncFrac* (blockierende Zugriffe), *collFrac* (kollektive Zugriffe), und *sharedFrac* (Zugriffe mit einem gemeinsamen Dateizeiger) fortgesetzt, so dass schließlich $2^5 = 32$ Anforderungstypen entstehen, denen jeweils eine Anzahl an Anforderungen zugeordnet wird. Der Parameter *contMean* bestimmt, ob Zugriffe verschränkt oder zusammenhängend durchgeführt werden. Ist *contMean* kleiner als die Größe einer Anforderung *sizeMean*, muss eine Dateisicht erzeugt werden, die zyklisch jedem Prozess *contMean* Bytes innerhalb einer Dateieinheit zuordnet. Es kann dabei sowohl passieren, dass am Ende der Dateieinheit Lücken entstehen, als auch, dass nicht allen Prozessen ein Teil der Zugriffseinheit zugeordnet wird. In dem Fall erhalten die noch nicht zugeordneten Prozesse Teile der nächsten Dateieinheit, so dass

unter Umständen die Erstellung von mehreren Sichten notwendig ist. In dem Fall, dass *contMean* einen größeren Wert besitzt, als die Dateieinheit groß ist, wird jeder Prozess eine Sicht erhalten, die ihm die gesamte Dateieinheit zuordnet. Allerdings fangen alle Prozesse an unterschiedlichen Bereichen der Datei mit dem Zugriff an, so dass jedem Prozess größere Dateiteile zugeordnet sind. Auf diese Weise können sowohl verschränkte als auch nicht-verschränkte Zugriffe realisiert werden.

Der PRIOmark ermöglicht aktuell nicht alle Arten der beschriebenen Sichtdefinitionen. Der Benchmark realisiert verschränkte Zugriffe, bei denen jedem Prozess ein gleicher Anteil der Dateieinheit zugeordnet wird. Dazu wird die Größe der Dateieinheit durch die Anzahl der Prozesse dividiert und eine entsprechende Anzahl an Bytes jedem Prozess zugeordnet. Diese Sicht ist typisch für HPC-Applikationen, die beispielsweise mit großen Matrizen rechnen. Lasten, die unverschränkte Zugriffe durchführen, werden deshalb aktuell bei der Nutzung von mehr als einem Prozess nicht korrekt wiedergegeben.

Tabelle 4.1 auf Seite 111 zeigt, welche MPI-IO-Funktionen jedem Anforderungstyp zugeordnet werden. Bei einigen Anforderungstypen handelt es sich um eine Sequenz von MPI-IO-Funktionen. Kollektive, nicht-blockierende MPI-IO-Funktionen bestehen immer aus einem Paar von Funktionen, wie dies bereits in Tabelle 3.3 auf Seite 70 zu sehen war. Ebenso werden einige nicht-sequentielle Zugriffe mit einem `MPI_File_seek_shared()` zum Versetzen des gemeinsamen Dateizeigers und der folgenden Datenzugriffsfunktion realisiert.

Reihenfolge

Da eine einzelne I/O-Operation zwar die Zusammenfassung mehrerer I/O-Anforderungen darstellt, aber sämtliche Reihenfolgeinformation durch den Prozess der Zusammenfassung verloren geht, kann die Abfolge der Wiedergabe der I/O-Anforderungen durch den Lastgenerator nur heuristisch festgelegt werden. Sie werden vom PRIOmark in einer immer gleichen Reihenfolge ausgeführt, wobei I/O-Anforderungen gleichen Typs hintereinander folgen. Für die Anforderungstypen wurde eine Reihenfolge gewählt, die einen regelmäßigen Wechsel der Zugriffsarten ermöglicht (es werden z. B. nicht alle lesenden Zugriffe gefolgt von allen schreibenden ausgeführt, sondern ein regelmäßiger Wechsel beider Zugriffsarten). Diese vom Lastgenerator ermittelte Abfolge der I/O-Anforderungen kann sich zweifellos von der Originallast unterscheiden. Es gibt an dieser Stelle also eine Ungenauigkeit beim Abbildungsvorgang, die die Messergebnisse verfälschen kann. Diese und weitere Ungenauigkeiten werden im folgenden Abschnitt genauer beleuchtet.

Ungenaugkeiten in der Abbildung

In Abschnitt 3.3.9 wurden bereits Abbildungsprobleme der Lastbeschreibung an sich erörtert. Diese Abbildungsprobleme erzeugen Ungenauigkeiten bei der Lasterzeugung,

lesend	sequentiell	blockierend	kollektiv	gem. Dat.-Zeig.	MPI-IO-Funktionen
-	-	-	-	-	MPI_File_iwrite_at()
-	-	-	-	✓	MPI_File_seek_shared() MPI_File_iwrite_shared()
-	-	-	✓	-	MPI_File_write_at_all_begin() MPI_File_write_at_all_end()
-	-	-	✓	✓	MPI_File_seek_shared() MPI_File_write_ordered_begin() MPI_File_write_ordered_end()
-	-	✓	-	-	MPI_File_write_at()
-	-	✓	-	✓	MPI_File_seek_shared() MPI_File_write_shared()
-	-	✓	✓	-	MPI_File_write_at_all()
-	-	✓	✓	✓	MPI_File_seek_shared() MPI_File_write_ordered()
-	✓	-	-	-	MPI_File_iwrite()
-	✓	-	-	✓	MPI_File_iwrite_shared()
-	✓	-	✓	-	MPI_File_write_all_begin() MPI_File_write_all_end()
-	✓	-	✓	✓	MPI_File_write_ordered_begin() MPI_File_write_ordered_end()
-	✓	✓	-	-	MPI_File_write()
-	✓	✓	-	✓	MPI_File_write_shared()
-	✓	✓	✓	-	MPI_File_write_all()
-	✓	✓	✓	✓	MPI_File_write_ordered()
✓	-	-	-	-	MPI_File_iread_at()
✓	-	-	-	✓	MPI_File_seek_shared() MPI_File_iread_shared()
✓	-	-	✓	-	MPI_File_read_at_all_begin() MPI_File_read_at_all_end()
✓	-	-	✓	✓	MPI_File_seek_shared() MPI_File_read_ordered_begin() MPI_File_read_ordered_end()
✓	-	✓	-	-	MPI_File_read_at()
✓	-	✓	-	✓	MPI_File_seek_shared() MPI_File_read_shared()
✓	-	✓	✓	-	MPI_File_read_at_all()
✓	-	✓	✓	✓	MPI_File_seek_shared() MPI_File_read_ordered()
✓	✓	-	-	-	MPI_File_iread()
✓	✓	-	-	✓	MPI_File_iread_shared()
✓	✓	-	✓	-	MPI_File_read_all_begin() MPI_File_read_all_end()
✓	✓	-	✓	✓	MPI_File_read_ordered_begin() MPI_File_read_ordered_end()
✓	✓	✓	-	-	MPI_File_read()
✓	✓	✓	-	✓	MPI_File_read_shared()
✓	✓	✓	✓	-	MPI_File_read_all()
✓	✓	✓	✓	✓	MPI_File_read_ordered()

Tabelle 4.1: Abbildung der Anforderungstypen auf MPI-IO-Funktionen

die zu Ungenauigkeiten bei der Leistungsmessung führen können. Es wurden in Abschnitt 3.3.9 auch Lösungsmöglichkeiten diskutiert. Tabelle 4.2 fässt diese Abbildungsprobleme zusammen und zeigt, welche Lösungsmöglichkeit der PRIOmark verwendet. Im Folgenden werden Abbildungsprobleme erläutert, die durch den eigentlichen Lasterzeugungsalgorithmus entstehen.

Abbildungsproblem	Lösungsmöglichkeit
fehlendes `WHILE`-Konstrukt	Näherung mittels `LOOP`-Schleife
Reihenfolge der I/O-Anforderungen einer I/O-Operation	immer gleiche Reihenfolge mit möglichst starker Abwechslung zwischen den Anforderungstypen
Anforderungsgrößen der I/O-Anforderungen einer I/O-Operation	alle Anforderungen sind gleich groß
Prozessparallelität	alle Prozesse der I/O-Operation haben identisches I/O-Verhalten (s. 3.3.9)
noncontiguous I/O	alle Prozesse verwenden gleichen Anteil der Dateieinheit (s. 3.3.9)
contiguous I/O	in der aktuellen Implementierung korrekt nur durch *processNum* = 1 abbildbar; Modell erlaubt aber genauere Abbildung

Tabelle 4.2: Einschränkungen der Lastbeschreibung und Realisierung beim PRIOmark

Rundungsfehler

Ein Abbildungsfehler, der nicht durch das Lastmodell bedingt ist, sondern durch den verwendeten Lasterzeugungsalgorithmus, entsteht durch die bei der Berechnung erforderlichen Rundungen. Die kleinste Einheit, mit der im oben beschriebenen Algorithmus gerechnet wird, ist die Größe einer I/O-Anforderung. Besteht eine I/O-Operation nur aus wenigen Anforderungen (bspw. 10) kann es leicht Probleme bei der Abbildung dieser 10 Anforderungen auf die 32 verschiedenen Anforderungstypen geben. In der Form funktioniert der Algorithmus also nur dann gut, wenn viele Anforderungen zu einer Operation zusammengefasst werden (also die Granularität der Beschreibung sehr grob ist). Das Problem lässt sich minimieren, wenn der Abbildungsalgorithmus während des gesamten Berechnungsvorganges nicht mit der Anforderungsgröße als kleinster Einheit rechnet, sondern mit Bytes. Erst im letzten Schritt wird die Anforderungsanzahl als Quotient der berechneten Bytes pro Anforderungstyp und *sizeMean* gebildet. Der dabei entstehende Divisionsrest wird als zusätzliche Anforderung mit der entsprechenden Größe ebenfalls ausgeführt, um Rundungsfehler zu minimieren und feingranulare Lastbeschreibungen zu realisieren. Der PRIOmark verwendet aus diesem Grund auch diese Art der Implementierung.

nicht-sequentielle Zugriffe

Nicht-sequentielle Zugriffe werden bei MPI-IO auf zwei verschiedene Arten realisiert. Einerseits gibt es Funktionen mit expliziten Offsets, denen die Position innerhalb der Datei direkt als Parameter übergeben wird. Explizite Zugriffe verwenden und manipulieren keinen Dateizeiger. Andererseits kann der Dateizeiger mittels einer der *Seek*-Funktionen verändert werden, um anschließend den Datenzugriff durchzuführen. Der

Profile-Analyzer behandelt beide Zugriffsmöglichkeiten für nicht-sequentielle Zugriffe identisch, weshalb der Lastgenerator keine Information über den originalen Zugriff mehr besitzt. Dieses Abbildungsproblem lässt sich nur komplett lösen, wenn die Dimensionalität des Lastmodells erhöht wird. Da beide Zugriffsfunktionen aber eine sehr ähnliche Semantik besitzen, wird davon ausgegangen, dass der Leistungseinfluss gering ist.

Die vom I/O-Benchmark in Phase III der Leistungsanalyse ermittelten Ergebnisse müssen dem Nutzer verständlich und übersichtlich aufbereitet werden. Dies geschieht in der vierten Phase des Benchmarkings, in der der Anwender des PRIOmarks vom Result-Analyzer unterstützt wird. Dessen Vorstellung ist Inhalt des folgenden Abschnitts.

4.3.5 Result-Analyzer

entfernte Ausführung

Der Result-Analyzer als letztes Werkzeug der PRIOmark-Toolchain dient der leichteren Analyse der Ergebnisdaten des PRIOmark. Der Result-Analyzer kann direkt mit dem PRIOmark-I/O-Benchmark über die Netzwerkschnittstelle kommunizieren. Diese Möglichkeit entstand aus der Anforderung, dass viele Hochleistungsrechner keine grafische Darstellung ermöglichen, da sie dem Nutzer nur einen Netzwerkzugang und keinen physikalischen Zugang zum Rechensystem zur Verfügung stellen. Mittels der PRIOmark-Architektur kann der Benchmark entfernt auf dem Hochleistungsrechner gestartet werden, während die Ergebnisse lokal mittels des grafischen Result-Analyzers gelesen und interpretiert werden. Auch eine Nutzung des Result-Analyzers bei lokaler Ausführung des PRIOmarks ist möglich, wenn zur Kommunikation beider Applikationen die lokale Netzwerkschnittstelle verwendet wird.

Darstellungsmöglichkeiten

Der Result-Analyzer besitzt verschiedene Darstellungsmöglichkeiten der durch den PRIOmark ermittelten Ergebnisse. Neben einer tabellarischen Darstellung ist eine Ergebnisdarstellung als zwei- oder dreidimensionaler Graph möglich. Durch dreidimensionale Graphen können vom Nutzer Schnittebenen gelegt werden, um den entsprechenden auf der Schnittebene liegenden zweidimensionalen Graphen genauer zu analysieren. Abbildung 4.6 zeigt einen Screenshot des PRIOmark-Result-Analyzers. Neben den genannten Möglichkeiten der Darstellung von Ergebnissen kann der Result-Analyzer auch zur entfernten Steuerung des PRIOmark-Benchmarks genutzt werden. Damit geht er über die Möglichkeiten eines Werkzeugs für die vierte Phase des I/O-Benchmarkings hinaus. Diese zusätzliche Möglichkeit bietet dem Nutzer den Vorteil der Ausführung des Benchmarks sowie der Analyse der Ergebnisse in einem Werkzeug und ermöglicht darüber hinaus eine Stapelverarbeitung mehrerer PRIOmark-Messungen.

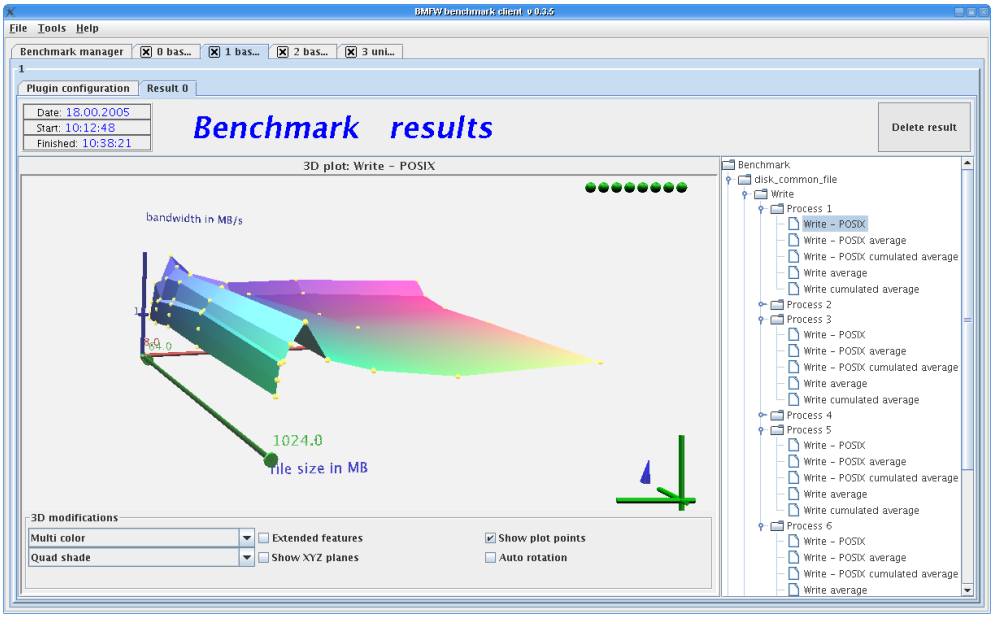

Abbildung 4.6: Screenshot des Result-Analyzers

4.3.6 Zusammenfassung Werkzeuge

Diese kurze Vorstellung der PRIOmark-Werkzeugkette gab einen Überblick über die Architektur und Möglichkeiten der einzelnen Werkzeuge. Es wurde gezeigt, dass der PRIOmark verschiedene Möglichkeiten der Definition von I/O-Lasten und deren messtechnische Auswertung zur Verfügung stellt. Allerdings erschwert dies auch die Vergleichbarkeit der Ergebnisse unterschiedlicher Nutzer, da diese prinzipiell unterschiedliche Lasten verwenden könnten und damit unvergleichbare Ergebnisse erhalten. Diesem bereits in Absatz 3.4.4 geschilderten Problem wird sich in den folgenden Abschnitten gewidmet.

4.4 Einheitliche Workload-Definitionen

Wie in Abschnitt 3.4.4 dargelegt wurde, kann es bei der Nutzung von applikationsbasierten Benchmarks leicht zu einem Vergleich von Ergebnissen kommen, die nicht vergleichbar sind, da sie bei Verwendung unterschiedlicher Last-Szenarien ermittelt wurden. Dieses Problem kann umgangen werden, in dem die Lasten, mit denen die unterschiedlichen Nutzer die I/O-Messungen durchführen, einheitlich festgelegt werden. Da Benchmarks mit definiertem Lastverhalten prinzipbedingt keine Variation des Lastverhaltens erlauben, gibt es das beschriebene Problem bei dieser größten Klasse von Benchmarks nicht. Bei applikationsbasierten Benchmarks müssen, trotz des Problems, Möglichkeiten geschaffen werden, eine Vergleichbarkeit zu erlauben.

Standard-Workloads

In den folgenden Absätzen werden Lasten definiert, die als Workload-Definition-Files mit dem PRIOmark mitgeliefert werden und die als Standard-Mess-Szenarien von den Anwendern eingesetzt werden sollten, wenn eine gemeinsame Applikation als Vergleichsgrundlage fehlt. Dennoch sollen auch diese *Standard-Workloads* Praxisrelevanz besitzen und ein breites Spektrum an möglichen Applikationen abdecken. Insbesondere sollten sie Applikationen abdecken, die im Fokus der Anwender des PRIOmarks liegen. Allerdings hat eine Definition von Applikationslasten als Grundlage der Vergleichbarkeit eines I/O-Benchmarks nicht nur für die Verwendung im PRIOmark eine Bedeutung. Sie kann auch von anderen I/O-Benchmarks, also z. B. von I/O-Benchmarks mit definierten Workloads als Grundlage ihrer Last verwendet werden, um so zukünftig eine Benchmark-übergreifende Vergleichbarkeit der Messergebnisse zu ermöglichen.

Praxisrelevanz

4.4.1 Einsatzszenarien des PRIOmark

Zur Ermittlung von I/O-Lasten, die nutzerrelevanten Applikationen entsprechen, ist eine Analyse der Einsatzszenarien des PRIOmarks notwendig. Der Benchmark wurde entwickelt, um Leistungsanalysen von I/O-Systemen, wie sie insbesondere in parallelen Hochleistungsrechensystemen eingesetzt werden, zu ermöglichen. Der Anwendungsfall des High-Performance-Computings ist also der wichtigste und muss in Hinblick auf die Nutzung im PRIOmark mit besonderer Sorgfalt betrachtet werden. Die wissenschaftlichen Probleme, die auf Hochleistungsrechensystemen berechnet werden, haben besondere I/O-Lasteigenschaften und sind nicht mit den Lasten klassischer Workstations zu vergleichen. Für den Vergleich verschiedener I/O-Systeme derartiger Rechner ist also mindestens ein Standardworkload zu definieren, der klassische Lasten wissenschaftlicher Applikationen abbildet.

parallele Hochleistungs-rechensysteme

Neben dem Einsatz des PRIOmarks in derartigen Szenarien ist auch eine Nutzung des Systems im herkömmlichen Workstation- und Servereinsatz relevant. Insbesondere bei Serversystemen soll eine Optimierung stattfinden, die die verfügbare Leistung des Systems maximal ausnutzt. Auch Workstations haben ein Optimierungspotential, so dass auch bei diesen Systemen eine Leistungsvermessung interessant sein kann. Insbesondere Privatanwender können auf diese Weise die Hard- und Softwarekomponenten ihrer Systeme auf korrekte Funktionsfähigkeit überprüfen. Zusammenfassend sind also mindestens drei *Standardworkloads* zu definieren, die die wesentlichen Anwendungsfälle des PRIOmark abdecken:

Serversysteme

1. I/O-Workload für parallele wissenschaftliche Applikationen
2. I/O-Workload für Serversysteme
3. I/O-Workload für Workstation-Systeme

Diese drei Anwendungsfälle werden im Folgenden genauer untersucht.

4.4.2 I/O-Workload für parallele wissenschaftliche Applikationen

Wie bereits in Abschnitt 3.3.4 vorgestellt wurde, besitzen parallele Applikationen typische I/O-Zugriffsmuster, die für herkömmliche Workstations oder Serversysteme ungewöhnlich sind. Zwei wesentliche Arten der Zugriffsmuster sind verschränkte und unverschränkte Zugriffe, die beide bei wissenschaftlichen Applikationen vorkommen [85]. Im Folgenden wird für jede Zugriffsart eine Last erarbeitet, um so alle wichtigen typischen I/O-Lasten wissenschaftlicher Applikationen als Standard-Lasten für den PRIOmark zur Verfügung zu stellen.

NWChem

Ziel der Berechnung

NWChem existiert aktuell in der Version 5.1 und ist ein Softwarepaket zur Berechnung von chemischen Problemen auf Hochleistungsrechensystemen und Cluster-Computern [100, 101]. Entwickelt wird NWChem von der *Molecular Sciences Software Group* des *Pacific Northwest National Laboratory* [102].

Der Code berechnet die Energie von Elektronen in und um Molekülen und kann so zur Ermittlung von Elektronendichten verwendet werden. Auf diese Weise ist u. a. eine Berechnung der bei chemischen Reaktionen freiwerdenden oder benötigten Energie möglich. Auch die Vorhersage des strukturellen Aufbaus entstehender Moleküle kann durchgeführt werden. In [87] wird das I/O-Verhalten einer NWChem-basierten Applikation, bestehend aus 64 Prozessen, analysiert. Das Zugriffsverhalten des NWChem-Codes ist vorwiegend lesend und sequentiell, sowie nicht verschränkt. Es lässt sich in drei Phasen unterteilen:

Zugriffsverhalten des NWChem

Konfigurationsdateien. Der Masterknoten liest die Konfigurationsdateien (nicht-sequentielles Lesen; Anforderungsgröße kleiner als 250 Byte; Datenvolumen ca. 11 MB; Zugriffe sind zufällig und werden aufgrund fehlender Angaben in [87] als blockierend, nicht-kollektiv und ohne gemeinsamen Dateizeiger definiert).

Initialisierungsdaten. Alle Knoten berechnen die Basis-Dateien zur späteren Arbeit und schreiben diese auf den Sekundärspeicher (sequentielles und unverschränktes Schreiben; Anforderungsgröße 64 kB; Datenvolumen ca. 29 MB pro Prozess). Da [87] die Lastcharakteristik bei Verwendung von POSIX-I/O beschreibt, wird der Anteil von kollektiven Zugriffen mit/ohne gemeinsamen Dateizeiger nicht spezifiziert. Es wird deshalb definiert, dass diese Zugriffe kollektiv seien, um ein größeres Optimierungspotential zu erlauben.

Programmberechnung. Alle Prozesse lesen sequentiell und unverschränkt die geschriebenen Daten, berechnen mit diesen die Ergebnisse. Der Masterknoten schreibt die Ergebnisse im Anschluss auf den Sekundärspeicher. Die Phase der Berechnung kann also wiederrum in zwei Teilphasen unterschieden werden, in denen jeweils gelesen und geschrieben wird und die 13 Mal wiederholt wird (die

Gesamtlaufzeit der Applikation beträgt 5439 Sekunden, während jede Berechnungsphase ca. 360 Sekunden andauert; abzüglich der Initialisierungsphase von ca. 600 Sekunden, ergibt dies ca. 13 Iterationen der Berechnungsphasen). Die einzelnen Parameter berechnen sich analog zu denen der Initialisierungsphase. Auch hier wird die Lesephase zur Erhöhung des Optimierungspotentials kollektiv durchgeführt. Bei der Schreibphase ist ein kollektiver Zugriff unnötig, da nur ein Prozess das Schreiben durchführt.

Die Gesamtmenge der transferierten Daten der Applikation in allen 4 Phasen beträgt somit ca. 28 GB. Abbildung 4.7 präsentiert die I/O-Last des NWChem in der eingeführten grafischen Repräsentation.

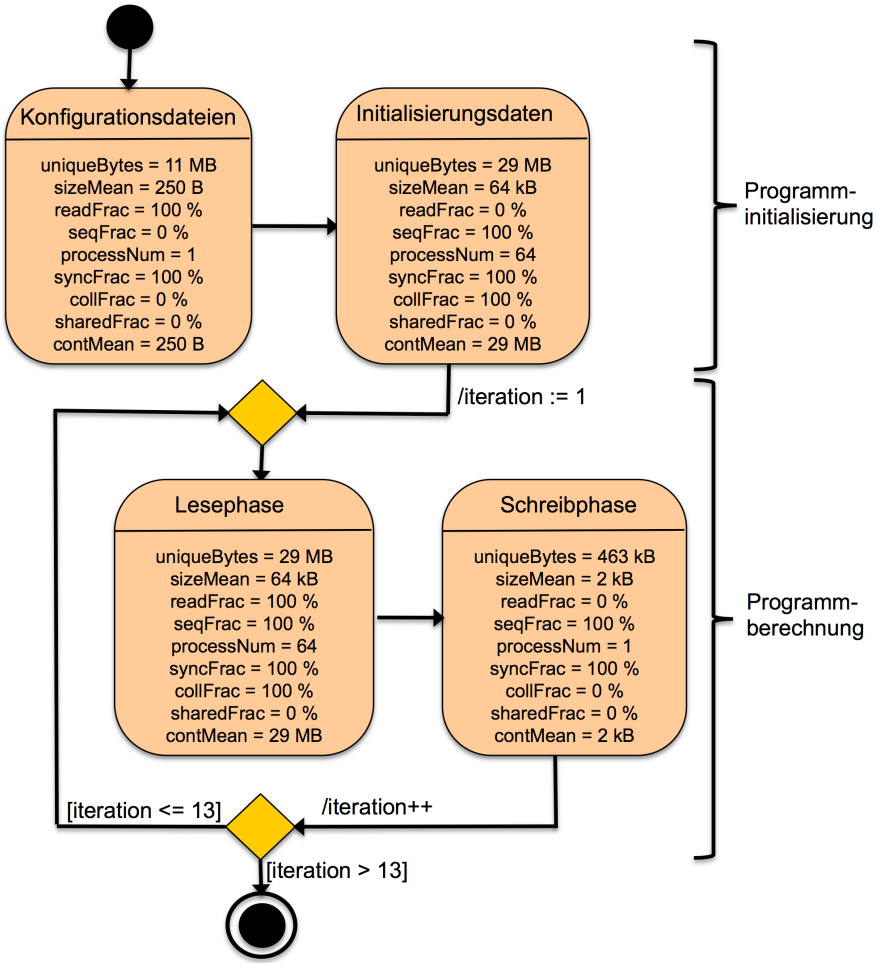

Abbildung 4.7: I/O-Workload des NWChem-Codes

QCRD

Ziel der Berechnung

QCRD (Quantum Chemical Reaction Dynamics) ist eine parallele Applikation zur Berechnung der quantenmechanischen Zustände von Atomen und Molekülen während einer chemischen Reaktion. Der Code berechnet die Lösungen der Schrödingergleichung („Grundgleichung der Quantenmechanik"), die die räumliche und zeitliche Entwicklung des Zustands einen Quantensystems beschreiben.

Zugriffsverhalten des QCRD

Dafür wird eine große globale Matrix erstellt, die nicht in den Hauptspeicher des Computersystems passt und deshalb im Sekundärspeicher abgelegt wird. Einzelne Prozesse arbeiten dann mit Portionen dieser Matrix, wobei ein klassisch verschränktes Zugriffsverhalten entsteht. Der originale QCRD-Code, wie er in [85] beschrieben wird, verwendet POSIX-I/O und führt deshalb vor jedem I/O-Zugriff eine Neupositionierung des Dateizeigers durch. Dieses extrem ineffiziente Zugriffsverhalten kann bei der Verwendung von MPI-IO erheblich verbessert werden, weshalb in dieser Arbeit eine Abbildung auf einen MPI-IO-Workload durchgeführt wird.

Das Zugriffsverhalten des QCRD ist zyklisch. Nach einer ersten Phase der Initialisierung, in der alle Prozesse parallel Initialisierungsinformationen lesen, folgt abwechselnd während der gesamten Applikationslaufzeit eine Lese- und eine Schreibphase. Zwischen beiden Phasen findet bei der Applikation eine Berechnung statt. In der hier erstellten Lastbeschreibung wird die Initialisierungsphase ignoriert, da sie pro Prozess nur aus zwei Lesezugriffen besteht, die anteilig sehr wenig Daten transportieren.

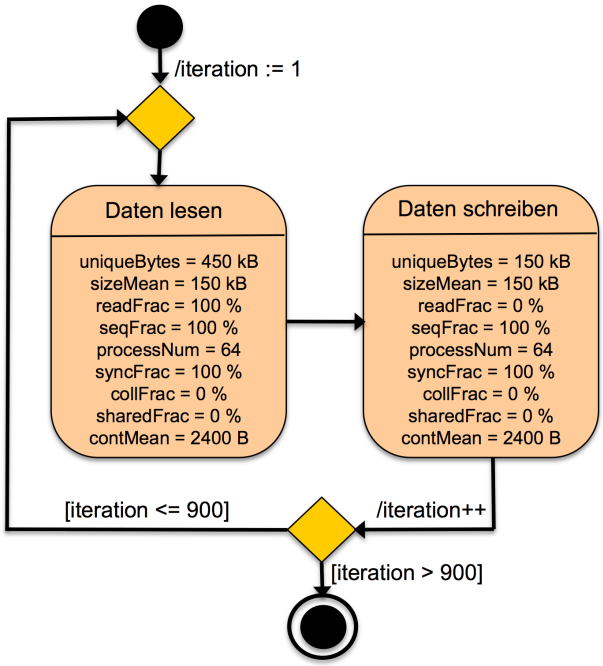

Abbildung 4.8: I/O-Workload des QCRD-Codes

Die Lese- und Schreibphasen, die insgesamt ca. 900 Mal wiederholt werden (jede Phase ist ca. 18 Sekunden lang bei einer Applikationslaufzeit von 16356 Sekunden), sind

mit ihren Eigenschaften in Abbildung 4.7 dargestellt. Verglichen mit dem NWChem-Code ist bemerkenswert, dass die Zugriffe des QCRD verschränkt sind. Erkennbar ist dies an dem verglichen mit *sizeMean* kleinen Wert des Parameters *contMean*. Bei diesem I/O-Lastverhalten greift also jeder der 64 Prozesse auf 2400 konsekutiv angeordnete Bytes einer 150 kByte großen Dateieinheit zu. Abbildung 4.8 zeigt den QCRD-I/O-Workload.

4.4.3 I/O-Workload für Serversysteme

Serversysteme dienen dem Anbieten eines Dienstes für sogenannte Klienten. Diese Dienste werden im lokalen Netzwerk (Local Area Network - LAN) oder im Internet angeboten.

World Wide Web

Eine Untersuchung aus dem Jahr 1997 zeigt, dass der größte Anteil der im Internet transferierten Daten WWW-Datenverkehr darstellte [103]. In aktuellen Untersuchungen ist der Anteil des WWW-Datenverkehrs am Gesamtvolumen von transferierten Daten gesunken, da der größte Anteil des Verkehrs mittlerweile durch Peer2Peer-Anwendungen verursacht wird, die keine klassischen durch Server angebotenen Dienste darstellen [104] und damit nicht im Fokus des PRIOmark liegen. Dennoch haben die WWW-Daten einen signifikanten Anteil am insgesamt verursachten Internet-Verkehr [105]. Der WWW-Dienst stellt noch immer einen der wichtigsten Dienste des Internets dar und der Webserver ist demnach ein wichtiges Einsatzszenario für Serversysteme im Internet. Aus diesen Gründen wird die durch einen Webserver verursachte I/O-Last im Folgenden als ein Standardworkload verwendet. In [106] wird die I/O-Last eines Web-Servers recht genau spezifiziert. Tabelle 4.3 gibt diese Spezifikation wieder. Um die aus dieser Beschreibung maximal erreichbare Genauigkeit zu verwenden, wird

Zugriffsverhalten eines Web-Servers

Anf.-Größe [Byte]	Anteil [%]	lesend [%]	zufällig [%]
512	22	100	100
1024	15	100	100
2048	8	100	100
4096	23	100	100
8192	15	100	100
16384	2	100	100
32768	6	100	100
65536	7	100	100
131072	1	100	100
524288	1	100	100

Tabelle 4.3: I/O-Last eines Web-Servers [106]

jede Zeile der Tabelle auf eine Operation im I/O-Workload-Raum abgebildet. Die Werte für Anforderungsgröße und für den lesenden und zufälligen Anteil der Zugriffe wer-

den dabei direkt auf die Parameter *sizeMean*, *readFrac* und *1-seqFrac* abgebildet. Der prozentuale Anteil der Zugriffe wird realisiert, indem *uniqueBytes* der einzelnen Operationen dem entsprechenden Anteil bezogen auf alle transferierten Daten entspricht. Die Gesamtmenge der transferierten Daten muss mindestens der Größe des Hauptspeichers des Systems entsprechen, um Caching-Effekte zu vermindern. Es werden 8 GB gewählt, da dieser Wert derzeit für die meisten Computersysteme die beschriebene Bedingung erfüllt. In der Praxis werden Web-Server eingesetzt, die Daten entweder blockierend oder nicht-blockierend vom Sekundärspeicher laden. Der Anteil blockierender Zugriffe eines Servers wird durch den Parameter `syncFrac` spezifiziert. Diese Form des Datenzugriffs kann leicht mit der Kommunikation des Webservers mit den Clients im Netzwerk verwechselt werden, die bei Verwendung von non-blocking Serversockets ebenfalls nicht-blockierend ist, hängt damit aber nicht zusammen, da `syncFrac` ausschließlich den Anteil blockierender Zugriffe beim Sekundärspeicherzugriff und nicht beim Netzwerkzugriff spezifiziert. Da Webserver blockierende und nicht-blockierende I/O-Zugriffe nicht vermischen, kann `syncFrac` sowohl den Wert 0 Prozent als auch den Wert 100 Prozent haben. Werte zwischen 0 und 100 Prozent werden in der Praxis nicht vorkommen. Es werden deshalb zwei Web-Server-Lasten als Standards festgelegt (*Web-Server mit nicht-blockierendem Disk-I/O* und *Web-Server mit blockierendem Disk-I/O*), wobei in den folgenden Messungen ausschließlich der Web-Server mit blockierenden Disk-I/O-Zugriffen als Beispiel-Workload eingesetzt wird. Der Parameter *processNum* wird im Standard-Workload auf 8 gesetzt, da die Standardkonfiguration des Apache-Webservers 5 bis 10 parallel arbeitende Prozesse definiert, die auf Anfragen seitens der Clients warten und diese dann beantworten [107]. Die beiden Parameter *collFrac* und *sharedFrac* haben bei der von Web-Servern verwendeten POSIX-I/O-Schnittstelle keine Bedeutung und werden deshalb auf 0 Prozent gesetzt, weil die Zugriffe weder eine kollektive Synchronisation noch gemeinsame Dateizeiger verwenden. Mittels des Parameters *contMean* kann die für Webserver typische unverschränkte Zugriffsart realisiert werden, indem der Wert dieses Parameters in jedem Zustand auf den Wert der Größe der Zugriffseinheit (*sizeMean*) gesetzt wird. Die gesamte Webserver-Last wird zweimal ausgeführt, um Caching-Effekte in die Leistungsermittlung einzubeziehen.

Das so entstehende Lastdiagramm besteht aus 10 Zuständen und wird in Abbildung 4.9 dargestellt.

4.4.4 I/O-Workload für Workstation-Systeme

Workstation-Systeme sind leistungsfähige Arbeitsplatzcomputer, an denen Endnutzer vielfältige Arbeiten durchführen. Entsprechend vielfältig sind auch die I/O-Lasten, die auf einer Workstation erzeugt werden. Die Beschreibung einer generischen I/O-Last für derartige Systeme ist also nur sehr abstrakt möglich. In [106] wird eine Workstation-Last beschrieben, die ursprünglich als Grundlage eines Benchmarks

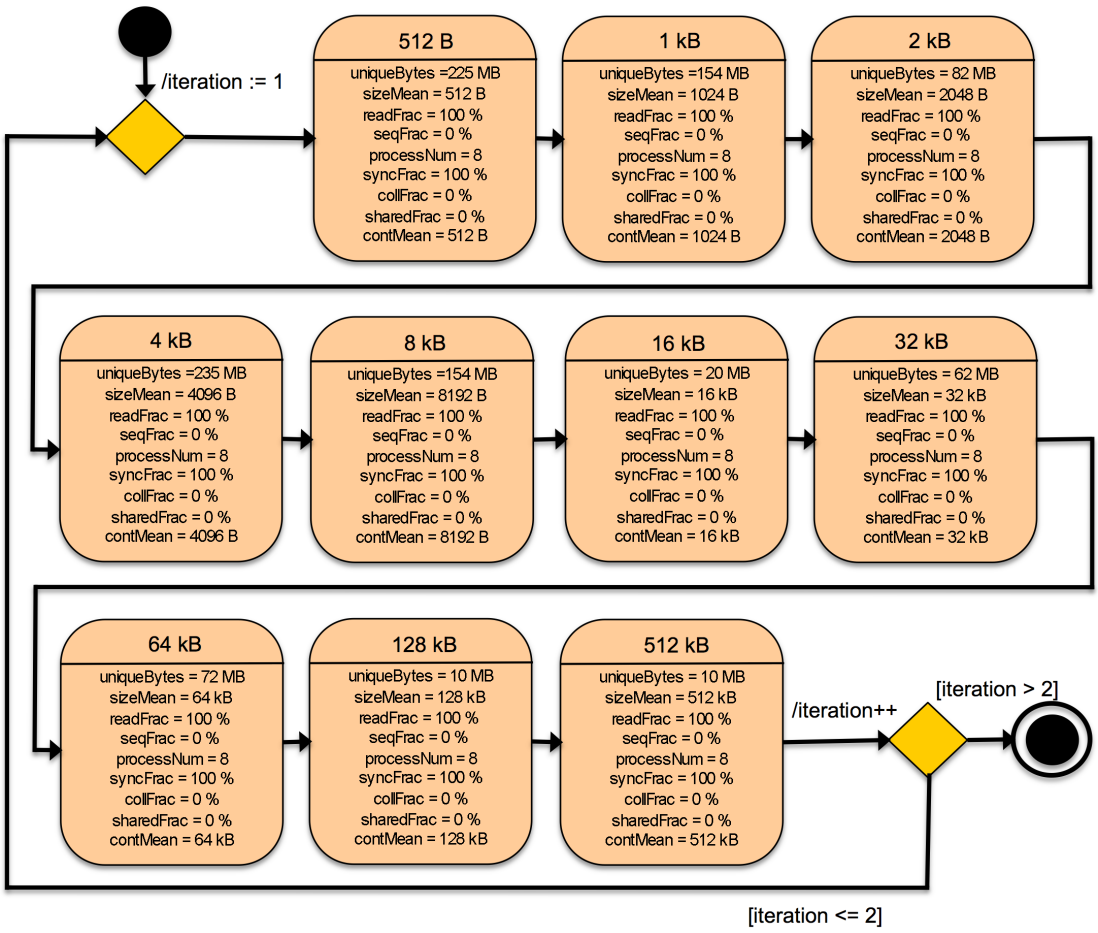

Abbildung 4.9: I/O-Workload des Webservers mit blockierendem Disk-I/O

Zugriffsverhalten einer Workstation

des US-amerikanischen Unternehmens *StorageReview.com* verwendet wurde. *StorageReview.com* beschäftigt sich mit der Leistungsvermessung von Speicherlösungen und unterhält eine umfangreiche Leistungsdatenbank [18]. Dieser Workstation-Workload verwendet 8 kB-Anforderungen, von denen 80 Prozent der Daten gelesen und 80 Prozent zufällig auf dem Sekundärspeicher verteilt abgelegt werden (also 20 Prozent sequentiell angeordnet sind). Diese Angaben entsprechen in Teilen dem Ergebnis der Untersuchung [108], welche den Aufbau von Workstation-Workloads beschreibt, die von 14 unterschiedlich qualifizierten Personen während ca. 45 Tagen gesammelt wurden. Diese Untersuchung kam unter anderem zu dem Ergebnis, dass 78 Prozent aller Zugriffe auf Workstations synchron stattfinden. Die Anzahl von parallelen Prozessen wird wie beim Webserver-Workload auf 8 gesetzt und orientiert sich damit an herkömmlichen POSIX-I/O-Benchmarks, die bis zu 10 parallele Prozesse erzeugen. Die Gesamtdatenmenge sollte ebenfalls größer als typische Hauptspeichergrößen sein und wird deshalb mit 8 GB verteilt auf die 8 Prozesse festgelegt. Da eine Workstation-Last typischerweise POSIX-I/O verwenden wird, werden die Parameter *collFrac*, *sharedFrac* und *contMean* Werte erhalten, die für POSIX-I/O-Zugriffe typisch sind. Abbil-

dung 4.10 gibt den Aufbau einer so entstehenden durchschnittlichen Workstation-I/O-Last zusammenfassend wieder.

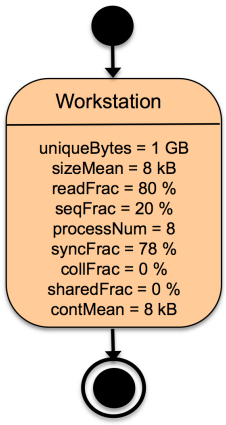

Abbildung 4.10: I/O-Workload einer Workstation

4.5 Zusammenfassung

In diesem Kapitel wurde die prototypische Implementierung einer I/O-Benchmark-Werkzeugkette vorgestellt, die die vorgestellte Architektur eines idealen I/O-Benchmark-Systems realisiert. Dazu wurde der Aufbau der einzelnen Werkzeuge des PRIOmarks, deren Funktionsweise und auch ihr Zusammenspiel erläutert. Um das Problem der schlechten Vergleichbarkeit von Messergebnissen bei applikationsbasierten Benchmarks zu vermeiden, wurden „Standard-I/O-Lasten" definiert, die der PRIOmark als Messgrundlage zur Verfügung stellt. Mit der Erstellung der Standard-Lasten konnte gezeigt werden, dass die Abbildung verschiedener Lasten mit dem eingeführten Lastmodell möglich ist.

Diese Lasten können aufgrund des klaren Aufbaus ihrer Struktur nicht nur im PRIOmark, sondern auch in anderen I/O-Benchmarks verwendet werden, um außerdem eine Vergleichbarkeit von Benchmarkergebnissen verschiedener Benchmarks zu ermöglichen.

Kapitel 5

PRIOmark im Messeinsatz

Die Praxis sollte das Ergebnis des Nachdenkens sein, nicht umgekehrt.
– HERMANN HESSE (1877 - 1962)

Inhalt dieses Kapitels ist die Vorstellung und Bewertung von Messergebnissen, die mit dem PRIOmark in der Praxis ermittelt wurden. Dazu werden zuerst die Ziele der Messungen vorgestellt, bevor die Messplattform und die Messungen präsentiert werden.

5.1 Ziele der Messungen

Ziel der in diesem Kapitel präsentierten Messungen ist es, herauszufinden, ob das in dieser Arbeit eingeführte Lastmodell eine gute Abbildung realer Lastszenarien realisiert und ob der im letzten Kapitel vorgestellte Algorithmus zur Lasterzeugung aus einer Lastbeschreibung realitätsgetreue Ergebnisse liefert. Zuerst werden dazu Messungen vorgestellt, die die im letzten Kapitel eingeführten Standardlasten gegenüberstellen, um die Unterschiedlichkeit der verschiedenen Lasten zu verdeutlichen. Im Anschluss wird in mehreren Messungen das I/O-Lastmodell untersucht. Dazu wird eine Standardlast als Grundlage verwendet, bei der jeweils einer der neun Parameter des I/O-Workload-Raumes variiert wird, während die restlichen acht konstant bleiben. Dieses Vorgehen ermöglicht den Lasteinfluss jedes einzelnen Parameters genau zu untersuchen und zu bewerten. Auf diese Weise werden u. U. Lastparameter gefunden, deren Lasteinfluss so gering ist, dass sie im Lastmodell ignoriert werden sollten. In der letzten präsentierten Messung wird die Genauigkeit des präsentierten I/O-Benchmarking-Ansatzes untersucht, indem die I/O-Lasten zweier herkömmlicher I/O-Benchmarks als Vergleichsgrundlage mit dem PRIOmark verwendet werden. Der PRIOmark wird die Last der beiden herkömmlichen Benchmarks mit unterschiedlichen Granularitäten nachbilden, damit mit den entstehenden Ergebnissen eine Aussage über die Genauigkeit der Nachbildung bei verschiedenen Beschreibungsgranularitäten erfolgt.

Vergleich der Lastszenarien

Lasteinfluss der Parameter des I/O-Workload-Raumes

Granularität und Genauigkeit

5.2 Messplattform

Als Messplattform für die präsentierten Messungen wird ein Clustersystem, bestehend aus einem Frontend und drei Rechenknoten, verwendet. Abbildung 5.1 stellt die wichtigsten Kenndaten der Hardware des für die Messungen verwendeten Clusters dar.

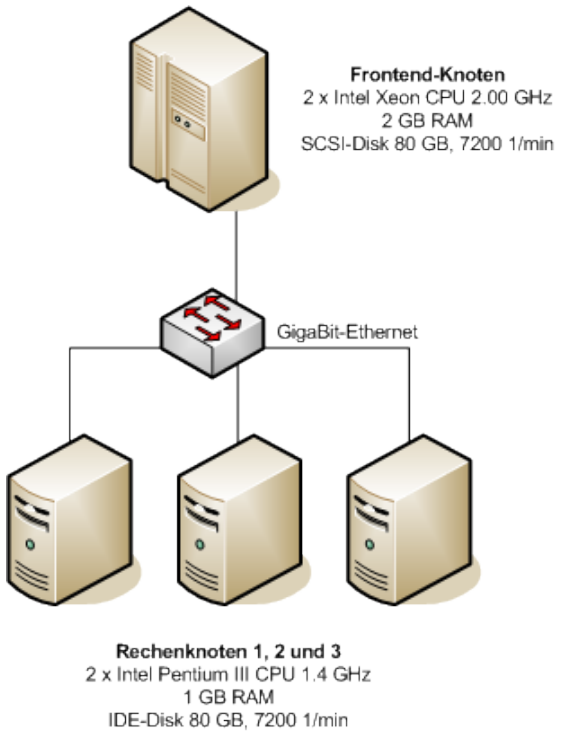

Abbildung 5.1: Hardware der Messplattform

Betriebssystem

Der Cluster-Computer verwendet *Rocks*-Linux als Betriebssystem [109]. *Rocks* ist ein von der amerikanischen *National Science Foundation* [110] gefördertes Projekt, das die Entwicklung einer speziell für Cluster-Computer entwickelten Linux-Distribution zum Ziel hat. Die Distribution enthält Pakete für wichtige Cluster-Werkzeuge zur Administration und Konfiguration von Clustern, sowie Werkzeuge zur Programmentwicklung und -ausführung, wie MPI oder Batch-Queueing-Systeme.

Message-Passing-Interface

Der PRIOmark wurde unter Verwendung einer aktuellen MPICH2-Implementierung übersetzt und ausgeführt. MPICH2 ist eine frei verfügbare Implementierung des Message-Passing-Interfaces in der Version 2 und wird an der *Mathematics and Computer Science Division* des *Argonne National Laboratory* unter Leitung von William Gropp entwickelt [111].

Als Dateisystem für die Messungen wurde das parallele Dateisystem PVFS (s. Abschnitt 2.1.3) eingesetzt, wobei der Frontend-Knoten den Metadatenserver und drei der vier Clusterknoten jeweils einen I/O-Server zur Verfügung stellen. Zur Verteilung der Applikationen wird auf dem Cluster NFS in der Version 3 verwendet.

5.3 Durchführung der Messungen

Auf dem vorgestellten Cluster wurden drei verschiedene Messreihen durchgeführt. Die erste Messreihe vergleicht Ergebnisse miteinander, die mit den eingeführten Standardlasten erzeugt wurden. Anschließend wird eine Messreihe präsentiert, die die I/O-Leistung in Abhängigkeit der eingeführten I/O-Lastparameter des Workload-Raumes darstellt. Ziel ist es, mit diesen Messungen zu zeigen, welche Parameter tatsächlich einen Leistungseinfluss besitzen und welche u. U. entfernt werden sollten.

Ein zentrales Element der vorgestellten Lastbeschreibung ist die Möglichkeit, die Lastbeschreibung in unterschiedlichen Feinheitsgraden (Granularitäten) darzustellen. Es kann vermutet werden, dass der Fehler einer Messung bei niedriger Granularität der Lastbeschreibung wächst, da nur bei einer hohen Granularität die Lastinformation vollständig ist. Die letzte Messreihe zeigt, wie stark die Granularität einer Lastbeschreibung den Messfehler beeinflusst.

5.3.1 Messreihe 1: Vergleich der Lastszenarien

Zielstellung

Um grundsätzlich zu zeigen, dass Lasten einen hohen Einfluss auf die gemessene Leistung haben, werden in der ersten Messreihe die vier in Abschnitt 4.4 vorgestellten Lasten auf dem Cluster ausgeführt. Um eine bessere Vergleichbarkeit der Lasten zu ermöglichen, werden im Gegensatz zu den Last-Definitionen in Abschnitt 4.4 alle Lasten mit nur einem Prozess gestartet. Damit ist der Realitätsbezug insbesondere der parallelen Lasten nicht mehr gegeben. Allerdings wird der Leistungseinfluss des Netzwerkes bei den Messungen verkleinert.

Durchführung

Ergebnisse

Abbildung 5.2 zeigt, dass die Leistungsunterschiede zwischen den Lasten erheblich sind. Die beim NWChem-Workload erreichte Leistung ist fast um den Faktor 5 höher als der vom Webserver erreichte I/O-Durchsatz.

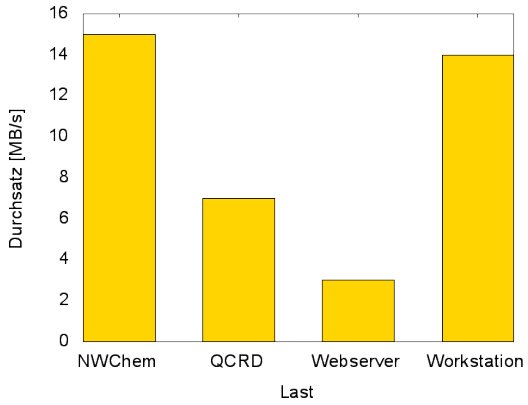

Abbildung 5.2: Vergleich des I/O-Durchsatzes bei Ausführung der I/O-Lasten NWChem, QCRD, Webserver und Workstation

Es ist aufgrund der hohen Komplexität der Lasten schwer bestimmbar, warum das Leistungsverhalten in dieser Form ausgeprägt ist. Sowohl NWChem-Last, als auch Workstation-Last sind leseintensiv. Dies könnte deren hohe I/O-Leistung erklären. Allerdings ist auch die QCRD-Last leseintensiv und erreicht nur ca. die halbe I/O-Leistung wie NWChem und Workstation. Die Sequentialität kann an dieser Stelle kaum ausschlaggebend sein, da sowohl QCRD als auch NWChem fast vollständig sequentielle Lasten sind. Der Webserver hingegen besitzt eine Last, die einen vollständig nicht-sequentiellen Charakter besitzt, was die schlechte Leistung des ansonsten rein lesenden Workloads erklären könnte. Es zeigt sich bereits an diesen wenigen spekulativen Betrachtungen, die längst nicht alle Parameter einbeziehen, dass eine genaue Beurteilung der komplexen I/O-Lasten der Applikationen kaum möglich ist. Der Einfluss einzelner Parameter an der gesamten Leistung kann im Allgemeinen nicht mehr ermittelt werden. Der nächste Abschnitt dieses Kapitels wird sich eingehend mit dem Einfluss einzelner Parameter auf die Gesamtleistung befassen, in dem in jeder Messreihe jeweils nur einer der neun Parameter variiert wird.

Lastabhängigkeit der Leistung

Dennoch unterstreicht diese erste Messung die deutliche Abhängigkeit der I/O-Leistung von der eingesetzten Last, wie es bereits in zahlreichen anderen wissenschaftlichen Publikationen belegt wurde. Sie zeigt auch, dass die große Anzahl an Benchmarks, die nur eine fest definierte Last unterstützen, in der Praxis unbrauchbar sind, wenn Leistungen spezifischer Applikationen untersucht werden müssen. In diesen Fällen ist eine möglichst detailgetreue, aber dennoch einfach zu erstellende Lastbeschreibung, wie die in dieser Arbeit entwickelte, notwendig.

5.3.2 Messreihe 2: Lasteinfluss der Parameter des I/O-Workload-Raumes

Im I/O-Lastmodell dieser Arbeit wurden neun I/O-Parameter für wichtige Merkmale der I/O-Last definiert, die verschiedene Einflüsse auf die I/O-Leistung besitzen. Messreihe 1 hat bereits gezeigt, dass durch Lastmessungen anhand der vordefinierten Lastszenarien kein Einfluss einzelner Parameter herauszufinden ist. Ziel dieser Messungen ist es, den Lasteinfluss aller Parameter der in dieser Arbeit eingeführten Lastbeschreibung zu ermitteln. So kann festgestellt werden, ob einzelne Parameter in der Lastbeschreibung u. U. überflüssig sind, da deren Lasteinfluss marginal ist. Um die Stärke des Lasteinflusses herauszufinden, wird in allen folgenden Messungen die gleiche Last als Messgrundlage verwendet. Es wird zur Untersuchung jedes Lastparameters jeweils nur dieser eine Parameter (der Faktor) variiert, während alle anderen konstant bleiben. Als Grundlage der Messungen dient die in Abbildung 4.10 vorgestellte Last einer Workstation, da diese im Gegensatz zu allen anderen vorgestellten Lastszenarien nur aus einer I/O-Operation besteht. Die Variation einzelner Parameter ist so einfacher möglich. Die Workstation-Last dient als Festpunkt (*focal point*), von welchem jeweils ein Parameter variiert wird. Es entsteht auf diese Weise für jeden untersuchten Parameter ein sogenannter Single-Parameter-Graph, der jeweils auf der Abszisse den Wertebereich des

Ziel

Durchführung

entsprechenden Parameters darstellt und auf der Ordinate die erreichte Leistung bei Verwendung des Workstation-Workloads mit der Variation dieses einen Lastparameters.

Eine Besonderheit gegenüber des in Abbildung 4.10 dargestellten Workloads ist, dass der Parameter *contMean* praktisch nicht untersucht werden konnte, da der PRIOmark die Variation dieses Parameters in der aktuellen Version nicht unterstützt. Untersuchungen bzgl. des Leistungseinflusses von *noncontiguous I/O* werden deshalb im Anschluss separat durchgeführt. Alle Messungen wurden auf dem Dateisystem PVFS ausgeführt.

Ergebnisse

Abbildung 5.3 stellt die Ergebnisse dieser Messungen dar. Die Abzisse zeigt die variierten Werte des entsprechenden Lastparameters, während auf der Ordinate die ermittelten Leistungswerte in MB/s angegeben werden. Die angegebene Leistung ist die von einem Prozess erreichte Leistung. Da der Workstation-Workload mit parallelen 8 Prozessen arbeitet, kann die Gesamtleistung der Applikation jeweils durch Multiplikation mit 8 ermittelt werden. Die Skalierungen sowohl der X-, als auch der Y-Achse sind nicht bei allen Diagrammen gleich. Dies erschwert zwar den Vergleich zwischen den Diagrammen, ermöglicht aber auch schwächere Tendenzen besser zu erkennen.

uniqueBytes

Der Parameter *uniqueBytes* hat erwartungsgemäß einen Leistungseinfluss. Mit steigendem *uniqueBytes* sinkt die I/O-Leistung, wobei der Abfall im kleineren Wertebereich des Parameters am stärksten ist. Ist die Menge an insgesamt transferierten Mengen klein, werden Caching-Effekte besser ausgenutzt, da ein größerer Anteil der benutzten Daten in den Caches gespeichert wird. Dieser Effekt ist stärker, wenn die I/O-Last dem Lokalitätsprinzip folgt, wenn also aufeinander folgende Zugriffe zeitlich und örtlich nah beieinander liegen (zeitliche und örtliche Lokalität). Da bei dem Workstation-Workload 80 Prozent aller Zugriffe eine zufällige Sekundärspeicheradresse verwenden, kann die örtliche Lokalität nur bei einem kleinen Teil der Zugriffe garantiert werden, weshalb der Leistungseinfluss des Parameters *uniqueBytes* bei dieser Last vergleichsweise gering ist.

seqFrac

Entsprechend hat die Erhöhung des Anteils der sequentiellen Aufrufe (Parameter *seqFrac*) eine Leistungssteigerung aufgrund der steigenden örtlichen Lokalität der Zugriffe zur Folge. Diese Leistungssteigerung erfolgt allerdings auch in Folge der weniger häufigen Umpositionierungen des Schreib-/Lesekopfes der verwendeten Festplatten. Derartige Umpositionierungen sind mechanischer Natur und kosten viel Zeit, so dass sie zu Lasten der I/O-Leistung gehen. In den Messungen zeigt sich bei dem verwendeten *focal point*, dass eine rein sequentielle I/O-Last eine mehr als doppelte I/O-Leistung erreicht, wie völlig zufällige Zugriffe.

readFrac

Die Leistungskurve des Parameters *readFrac* zeigt ein ähnliches Verhalten, wie die des Parameters *seqFrac*. Bei Erhöhung des Anteils von lesenden Zugriffen, steigt die I/O-Leistung nicht-linear an. Parallele Lesezugriffe sind offensichtlich wesentlicher performanter als parallele Schreibzugriffe. Dies liegt im Wesentlichen an den bei Schreibzugriffen notwendigen Synchronisierungsmaßnahmen. Sobald von zwei Zugriffen auf ei-

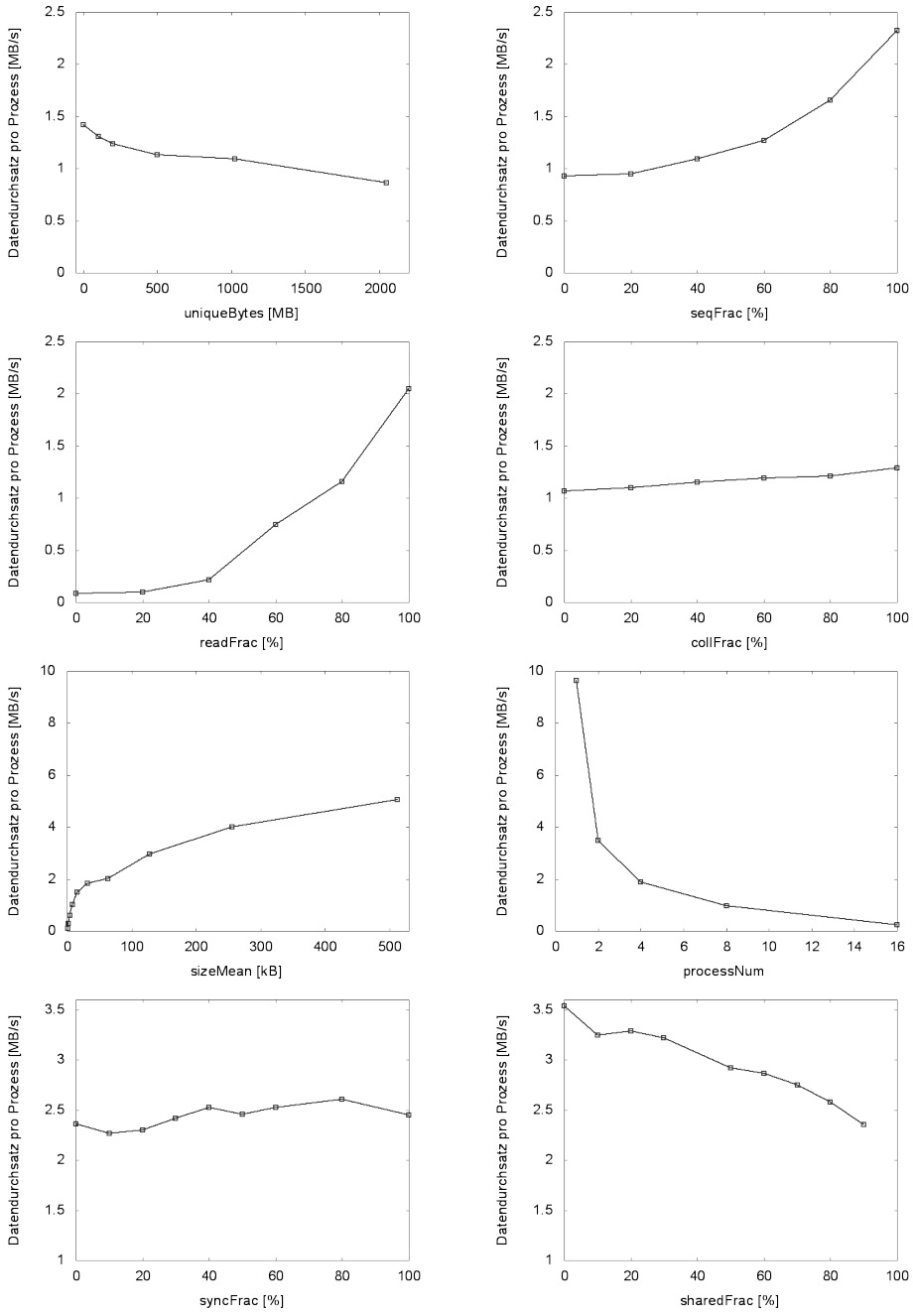

Abbildung 5.3: Variation einzelner Lastparameter bei Verwendung der Workstation-Last zur Ermittlung der Parameterabhängigkeit der I/O-Leistung

ne Speicherposition mindestens einer der beiden Zugriffe schreibender Natur ist, muss das System eine korrekte Reihenfolge der beiden Zugriffe gewährleisten, da sonst beim lesenden Zugriff falsche Daten (entweder zu alte oder zu neue) gelesen werden, bzw. bei zwei Schreibzugriffen nach beiden Zugriffen das falsche der beiden Daten im Speicher abgelegt wurde. Bei zwei Lesezugriffen ist eine derartige Synchronisierung, die

zusätzliche Wartezeiten und damit eine Degradierung der Leistung verursacht, nicht notwendig. Lesezugriffe sind in ihrer Reihenfolge beliebig austauschbar.

Ähnlich verhält es sich auch mit den verschiedenen Caches des I/O-Subsystems. Bei Schreibzugriffen muss die Konsistenz der Inhalte aller Kopien des veränderten Datums in den verschiedenen Cache-Speichern gewährleistet werden. Dieser Mehraufwand vermindert die Gesamtleistung.

sizeMean

Der Parameter *sizeMean* definiert die durchschnittliche Größe einer I/O-Anforderung. Führt eine Applikation viele kleine I/O-Anforderungen durch, werden die entsprechenden Anfragen sehr oft durch die zahlreichen Schichten des komplexen I/O-Subsystems (vgl. Abschnitt 2.1.1) geleitet. Der dabei entstehende Mehraufwand gegenüber wenigen großen Anforderungen verringert den Durchsatz, da viel Zeit für den administrativen Aufwand verbraucht wird. Eine höhere Leistung ist also erreichbar, wenn die I/O-Anforderungsgröße erhöht wird. Abbildung 5.3 macht dies deutlich.

processNum

Abbildung 5.3 zeigt weiterhin den deutlichen Einfluss der Anzahl paralleler Prozesse auf die I/O-Leistung. Erwartungsgemäß wird die I/O-Leistung pro Prozess schlechter, wenn mehr Prozesse parallele Last erzeugen, da sich die Prozesse die im System vorhandenen Ressourcen teilen müssen. In der Messung fällt der I/O-Durchsatz von fast 10 MB/s mit einem Prozess auf deutlich unter 1 MB/s bei 16 Prozessen. Dieser starke Einbruch der Leistung bei der Verwendung von PVFS wurde nicht erwartet. PVFS als paralleles Dateisystem besitzt in der verwendeten Konfiguration drei I/O-Knoten, die einen parallelen Zugriff mehrerer Prozesse erlauben und damit eine gute Skalierung mit der Prozessanzahl ermöglichen sollten. So lange die Anzahl der I/O-aktiven Prozesse nicht wesentlich größer ist als die Anzahl der I/O-Knoten, dürfte ein so starker Leistungsabfall wie in den erfolgten Messungen nicht auftreten. In [94] konnte dies mit einer anderen I/O-Last gezeigt werden. Der Grund für diesen starken Leistungsabfall liegt vermutlich an der geringen Größe der I/O-Anforderungen bei der Workstation-Last. PVFS teilt eine Datei standardmäßig in 64 kB große Blöcke, die dann auf alle Knoten des PVFS-Systems verteilt werden. Die Parallelität von PVFS kann also nur dann effektiv genutzt werden, wenn unterschiedliche Prozesse mindestens in einem Abstand von 64 kB auf die Datei zugreifen. Nur dann steht jedem Prozess exklusiv die komplette Bandbreite (sowohl des Netzwerks als auch der Festplatte) zum Datenzugriff zur Verfügung. Bei zu kleinen Anforderungen, die lokal nahe bei einander liegen, greifen alle Prozesse bei einem I/O-Zugriff auf denselben durch PVFS verwalteten Block zu, so dass sich eine Situation ähnlich zu der bei NFS entwickelt: Die Bandbreite der Netzanbindung zu dem angefragten Server bzw. auf dem Server zur Festplatte steht keinem Prozess mehr exklusiv zur Verfügung und muss geteilt werden. Es ergibt sich das in Abbildung 5.3 gezeigte Verhalten, das auch typisch für einen NFS-Server ist.

Für die ersten fünf gezeigten Parameter konnte in [80] bei Verwendung eines lokalen Sekundärspeichers ein ähnliches Verhalten, wie das hier nachgewiesene, gezeigt werden. Der Leistungseinfluss des Parameters *processNum* ist bei Ausführung des Bench-

marks auf einem Cluster-Computer wesentlich deutlicher, da die Prozesse verteilt auf den einzelnen Knoten des Rechners ablaufen. Damit ist ein höherer Kommunikationsaufwand verbunden, der die Leistung bei höherer Prozessanzahl stärker sinken lässt.

Im Folgenden wird die Abhängigkeit der I/O-Leistung von den in dieser Arbeit eingeführten neuen Lastparametern *collFrac*, *syncFrac*, *sharedFrac* und *contMean*, sowie der Iterationsanzahl einer Operation untersucht. Sowohl bei den Messungen des Parameters *syncFrac*, als auch des Parameters *sharedFrac* musste die Workstation-Last leicht abgeändert werden. Bei der Verwendung von acht Prozessen führten beide Lasten zu regelmäßigen Abstürzen der MPI-Umgebung aufgrund eines internen MPI-Fehlers. Bei Verwendung von zwei Prozessen lief das System wesentlich stabiler, wobei auch dann nicht für alle Werte der Parameter Ergebnisse ermittelt werden konnten. Die Ergebnisse sind dennoch ausreichend, um eine Bewertung der Parameter zu ermöglichen.

collFrac

Der Parameter *collFrac* gibt den Anteil der kollektiv ausgeführten I/O-Aufrufe an. Kollektiv ausgeführte I/O-Zugriffe werden durch das MPI-System länger gepuffert, so dass ein Schreiben der Daten aller kollektiven Aufrufe erst geschehen muss, wenn der letzte kollektive Aufruf beendet wurde. Dies eröffnet ein höheres Optimierungspotential beim Schreiben der Daten, da größere zusammenhängende Blöcke gebildet werden, die weniger I/O-Anforderungen an den Treiber zur Folge haben. Ein höherer Anteil kollektiver Aufrufe sollte also auch eine höhere I/O-Leistung ermöglichen, was Abbildung 5.3 bestätigt. Die Messungen zeigen einen Leistungsgewinn bei Verwendung von kollektiven Zugriffen, der aber vergleichsweise gering ist. Der geringe Einfluss kann an der Last selbst liegen, da auch bei kollektiven Zugriffen nur wenig Optimierungspotential besteht, wenn die Zugriffe stark auf dem Sekundärspeicher verteilt werden, wie es bei der Workstation-Last der Fall ist. Andererseits muss das Optimierungspotential, das kollektive Aufrufe ermöglichen, auch durch die MPI-Implementierung ausgenutzt werden. An dieser Stelle kann es also deutliche Unterschiede zwischen verschiedenen MPI-Implementierungen und -versionen geben.

syncFrac

Asynchrone (nicht-blockierende) Aufrufe ermöglichen insbesondere bei rechenintensiven Applikationen Leistungsgewinne, da die Applikationslaufzeit insgesamt verringert werden kann, wenn bereits berechnete Daten geschrieben werden, während die Applikation parallel neue Ergebnisse berechnet. Allerdings wird sich dieses Verhalten auf den reinen I/O-Durchsatz nicht zwangsläufig positiv auswirken, da asynchrones I/O verglichen mit einem identischen blockierenden I/O-Zugriff kein zusätzliches Optimierungspotential in Hinblick auf den eigentlichen I/O-Zugriff bietet. Entsprechend zeigt Abbildung 5.3 auch nur eine geringe Abhängigkeit der I/O-Leistung vom Anteil der asynchronen Aufrufe bei der verwendeten Last. Es ist eine leichte Leistungssteigerung zu erkennen, solange *syncFrac* bis 80 Prozent steigt, danach fällt die Leistung ab.

Deutlicher ist dieses Verhalten bei Verwendung kleinerer Anforderungsgrößen erkennbar, wie Abbildung 5.4 zeigt. Eine Steigerung des I/O-Durchsatzes wird erreicht, solan-

ge der Anteil der blockierenden Zugriffe bis auf ca. 60-80 Prozent steigt, anschließend sinkt der I/O-Durchsatz wieder.

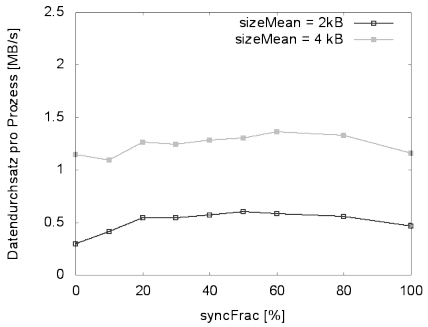

Abbildung 5.4: Abhängigkeit der I/O-Leistung vom Parameter *syncFrac* bei unterschiedlichen Anforderungsgrößen

Dieses Verhalten entspricht nicht dem, das beim Einsatz asynchroner I/O-Aufrufe erwartet werden würde. Mehr asynchrone Aufrufe sollen eine Erhöhung der Applikationsleistung erzielen. Tatsächlich ist eine Vermessung der gesamten Applikationsleistung mit Hilfe dieses Benchmarks auch nicht möglich. Der PRIOmark misst den reinen I/O-Durchsatz der Applikation, der sich bei einem hohen Anteil von asynchronen Aufrufen offensichtlich verringert. Dies liegt daran, dass bei Nutzung asynchroner Aufrufe parallel von der Applikation andere Aufgaben (z. B. Berechnungen) durchgeführt werden. Diese parallele Ausführung mehrerer Aufgaben führt ab einem gewissen Grad zu Engpässen bei der Nutzung gemeinsamer Ressourcen (wie z. B. Hauptspeicher), so dass eine Verringerung des I/O-Durchsatzes stattfindet. Ein kleiner Anteil von asynchronen Zugriffen (< 20 Prozent) führt noch nicht zu Ressourcenkonflikten, was erklärt, dass das Leistungsmaximum bei etwa 80 Prozent synchronen Zugriffen zu finden ist.

Die gesamte Applikationslaufzeit wird sich bei Erhöhung des Anteils von asynchronen Zugriffen trotz Verringerung der reinen I/O-Leistung aber dennoch vermindern (und dadurch eine Erhöhung der Applikationsleistung zur Folge haben), da mehrere Aufgaben parallel (wenn auch jeweils mit etwas verringerter Leistung) durchgeführt werden. Der eigentlich positive Effekt der asynchronen I/O-Aufrufe kann mit einem reinen I/O-Benchmark demnach nicht nachgewiesen werden. Dazu ist die Kombination mit einem Rechenbenchmark notwendig, wie es bei einigen Applikationsbenchmarks gemacht wird. Tatsächlich ist in diesem Zusammenhang die Frage zu stellen, ob die Anforderung *I/O-Bezug* an einen idealen I/O-Benchmark, wie sie in Abschnitt 2.3.2 beschrieben wurde, wirklich notwendig (oder sogar hinderlich) ist. Nur ein I/O-Benchmark, der das Kriterium *I/O-Bezug* nicht erfüllt, kann den Leistungseinfluss des Parameters *syncFrac* realitätsnah dokumentieren.

Dennoch kann aus der erfolgten Messung eine interessante Schlussfolgerung gewonnen werden. I/O-lastige Prozesse, die kaum Berechnungen durchführen (wie dieser

I/O-Benchmark oder bspw. Checkpoint-Speicherungen bei HPC-Applikationen) erzielen durch die Verwendung von asynchronem I/O keinerlei Leistungsgewinne, da bei ihnen die Applikationsleistung weitestgehend von der I/O-Leistung bestimmt wird, die sich aber bei Erhöhung des asynchronen Anteils von I/O-Anforderungen verringert. Hier kann der höhere Implementierungsaufwand, der durch asynchrones I/O entsteht, eingespart werden.

sharedFrac

Der Parameter *sharedFrac*, der den Anteil der Aufrufe mit einem gemeinsamen Dateizeiger vieler Prozesse wiederspiegelt, zeigt in den durchgeführten Messungen einen deutlichen Leistungseinfluss. Je höher der Anteil von Zugriffen mit gemeinsamem Dateizeiger, desto schlechter ist die I/O-Leistung. Der I/O-Durchsatz fällt in den Messungen von 3,5 MB/s auf unter 2,5 MB/s ab. Dieses Verhalten wurde in dieser Form erwartet. Die Verwendung eines gemeinsamen Dateizeigers erfordert eine stärkere Synchronisation der einzelnen Prozesse durch das MPI-System, da alle Zugriffe mit gemeinsamem Dateizeiger voneinander abhängig sind. Es ist somit öfter notwendig, dass Prozesse auf die Fertigstellung anderer I/O-Anforderungen warten, was zu einer Leistungsdegradierung führt.

contMean

Der Parameter *contMean* beschreibt die Menge zusammenhängender Daten eines Prozesses in der geschriebenen Datenmenge. Mittels *contMean* ist also definierbar, ob Daten *contiguous* oder *noncontiguous* auf dem Sekundärspeicher abgelegt oder vom Sekundärspeicher gelesen werden. Der PRIOmark unterstützt diesen Parameter bis zum aktuellen Zeitpunkt nicht. Tatsächlich wird die Leistungseinfluss von *noncontiguous I/O* oft diskutiert. Es gibt aber nur wenige Benchmarks, die ihn gezielt untersuchen. Der HPIO (ehemals NCIO) ist ein derartiger Benchmark. In [68] dient er deshalb als Grundlage der Untersuchung des Leistungseinflusses von *noncontiguous I/O*. Der in dem Artikel als *Region Size* bezeichnete Parameter entspricht *contMean*. Der Artikel zeigt eine deutliche Steigerung des Datendurchsatzes bei Erhöhung des Parameters *contMean* und damit einen deutlichen Leistungseinfluss. Die Autoren des Artikels [112] kommen zu dem gleichen Ergebnis. Es wird deshalb an dieser Stelle der Leistungseinfluss des Parameters *contMean* nicht weiter untersucht.

iteration

Neben den vorgestellten neun Lastparametern des I/O-Workload-Raumes, ermöglicht das vorgestellte Lastmodell auch die Definition von Iterationsschleifen. Teile einer Lastbeschreibung können mehrfach wiederholt werden. Wie in Abschnitt 4.4.2 anhand der Beispiele der NWChem- und QCRD-Lasten gezeigt wurde, ist dieser Parameter zur Darstellung und Dokumentation des I/O-Verhaltens von Applikationen unerlässlich. An dieser Stelle wird geklärt, wie hoch der Einfluss der Anzahl der Wiederholungen einer Last auf die messbare I/O-Leistung ist. Dazu wurde die I/O-Operation, die die Workstation-Last beschreibt, in mehreren Messungen unterschiedlich oft wiederholt. Zwischen den Messungen wurden andere Applikationen auf der Messplattform ausgeführt, um die Füllung aller Caches mit anderen Instruktionen und Daten als denen des Benchmarks zu sichern. Abbildung 5.5 zeigt die Abhängigkeit des I/O-Durchsatzes von der Anzahl der Wiederholungen der Workstation-Last. Es ist eine Abhängigkeit der I/O-Leistung von der Iterationsanzahl erkennbar, die aber relativ gering

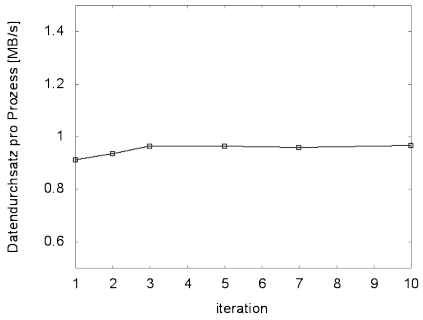

Abbildung 5.5: Abhängigkeit der I/O-Leistung von der Anzahl der Operationswiederholungen

ist (im Beispiel ca. 10 Prozent). Bei niedrigen Iterationsanzahlen findet ein Leistungsanstieg statt. Ab ca. 3 Iterationen bleibt die I/O-Leistung konstant und ist unabhängig von der Iterationsanzahl. Erklärbar ist dieser Zusammenhang mit dem Verhalten der Caches der I/O-Subsysteme der I/O-Knoten. Caches brauchen typischerweise eine Iteration, bis sie mit den Daten befüllt sind, die während des Durchlaufs bearbeitet werden. Dass die maximale Leistung nicht bereits bei der zweiten Iteration erreicht wird, liegt vermutlich an der besonders komplexen Struktur des verwendeten I/O-Systems, das Caches an zahlreichen Stellen einsetzt. Neben den Hauptspeichercaches, die jeder Prozessor jedes I/O-Knotens besitzt, gibt es lokale Festplattencaches und Puffer des MPI-Systems, die die Anzahl der langsamen Datentransfers verringern. Da als Dateisystem PVFS eingesetzt wird, das seinerseits das *2nd-extended file system* als lokales Dateisystem zur Speicherung der Daten einsetzt (s. 2.1.3), werden auch an dieser Stelle mehrere Caches der einzelnen Dateisysteme verwendet.

Die Iterationsanzahl einer Operation ist also ebenfalls ein I/O-beeinflussender Parameter. Es kann aber festgestellt werden, dass ein Einfluss nur bei kleinen Iterationszahlen besteht. Ein I/O-Benchmark sollte also zugunsten einer kürzeren Benchmarklaufzeit hohe Iterationszahlen verringern. Die Messung zeigt, dass auch bei komplexen I/O-Systemen eine Reduktion hoher Iterationsanzahlen auf 3 Iterationen erfolgen kann, ohne die Messergebnisse zu verfälschen. Dennoch sollte diese Reduktion erst im Benchmark geschehen und nicht im Workload-Analyzer, da die Iterationszahl durchaus wichtig ist, um zyklische Lasten als solche zu kennzeichnen.

Bewertung

Die vorgestellten Messungen der zweiten Messreihe konnten zeigen, dass alle Lastparameter, die im Rahmen dieser Arbeit zur Beschreibung einer I/O-Last eingeführt wurden, einen Lasteinfluss besitzen. Erwartungsgemäß ist der Lasteinfluss der Parameter unterschiedlich stark. Besonders großen Einfluss bei den durchgeführten Messungen besitzen die Parameter *sizeMean*, *readFrac*, *seqFrac*, *processNum* und *sharedFrac*. Die Variation des Parameters *syncFrac* zeigte einen Leistungseinfluss, der aber nicht den wirklichen Leistungseinfluss der asynchronen I/O-Zugriffe wiederspiegelt. An dieser Stelle muss darüber nachgedacht werden, parallel zu den I/O-Anforderungen des Benchmarks auch Berechnungen anzustellen, so dass die Parallelität dieser

beiden Aktionen, die erst durch asynchrone Zugriffe ermöglicht wird, eine Steigerung der gesamten Applikationsleistung ermöglicht. Der Einfluss der Iterationsanzahl von Schleifen in der Lastbeschreibung auf die I/O-Leistung wurde ebenfalls untersucht. Es konnte im Rahmen der Messungen festgestellt werden, dass nur die ersten 3 Iterationsdurchläufe eine Änderung der I/O-Leistung bewirken. Bei mehr als 3 Durchläufen bleibt die I/O-Leistung, die der Benchmark ermittelt, gleich. Dies entspricht den Erwartungen, da die Caches des Systems (Hauptspeicher-, Sekundärspeicher-, Festplattencache) in den ersten Durchläufen befüllt werden und dadurch Leistungsverbesserungen erzielen. Ab 3 Durchläufen sind alle Caches mit den Daten vollständig befüllt, so dass weitere Verbesserungen nicht möglich sind.

5.3.3 Messreihe 3: Granularität und Genauigkeit

Ziel

Die eingeführte Lastbeschreibung erlaubt es, Lasten mit unterschiedlichen Granularitäten zu beschreiben. Feingranulare Beschreibungen enthalten viel Informationen und sind entsprechend komplexer, während grobgranulare Beschreibungen eine starke Abstraktion der Last ermöglichen.

Durchführung

Ziel der dritten Messreihe ist die Untersuchung des Einflusses der Granularität einer Lastbeschreibung auf die Genauigkeit der Messergebnisse. Dazu werden die Lasten von zwei herkömmlichen Benchmark-Applikationen als Grundlage verwendet. Diese wurden mit dem I/O-Profiler aufgezeichnet und anschließend mit Hilfe des Analyzers als Workload-Definitionen mit unterschiedlichen Granularitäten abgespeichert. Der dabei verwendete Indexwert ist ein Maß für die Granularität und entspricht der Anzahl von aufeinander folgenden I/O-Anforderungen, die einer I/O-Operation (also einem I/O-Arbeitspunkt im I/O-Workload-Raum) zugeordnet werden. Je höher der Indexwert, desto niedriger ist also die Granularität – desto weniger I/O-Arbeitspunkte beschreiben die Last im I/O-Workload-Raum. Als Grundlagen dienten einerseits der *b_eff_io* von Rolf Rabenseifner [65] und andererseits der *strided*-Benchmark, der als einer der ersten I/O-Benchmarks im IPACS-Projekt entwickelt wurde [58]. Bei beiden Benchmarks handelt es sich um reine MPI-IO-Benchmarks.

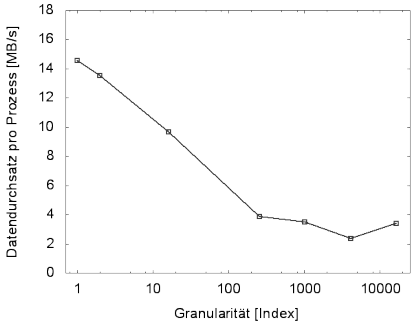

Abbildung 5.6: Abhängigkeit der I/O-Leistung von der Beschreibungsgranularität beim *b_eff_io*

Ergebnisse: b_eff_io als Grundlage

Abbildung 5.6 zeigt die Abhängigkeit des erreichten I/O-Durchsatzes bei einer variablen Beschreibungsgranularität beim *b_eff_io*. Das Ergebnis der Messung mit dem Indexwert 1 (also der höchsten Genauigkeit, bei der eine I/O-Operation einer I/O-Anforderung in der originalen I/O-Last entspricht) entspricht mit 14.57 MB/s etwa dem vom Benchmark selbst ermittelten Durchsatz von 15.31 MB/s. Messungen mit steigendem Indexwert haben steigende Abweichungen vom originalen Messwert. Der *b_eff_io* verwendet nacheinander viele verschiedene Zugriffsmuster, so dass das Zusammenfassen der I/O-Operationen schon bei kleinen Indexwerten einen signifikanten Informationsverlust verursacht (die unterschiedlichen Zugriffsmuster werden umso mehr miteinander zu einem einzigen Zugriffsmuster vermischt, je mehr die Operationen zusammengefasst werden). Tatsächlich zeigt diese Messung, dass die Zusammenfassung von mehreren I/O-Anforderungen zu einer I/O-Operation beim Beispiel des *b_eff_io* zu signifikanten Abweichungen von der wirklichen I/O-Leistung führt. Eine realitätsnahe Messung ist nur mit höchster Granularität möglich.

Ergebnisse: strided als Grundlage

Ein etwas anderes Bild ergibt sich bei der Untersuchung der Last des *strided*, deren Ergebnisse Abbildung 5.7 zeigt. Diese Applikation verfügt im Vergleich zum *b_eff_io* über eine I/O-Last, die nur zwei unterschiedliche Zugriffsmuster bei zwei verschiedenen Zugriffsgrößen verwendet. Solange die vier entstehenden Applikationsphasen nicht durch die Zusammenfassung von I/O-Operationen durchmischt werden, sind die Messergebnisse relativ einheitlich und liegen bei etwa 7.5 MB/s. Stärkere Informationsverluste treten erst bei den größten Indexwerten (den niedrigsten Granularitäten) auf, wenn die vier Applikationsphasen zusammengefasst werden. Entsprechend steigen die Abweichungen der Messergebnisse erst bei den höchsten Indexwerten.

Notwendigkeit von Phasen

Diese Messungen zeigen deutlich, dass der Ansatz, die I/O-Last einer Applikation als nur eine I/O-Operation abzubilden, wie es in [61] vorgeschlagen wird, nur für spezielle Applikationslasten eingesetzt werden kann. Dieses Verfahren funktioniert nur bei Applikationen, die während des gesamten Applikationslaufes ein einheitliches Zugriffsmuster verwenden. Abschnitt 4.4.2 dieser Arbeit zeigte aber bereits, dass ein derartiges Verhalten längst nicht bei allen Applikationen vorausgesetzt werden kann. Bei Applikationen, die mehrere Phasen mit unterschiedlichen I/O-Zugriffsmustern besitzen,

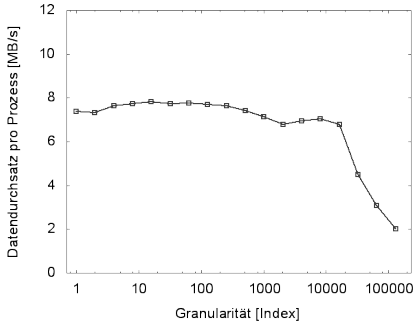

Abbildung 5.7: Abhängigkeit der I/O-Leistung von der Beschreibungsgranularität beim *strided*

können die Verluste, die durch Zusammenfassung der I/O-Anforderungen entstehen, so groß sein, dass die ermittelten Ergebnisse das Original nicht mehr abbilden.

Dies zeigt, dass ein Lastmodell zur Abbildung von I/O-Lasten auf jeden Fall Applikationsphasen unterstützen muss. Es wird außerdem deutlich, dass ein Workload-Analyzer Algorithmen verwenden muss, die Phasen unterschiedlicher Applikationslasten ermitteln, um diese in einzelne I/O-Operationen abzubilden.

5.4 Zusammenfassung

Dieses Kapitel der vorliegenden Arbeit zeigte anhand von 3 Messreihen die Nutzungsmöglichkeiten und Einschränkungen des PRIOmark-Benchmarks.

Messreihe 1 zeigte eklatante I/O-Leistungsunterschiede, bei Verwendung der definierten Standard-I/O-Lasten. Es wurde auch gezeigt, dass die Interpretation dieser Ergebnisse schwierig ist. Aus derartigen Messungen Aussagen über die Leistungsfähigkeit eines I/O-Systems unter anderen Lastszenarios als den definierten abzuleiten, ist fast unmöglich und beinhaltet ein großes Fehlerpotential. Die aktuell übliche Methode, bei der ein Nutzer einen I/O-Benchmark seiner Wahl mit einem fest definierten I/O-Zugriffsverhalten verwendet, um die I/O-Leistung seines Systems für ganz andere Applikationen zu bestimmen, kann keine gesichert richtigen Ergebnisse ermöglichen. Die Unterstützung nutzerdefinierbarer Lastszenarien ist also notwendig, um Ergebnisse zu erhalten, die für den Nutzer relevant sind.

Messreihe 2 zeigte die Abhängigkeit der I/O-Leistung von den Lastparametern des eingeführten I/O-Lastmodells. Es konnte eine Abhängigkeit bei allen Parametern nachgewiesen werden. Das Lastmodell beinhaltet also nur Parameter, die für eine korrekte Lastbeschreibung notwendig sind.

Die letzte Messreihe schließlich untersuchte die Genauigkeit der Messungen bei Nutzung unterschiedlicher Beschreibungsgranularitäten der gleichen I/O-Last. Es wurde gezeigt, dass eine hohe Genauigkeit der Messergebnisse nur bei feinster Granularität der Beschreibung gesichert werden kann. Es hängt von der Art der Last ab, ob auch eine niedrige Granularität noch genaue Messergebnisse erzielt. Homogene Lasten (d. h. Lasten, die über den gesamten Applikationszeitraum gleichartig sind), ermöglichen auch bei grober Granularität der Lastbeschreibung genaue Benchmark-Ergebnisse, während dies bei inhomogenen Lasten nicht der Fall ist.

Letztendlich konnten die durchgeführten Messungen die Funktionsfähigkeit des in dieser Arbeit vorgestellten I/O-Benchmarking-Ansatzes zeigen. Insbesondere wurde die Leistungsfähigkeit des dem Benchmark zugrunde liegenden neun-dimensionalen Lastbeschreibungsmodells nachgewiesen.

Kapitel 6

Zusammenfassung und Ausblick

Das schönste Glück des denkenden Menschen ist, das Erforschliche erforscht zu haben und das Unerforschliche zu verehren.

– JOHANN WOLFGANG VON GOETHE
(1749 - 1832)

Memory Wall des Sekundärspeichers

Die Leistungen von Prozessoren und Zugriffsgeschwindigkeiten aktueller Sekundärspeicher erhöhen sich zwar stetig, aber in gleichem Maße, wie deren Performance-Kenngrößen steigen, wird auch der Abstand zwischen der Leistung der schnellen Prozessoren und der vergleichsweise langsamen Sekundärspeichersysteme größer. Damit die Prozessoren nicht zu viel ihrer Rechenzeit damit verbringen, auf Daten des Sekundärspeichers zu warten, hilft es, die Datenmenge der transferierten Daten zu verringern. Aber auch hier geht die aktuelle Entwicklung in eine andere Richtung. Da die Sekundärspeicher immer größer werden, werden auch Applikationen „datenhungriger". Es steigt die Menge an Daten, die zwischen Speicher und Prozessor transferiert werden. Applikationen warten also einen immer größeren Teil ihrer Laufzeit nur auf Daten von I/O-Subsystemen. Ihre Laufzeit wird damit immer stärker durch die Leistung der Sekundärspeicher beeinflusst.

6.1 Ergebnisse

Diese Arbeit widmete sich dem genannten Problem, der *Memory Wall des Sekundärspeichers*, in dem neue Methoden zur Leistungsermittlung und -bewertung von Sekundärspeichersystemen entwickelt wurden. Diese Methoden wurden im Rahmen der Arbeit und im Umfeld des Verbundprojektes IPACS validiert, in dessen Kontext die Arbeit entstand. Die vorgestellten Methoden gehen über den Stand der Technik hinaus, da sie den Nutzer befähigen, beliebige I/O-Lasten von MPI-IO-basierten Applikationen nachzubilden und anhand dieser Lasten eine nutzerrelevante Leistungsanalyse durchzuführen.

In Abschnitt 2.4.3 wurden drei Kernfragen formuliert, die im Rahmen dieser Arbeit beantwortet werden konnten:

Benchmark-Architektur

1. Die zwei größten Probleme aktueller I/O-Benchmarks sind die fehlende Repräsentativität und die mangelnde Nutzerunterstützung. Diese Arbeit präsentiert eine neue Benchmark-Architektur, die diese Probleme löst und so für den Nutzer relevante Ergebnisse ermittelt und ihn außerdem beim Prozess des Benchmarkings unterstützt. Repräsentativität wird in dem vorgestellten Benchmark-Ansatz durch die Nachbildung von Lasten erreicht, die der Anwender selbst definieren kann. Anhand dieser Lasten wird eine für den Anwender relevante Leistungsanalyse durchgeführt. Diese Arbeit zeigt anhand einer neuen Klassifikation von Benchmark-Messmethoden, dass diese Art der I/O-Benchmark-Messung eine größtmögliche Repräsentativität ermöglicht. Um die Nutzerunterstützung zu gewährleisten, werden für alle Phasen II bis IV (vgl. Abschnitt 3.2) des Benchmarking-Prozesses Werkzeuge angeboten, die den Anwender bei der Erstellung relevanter I/O-Lasten, beim Benchmarking selbst und auch bei der Auswertung der Messergebnisse unterstützen. In Phase I muss der Nutzer durch unterstützende Maßnahmen (bspw. durch den in Abschnitt 3.2.1 vorgestellten Entscheidungsbaum) geführt werden. Nur so werden die 10 Teilschritte der Leistungsanalyse (vgl. Abschnitt 3.1) durchgeführt und dem Anwender ein fehlerfreies Benchmarking ermöglicht. Der PRIOmark als eine Werkzeug-Sammlung für das I/O-Benchmarking besteht aus einem Workload-Profiler, Workload-Analyzer und einem Workload-Definition-Tool für Phase II, einem Benchmark für Phase III und dem PRIOmark Result-Analyzer für Phase IV des Benchmarkings.

Lastbeschreibung

2. Ein I/O-Benchmark, der I/O-Lasten von Anwendungen nachbildet, muss diese Lasten in einer Form präsentiert bekommen, die er verarbeiten kann. Eine derartige Lastbeschreibung muss komplexe parallele I/O-Lasten genau abbilden und dennoch möglichst einfach gestaltet sein, um eine leichte Verständlichkeit und Analyse der Last zu ermöglichen. Vorhandene I/O-Lastbeschreibungen verbieten die Abbildung von Lasten der hochkomplexen MPI-IO-Schnittstelle und vernachlässigen besondere Applikationsspezifika wie inhomogene Lasten. In dieser Arbeit wurden existierende Abhandlungen in diesem Bereich dahingehend deutlich erweitert, dass nun die Definition komplexer paralleler I/O-Lasten mit homogener bzw. inhomogener Struktur in beliebigen Granularitäten in dem vorgestellten neuen Lastmodell ermöglicht wird. Die entwickelte Lastbeschreibung besteht aus einer Sequenz von I/O-Operationen, von denen jede mit den neun Lastparametern *uniqueBytes*, *sizeMean*, *readFrac*, *seqFrac*, *processNum*, *sharedFrac*, *syncFrac*, *collFrac* und *contMean* beschrieben wird (vgl. Abschnitt 4.4.2). Jede I/O-Operation beschreibt eine Reihe von I/O-Anforderungen. Die Menge der I/O-Anforderungen, die einer I/O-Operation entspricht, kann durch Nutzer oder Werkzeug gewählt werden, so dass eine hohe Flexibilität bei der Feinheit der Lasterstellung gewährleistet ist. Lasten stellen entweder eine sehr genaue Abbildung reeller Lasten dar oder eine abstraktere Darstellung, die

leichter überschaubar und verständlich ist, jedoch auch ungenauere Messwerte liefert. Die optimale Granularität der Beschreibung ist von der I/O-Last abhängig. Homogene Lasten, die während der gesamten Laufzeit ähnliche Zugriffsmuster verwenden, erzielen auch bei grober Granularität genaue Messergebnisse, während inhomogene Lasten feingranular in Form mehrerer Applikationsphasen beschrieben werden müssen. Teile der I/O-Lasten (also bestimmte I/O-Operationen) werden in der Lastbeschreibung iteriert, wobei Messungen zeigen, dass die Verwendung von mehr als drei Iterationen beim Benchmarking keine signifikanten Leistungsunterschiede mehr bewirkt.

3. *Erstellung von Lastbeschreibungen*
   Für die Erstellung solcher Lastbeschreibungen gibt es zwei Möglichkeiten: Dem Nutzer muss die Möglichkeit der selbstständigen Erstellung von Lasten gegeben werden. Dazu sind grafische Werkzeuge sinnvoll, um die Komplexität der Beschreibungen für den Nutzer einfacher überschaubar zu machen. Andererseits muss eine automatisierte Lastenerstellung durch Analyse des Lastverhaltens reell ablaufender Applikationen möglich sein. Nur diese Form der Lasterzeugung produziert Lasten mit größtmöglicher Applikationsnähe bei geringstmöglichem Aufwand für den Nutzer. Um eine geeignete Granularität der durch Applikationsanalyse entstehenden Lastbeschreibungen zu realisieren, muss das Analyse-Werkzeug Applikationsphasen, die ein unterschiedliches Lastverhalten besitzen, voneinander abgrenzen.

Neben der Klärung dieser Fragen wurde in der Arbeit exemplarisch eine Benchmark-Architektur, wie die beschriebene, implementiert und es wurde mit Messungen die Gültigkeit der gemachten Aussagen validiert. In den Messungen wurde insbesondere die Abhängigkeit der I/O-Leistung von allen Parametern des Lastmodells nachgewiesen, sowie die hohe Genauigkeit der I/O-Leistungsmessung unter Verwendung zweier Beispielapplikationen mit unterschiedlichem Lastverhalten bestätigt.

6.2 Offene Fragen

Dennoch sind im Rahmen dieser Arbeit Fragen entstanden, die eine Basis für weitere wissenschaftliche Untersuchungen bilden können.

- *contMean*
  Das in dieser Arbeit beschriebene Modell zur Lastbeschreibung enthält den Parameter *contMean*, der in der prototypischen PRIOmark-Implementierung nicht umgesetzt ist. Es gibt zahlreiche Publikationen, die die Abhängigkeit der Leistung von *contMean* bestätigen, eine Implementierung im PRIOmark sollte zukünftig auch Thema der weiteren Entwicklung sein.

- *Granularität*
  Die Messungen im Rahmen dieser Arbeit zeigen deutlich, dass eine geringe Granularität der Lastbeschreibung unter Umständen einen großen Messfehler zur Folge hat. Wenn es gelingt, den Messfehler in Abhängigkeit von der Granularität der Zielbeschreibung vorab zu schätzen, kann der Workload-Analyzer sehr

effektiv Zielbeschreibungen erstellen, die trotz Einfachheit eine hohe Genauigkeit erreichen.

I/O-Bezug

- In Abschnitt 2.3.2 wird der I/O-Bezug eines Benchmarks als eine zentrale Eigenschaft von I/O-Benchmarks dargestellt, da diese Eigenschaft die Komplexität des Benchmarks erheblich vereinfacht und so eine einfachere Analyse von Messergebnissen ermöglicht. Allerdings wurde in den Messungen in dieser Arbeit gezeigt, dass der Parameter *syncFrac* sinnvoll nur ausgewertet werden kann, wenn der Benchmark auch andere Lasten, als reine I/O-Lasten in die Betrachtung einbezieht. In Zukunft ist zu untersuchen, wie eine Messung des Leistungseinflusses des Parameters *syncFrac* geschehen kann, ohne die Komplexität der Lastbeschreibung durch zusätzliche Lastkomponenten zu erhöhen.

UMA/NUMA

- In Abschnitt 3.3.2 wurde die Hauptspeicheradresse als Teil einer I/O-Anforderung definiert, im Folgenden aber ignoriert, da im Allgemeinen die Geschwindigkeit des Zugriffs auf den Hauptspeicher von der Hauptspeicheradresse unabhängig ist. Tatsächlich ist dies so nur bei UMA-Systemen der Fall. NUMA-Systeme besitzen unterschiedliche Zugriffszeiten auf verschiedene Teile des Hauptspeichers, da einige Speicherteile zu anderen Prozessoren gehören und deshalb nicht mit bester Leistung von jedem Prozessor zugreifbar sind. Der Einfluss von NUMA auf die Sekundärspeicherleistung wird gering eingeschätzt, ist aber für die Zukunft zu klären.

Trotz dieser noch offenen Fragen konnte die vorliegende Arbeit systematisch Fragen und Probleme zum aktuellen I/O-Benchmarking aufdecken und beantworten. Die vorgeschlagenen Lösungen sind ein neuer Ansatz, das I/O-Benchmarking insbesondere bei Sekundärspeicherlösungen für HPC-Anwendungen zu verbessern und dem Nutzer Werkzeuge an die Hand zu geben, die I/O-Leistung der vorhandenen Systeme genau zu analysieren und zu optimieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] JAIN, Raj: *The Art of Computer Systems Performance Analysis*. New York : John Wiley and Sons, Inc., 1991

[2] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006

[3] KANT, K.: *Introduction to Computer Systems Performance Evaluation*. New York : McGraw-Hill, Inc., 1992

[4] *TOP500: Supercomputer Sites*. `http://www.top500.org`. Version: November 2007

[5] PETITET, Antoine ; WHALEY, Clint ; DONGARRA, Jack ; CLEARY, Andy: *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. `http://www.netlib.org/benchmark/hpl/`. Version: 2008

[6] GEAR6: White Paper: The Server-Storage Performance Gap - How disk drive throughput and access time affect performance / Gear6, Mountain View, USA. Version: 2006. `http://www.gear6.com/files/Gear6_WP_SSPG.pdf`. 2006. – Forschungsbericht

[7] MCKEE, Sally A.: Reflections on the memory wall. In: *CF '04: Proceedings of the 1st conference on Computing frontiers*. New York, NY, USA : ACM, 2004, S. 162

[8] WULF, Wm. A. ; MCKEE, Sally A.: Hitting the memory wall: implications of the obvious. In: *SIGARCH Comput. Archit. News* 23 (1995), Nr. 1, S. 20–24

[9] SHAN, Hongzhang ; STROHMAIER, Erich: Performance characteristics of the Cray X1 and their implications for application performance tuning. In: *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. New York, NY, USA : ACM Press, 2004, S. 175–183

[10] VASSILIADIS, Stamatis (Hrsg.) ; GAUDIOT, Jean-Luc (Hrsg.) ; PIURI, Vincenzo (Hrsg.): *Proceedings of the First Conference on Computing Frontiers, 2004, Ischia, Italy, April 14-16, 2004*. ACM, 2004

[11] JUURLINK, Ben H. H. ; LANGEN, Pepijn J.: Dynamic techniques to reduce memory traffic in embedded systems. In: [10], S. 192–201

[12] KÄMPE, Martin ; STENSTRÖM, Per ; DUBOIS, Michel: Self-correcting LRU replacement policies. In: [10], S. 181–191

[13] MOORE, Samuel K.: Multicore Is Bad News For Supercomputers. In: *IEEE Spectrum* (2008), Nr. 11, S. 11

[14] BEHNKE, Ralf: *Konzeption und Implementierung eines Profilers für MPI-IO in verteilten HPC-Anwendungen*, Universität Rostock, Institut für Informatik, Diplomarbeit, 2006

[15] *c't: Magazin für Computertechnik* (1990 - 2006). – 90/1 S. 68, 90/11 S. 322, 92/5 S. 187, 94/10 S. 182, 95/6 S. 120, 98/3 S. 128, 00/4 S. 240, 02/1 S. 150, 04/3 S. 146, 06/4 S. 150

[16] Harddisk mit 27 MByte/s. In: *c't: Magazin für Computertechnik* (1991), Nr. 1, S. 14

[17] STEUDTEN, Thomas: *50 Jahre Festplatte: Vom lahmen Riesen zum flotten Winzling*. `http://www.tecchannel.de/storage/komponenten/447433/50_jahre_festplatte_vom_lahmen_riesen_zum_flotten_winzling`. Version: 2006

[18] STORAGEREVIEW.COM: *StorageReview.com Homepage*. `http://www.storagereview.com`. Version: 2008

[19] HSU, W. W. ; SMITH, A. J.: The performance impact of I/O optimizations and disk improvements. In: *IBM Journal of Research and Development* 48 (2004), 3, Nr. 2, S. 255–289

[20] SOLLBACH, Wolfgang: *Storage Area Networks / Network Attached Storage*. München : Addison-Wesley Verlag, 2002

[21] ROBBE, Björn: *SAN - Storage Area Network*. Carl Hanser Verlag, 2004

[22] TANENBAUM, Andrew S.: *Modern Operating Systems*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2001

[23] RUBINI, Alessandro: *Linux Device Drivers (Nutshell Handbook)*. O'Reilly, 1998

[24] BOVET, Daniel P. ; CASSETTI, Marco ; ORAM, Andy (Hrsg.): *Understanding the Linux Kernel*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2000

[25] CARD, Rémy ; TS'O, Theodore ; TWEEDIE, Stephen: Design and Implementation of the Second Extended Filesystem. In: *Proceedings of the First Dutch International Symposium on Linux*, 1995

[26] SILBERSCHATZ, Abraham ; GALVIN, Peter B. ; GAGNE, Greg: *Operating System Concepts*. New York, NY, USA : John Wiley & Sons, Inc., 2001

[27] STEVENS, W. R.: *Advanced programming in the UNIX environment*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 1992

[28] WALLI, Stephen R.: The POSIX family of standards. In: *StandardView* 3 (1995), Nr. 1, S. 11–17

[29] GALLMEISTER, Bill O.: *POSIX.4: programming for the real world*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1995

[30] GROPP, William ; HUSS-LEDERMAN, Steven ; LUMSDAINE, Andrew ; LUSK, Ewing ; NITZBERG, Bill ; SAPHIR, William ; SNIR, Marc: *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998

[31] GROPP, William ; LUSK, Ewing ; DOSS, Nathan ; SKJELLUM, Anthony: High-performance, portable implementation of the MPI Message Passing Interface Standard. In: *Parallel Computing* 22 (1996), Nr. 6, S. 789–828

[32] CORBETT, Peter ; HSU, Yarsun ; PROST, Jean-Pierre ; SNIR, Marc ; FINEBERG, Sam ; NITZBERG, Bill ; TRAVERSAT, Bernard ; WONG, Parkson ; FEITELSON, Dror: *MPI-IO: A Parallel File I/O Interface for MPI*. December 1995. – Version 0.4

[33] LATHAM, Rob ; MILLER, Neill ; ROSS, Robert ; CARNS, Phil: A Next-Generation Parallel File System for Linux Clusters. In: *LinuxWorld Magazine* (2004), 1, S. 56–59

[34] MICROSOFT CORPORATION: Microsoft Extensible Firmware Initiative - FAT32 File System Specification / Microsoft Corporation. Version: 2000. `http://www.microsoft.com/whdc/system/platform/firmware/fatgendown.mspx?` 2000. – Forschungsbericht

[35] PAWLOWSKI, B. ; JUSZCZAK, C. ; STAUBACH, P. ; SMITH, C. ; LEBEL, D. ; HITZ, D.: NFS Version 3: Design and Implementation. In: *Proceedings of Summer 1994 USENIX Conference*, 1994, S. 137 – 152

[36] MICROSYSTEMS, Sun: *NFS: Network File System Protocol specification*. RFC 1094 (Informational). `http://www.ietf.org/rfc/rfc1094.txt`. Version: März 1989 (Request for Comments)

[37] CALLAGHAN, B. ; PAWLOWSKI, B. ; STAUBACH, P.: *NFS Version 3 Protocol Specification*. RFC 1813 (Informational). `http://www.ietf.org/rfc/rfc1813.txt`. Version: Juni 1995 (Request for Comments)

[38] SHEPLER, S. ; CALLAGHAN, B. ; ROBINSON, D. ; THURLOW, R. ; BEAME, C. ; EISLER, M. ; NOVECK, D.: *Network File System (NFS) version 4 Protocol*. RFC 3530 (Proposed Standard). `http://www.ietf.org/rfc/rfc3530.txt`. Version: April 2003 (Request for Comments)

[39] RED HAT, INC.: *Global File System 5.1: Red Hat Global File System*. http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/en-US/RHEL%510/pdf/Global_File_System.pdf. Version: 2007

[40] GFS: *Red Hat Cluster Project Page*. http://sources.redhat.com/cluster. Version: 2005

[41] CLUSTER FILE SYSTEMS, INC.: *Lustre: A Scalable, High-Performance File System*. White Paper, 2005

[42] SUN MICROSYSTEMS: *Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System*. http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.%pdf. Version: December 2007

[43] CARNS, P. H. ; III, W. B. L. ; ROSS, R. B. ; THAKUR, R.: PVFS: A Parallel File System for Linux Clusters. In: *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, S. 317 – 327

[44] ABD-EL-BARR, Mostafa ; EL-REWINI, Hesham: *Fundamentals of Computer Organization and Architecture*. Wiley-IEEE, 2005

[45] HÄNDLER, W.: Standards, classification and taxonomy - Experiences with ECS. In: HÄNDLER, W. (Hrsg.) ; BLAAUW, G. (Hrsg.): *Workshop on Taxonomy in Computer Architecture*. Erlangen, 1981, S. 39–75

[46] HOCKNEY, Roger W.: *The Science of Computer Benchmarking*. Philadelphia : Society for Industrial and Applied Mathematics, 1996

[47] KRIETEMEYER, Michael ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: A Mathematical Model for the Transitional Region Between Cache Hierarchy Levels. In: BÖHME, Thomas (Hrsg.) ; LARIOS-ROSILLO, Victor (Hrsg.) ; UNGER, Helena (Hrsg.) ; UNGER, Herwig (Hrsg.): *IICS* Bd. 3473, Springer, 2004 (Lecture Notes in Computer Science), S. 178–188

[48] ALPERN, Bowen ; CARTER, Larry ; FEIG, Ephraim ; SELKER, Ted: The Uniform Memory Hierarchy Model of Computation. In: *Algorithmica* 12 (1994), Nr. 2/3, S. 72–109

[49] ELNAFFAR, S. ; MARTIN, P.: *Characterizing Computer Systems' Workloads*. 2002

[50] BELADY, L. A.: A Study of Replacement Algorithms for a Virtual Storage Computer. In: *IBM Systems Journal* 5 (1966), Nr. 2, S. 78–101

[51] CALZAROSSA, Maria ; SERAZZI, Giuseppe: Workload Characterization: A Survey. In: *Proc. IEEE* 81 (1993), Nr. 8, S. 1136–1150

[52] ROSTI, Emilia ; SERAZZI, Giuseppe ; SMIRNI, Evgenia ; SQUILLANTE, Mark S.: Models of Parallel Applications with Large Computation and I/O Requirements. In: *IEEE Trans. Softw. Eng.* 28 (2002), Nr. 3, S. 286–307

[53] ROSTI, Emilia ; SERAZZI, Giuseppe ; SMIRNI, Evgenia ; SQUILLANTE, Mark S.: The impact of I/O on program behavior and parallel scheduling. In: *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. New York, NY, USA : ACM Press, 1998, S. 56–65

[54] RAUBER, Thomas ; RÜNGER, Gudula: *Parallele und Verteilte Programmierung*. Springer Verlag, 2000

[55] BODE, A. ; HELLWAGNER, H.: Leistungsbewertung und Fehlertoleranz. In: RECHENBERG, P. (Hrsg.) ; POMBERGER, G. (Hrsg.): *Informatik-Handbuch*, Carl Hanser Verlag, 1999, S. 409–422

[56] RAU, Harald: *Mit Benchmarking an die Spitze*. Wiesbaden : Gabler Verlag, 1996

[57] SPENDOLINI, Michael J.: *The Benchmarking Book*. New York : Amacom Books, 1992

[58] KRIETEMEYER, Michael ; RISTAU, Henry ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: Chapter 7. In: MERZ, Matthias (Hrsg.) ; KRIETEMEYER, Michael (Hrsg.): *IPACS Benchmark - Integrated Performance Analysis of Computer Systems*, Logos Verlag, 2006, S. 121–156

[59] HERNANDEZ-GONZALEZ, Emilio: *A Methodology for the Design of Parallel Benchmarks*, University of Southhampton, Diss., 12 1996

[60] KRIETEMEYER, Michael ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: Workload-basierte Klassifikation von Benchmarks für lokale und verteilte I/O-Systeme. In: WOLFINGER, Bernd (Hrsg.) ; HEIDTMANN, Klaus (Hrsg.): *Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, Universität Hamburg, Fachbereich Informatik, 2005, S. 119–127

[61] CHEN, Peter M.: *Input/Output Performance Evaluation: Self-Scaling Benchmarks, Predicted Performance*. Berkeley, CA 94720, University of California, Berkeley, Dissertation, 1992

[62] CHEN, Peter M. ; PATTERSON, David A.: Storage Performance—Metrics and Benchmarks. In: *Proceedings of the IEEE, 81(8)*, 1993, S. 1151–1165

[63] *Bonnie Benchmark*. `http://www.textuality.com/bonnie`. Version: 2005

[64] *IOzone Filesystem Benchmark*. http://www.iozone.org/. Version: 2005

[65] RABENSEIFNER, Rolf ; KONIGES, Alice E. ; PROST, Jean-Pierre ; HEDGES, Richard: The Parallel Effective I/O Bandwidth Benchmark: b_eff_io. In: *2e soumission a Calculateurs paralleles*, 2001

[66] *FLASH I/O Benchmark*. http://www.ucolick.org/~zingale/flash_benchmark_io/. Version: 2006

[67] ROSS, Robert ; NURMI, Daniel ; CHENG, Albert ; ZINGALE, Michael: A Case Study in Applikation I/O on Linux Clusters. In: *Proceedings of the 2001 ACM/IEEE Conference in Supercomputing SC2001*. Denver, USA, 11 2001, 1–12

[68] CHING, Avery ; CHOUDHARY, Alok ; LIAO, Wei K. ; WARD, Lee ; PUNDIT, Neil: Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data. In: *Proceedings of the International Parallel and Distributed Processing Symposium*, 2006

[69] CHING, Avery ; CHOUDHARY, Alok ; COLOMA, Kenin ; LIAO, Wei keng ; ROSS, Robert ; GROPP, William: Noncontiguous I/O Accesses Through MPI-IO. In: *ccgrid* 00 (2003), S. 104

[70] NUNEZ, James: *LANL MPI-IO Test*. http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm. Version: 2006

[71] ROSS, Rob: Parallel I/O Benchmarking Consortium / Argonne National Laboratory. Version: 2002. http://www-unix.mcs.anl.gov/pio-benchmark/pio-benchmark.pdf. 2002. – Forschungsbericht

[72] BAI, Yihua ; WARD, Robert C.: A parallel symmetric block-tridiagonal divide-and-conquer algorithm. In: *ACM Trans. Math. Softw.* 33 (2007), Nr. 4, S. 25

[73] WONG, Parkson ; WIJNGAART, Rob F. d.: NAS Parallel Benchmarks I/O Version 2.4 / NASA Advanced Supercomputing (NAS) Division. 2003 (NAS-03-002). – Forschungsbericht

[74] LANCASTER, David: User Guide for the Parallel Input/Output Test Suite Version 1.0 / University of Southampton. 1999. – Forschungsbericht

[75] SHORTER, Frank: *Design and Analysis of a Performance Evaluation Standard for Parallel File Systems*, Clemson University, Diplomarbeit, 2003

[76] LAYTON, Jeff: A Benchmark for Parallel File Systems. In: *ClusterWorld Magazine* (2006)

[77] TAVIS BARR AND NICOLAI LANGFELDT AND SETH VIDAL AND TOM MCNEAL: *Linux NFS-HOWTO*. http://tldp.org/HOWTO/NFS-HOWTO/. Version: 2002

[78] ASSOCIATION, Storage Networking I.: *A Dictionary of Storage Networking Terminology - Version v.2006.2.ENG*. `http://www.snia.org/education/dictionary/`. Version: 2006

[79] BRYANT, Ray ; HAWKES, John: Linux® Scalability for Large NUMA Systems. In: *Proceedings of the Linux Symposium*, 2003

[80] CHEN, Peter M. ; PATTERSON, David A.: A new Approach to I/O Performance Evaluation—Self-Scaling I/O Benchmarks, Predicted I/O Performance. In: *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Santa Clara, CA, USA, 10–14 1993, S. 1–12

[81] CHEN, Peter M. ; PATTERSON, David A.: Unix I/O Performance in Workstations and Mainframes. – Forschungsbericht

[82] KOPP, Heiko ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: Ein Lastmodell für die Definition von I/O-Lasten beim Zugriff auf Sekundärspeicher paralleler Systeme. In: WOLFINGER, Bernd (Hrsg.) ; HEIDTMANN, Klaus (Hrsg.): *Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, Universität Hamburg, Fachbereich Informatik, 2007

[83] FORUM, Message Passing I.: MPI-2: Extensions to the Message-Passing Interface / University of Tennessee. Knoxville, nov 2003. – Forschungsbericht

[84] NIEUWEJAAR, Nils ; KOTZ, David ; PURAKAYASTHA, Apratim ; ELLIS, Carla S. ; BEST, Michael: File-Access Characteristics of Parallel Scientific Workloads. In: *IEEE Transactions on Parallel and Distributed Systems* 7 (1996), Nr. 10, S. 1075–1089

[85] SMIRNI, E. ; REED, D. A.: Workload characterization of input/output intensive parallel applications. In: *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation* Bd. 1245, Springer-Verlag, 1997, 169–180

[86] CRANDALL, Phyllis E. ; AYDT, Ruth A. ; CHIEN, Andrew A. ; REED, Daniel A.: Input/output characteristics of scalable parallel applications. In: *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA : ACM, 1995, S. 59

[87] SMIRNI, Evgenia ; REED, Daniel: Lessons from Characterizing Input/Output Behavior of Parallel Scientific Applications. In: *Performance Evaluation* 1245 (1998), November, S. 169–180

[88] BRONSTEIN, I. N. ; SEMENDJAEV, K. A. ; MUSIOL, G. ; MÜHLIG, H.: *Taschenbuch der Mathematik*. 5. Frankfurt am Main : Verlag Harri Deutsch, 2000

[89] GROUP, Object M.: *OMG Unified Modeling Language (OMG UML), Superstructure*. http://www.omg.org/spec/UML/2.2/Superstructure/PDF. `http://www.omg.org/spec/UML/2.2/Superstructure/PDF`. Version: Mai 2008

[90] HERRMANN, Paul: *Rechnerarchitektur*. Wiesbaden : Vieweg+Teubner Verlag, 2002

[91] *Kapitel* Parallel Programming Models and Paradigms. In: SILVA, Luis M. ; BUYYA, Rajkumar: *High Performance Cluster Computing: Programming and Applications*. Bd. 2. Prentice Hall, 1999, S. 4–27

[92] FALCONE, Giovanni ; KREDEL, Heinz ; KRIETEMEYER, Michael ; MERTEN, Dirk ; MERZ, Matthias ; PFREUNDT, Franz-Joseph ; SIMMENDINGER, Christian ; VERSICK, Daniel: Integrated Performance Analysis of Computer Systems (IPACS). In: *Praxis der Informationsverarbeitung und Kommunikation* (2005), Nr. 3, S. 154–163

[93] ROSE, Luiz A. ; REED, Daniel A.: SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In: *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*. Washington, DC, USA : IEEE Computer Society, 1999, S. 311

[94] BEHNKE, Ralf ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: Leistungsvermessung von I/O Systemen mit der MPI-IO-Schnittstelle. In: HOEFLER, Torsten (Hrsg.) ; MEHLAN, Torsten (Hrsg.) ; REHM, Wolfgang (Hrsg.): *Kommunikation in Clusterrechnern und Clusterverbundsystemen, Tagungsband zum 2. Workshop*. Chemnitz, Februar 2007

[95] W3C-RECOMMENDATION: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. http://www.w3.org/TR/2006/REC-xml-20060816/. `http://www.w3.org/TR/2006/REC-xml-20060816/`. Version: 09 2006

[96] SHAFRANOVICH, Y.: *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180 (Informational). `http://www.ietf.org/rfc/rfc4180.txt`. Version: Oktober 2005 (Request for Comments)

[97] RATAJ-WEINREBEN, Christian: *Entwicklung eines Workload-Modells für verteilte I/O-Systeme*. Rostock, Universität Rostock, Diplomarbeit, 09 2006

[98] KRIETEMEYER, Michael ; KOPP, Heiko ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: Environment for I/O Performance Measurement of Distributed and Local Secondary Storage. In: *Proceedings of the International Conference on Parallel Processing Workshops*, IEEE, 2005, S. 501–508

[99] KRIETEMEYER, Michael ; VERSICK, Daniel ; TAVANGARIAN, Djamshid: The PRIOmark – Parallel I/O Benchmark. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, 2005, S. 595 – 600

[100] MOLECULAR SCIENCES SOFTWARE GROUP: *NWChem Home Page*. `http://www.emsl.pnl.gov/docs/nwchem/nwchem.html`. Version: 2007

[101] MOLECULAR SCIENCES SOFTWARE GROUP: *NWChem User Documentation Release 5.1*. `http://www.emsl.pnl.gov/docs/nwchem/doc/user/index.html`. Version: 2007

[102] PACIFIC NORTHWEST NATIONAL LABORATORY: *Pacific Northwest National Laboratory - Homepage*. `http://www.pnl.gov`. Version: 2008

[103] THOMPSON, K. ; MILLER, G. ; WILDER, R.: Wide-Area Internet Traffic Patterns and Characteristics (Extended Version). In: *IEEE Network* (1997), November, S. 10–23

[104] BASHER, Naimul: *Characterization of Peer-to-Peer and Web Traffic at a Network Edge*, University of Calgary, Diplomarbeit, June 2007

[105] PLOUMIDIS, Manolis ; PAPADOPOULI, Maria ; KARAGIANNIS, Thomas: Multi-level application-based traffic characterization in a large-scale wireless network. In: *WOWMOM*, IEEE, 2007, S. 1–9

[106] BRYANT, Ray ; RADDATZ, Dave ; SUNSHINE, Roger: PenguinoMeter: a new file-I/O benchmark for Linux. In: *ALS '01: Proceedings of the 5th annual conference on Linux Showcase & Conference*. Berkeley, CA, USA : USENIX Association, 2001, S. 10–10

[107] THE APACHE SOFTWARE FOUNDATION: *Dokumentation zum Apache HTTP Server Version 2.0*. `http://httpd.apache.org/docs/2.0/`. Version: 2008

[108] HSU, W. W. ; SMITH, A. J.: Characteristics of I/O traffic in personal computer and server workloads. In: *IBM Systems Journal* 42 (2003), Nr. 2, S. 347–372

[109] ROCKS CLUSTERS: *Rocks Clusters Homepage*. `http://www.rocksclusters.org`. Version: 2008

[110] NATIONAL SCIENCE FOUNDATION: *US NSF - National Science Foundation*. `http://www.nsf.gov`. Version: 2008

[111] GROPP, William: MPICH2: A New Start for MPI Implementations. In: *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK : Springer-Verlag, 2002, S. 7

[112] ROB LATHAM AND ROB ROSS: *Noncontiguous MPI-IO Performance on PVFS*. http://www-unix.mcs.anl.gov/hpio/papers/noncontig-perf.pdf