UNITED STATES PATENT AND TRADEMARK OFFICE

_____

BEFORE THE PATENT TRIAL AND APPEAL BOARD

_____

NetApp, Inc., Rackspace US, Inc.,

Petitioner

v.

Realtime Data LLC

Patent Owner

_____

_____

DECLARATION OF SCOTT BENNETT, Ph.D.

1 May 2017

I, Scott Bennett, Ph.D., resident of Urbana, Illinois, hereby declare as follows:

Introduction and Qualifications

1. I have been retained by Morrison & Foerster LLP to provide my opinions concerning the public availability of certain documents at issue in an *inter partes* review proceeding.

2. My curriculum vitae is appended to this document as Appendix A. From 1956 to 1960, I attended Oberlin College, where I received an A.B. in English. I then attended Indiana University, where I received an M.A. in 1966 and a Ph.D. in 1967, both in English. In 1976, I received a M.S. in Library Science from the University of Illinois. I also served at the University of Illinois at Urbana-Champaign in two capacities. First, from 1967 to 1974, I was an Assistant Professor of English; then from 1974 to 1981, I was successively an Instructor, Assistant Professor, and Associate Professor of Library Science.

3. From 1981 to 1989, I served as the Assistant University Librarian for Collection Management, Northwestern University. From 1989 to 1994, I served as the Director of The Milton S. Eisenhower Library at The Johns Hopkins University. From 1994 to 2001, I served as the University Librarian at Yale University. In 2001, I retired from Yale University.

4. Since then, I have served in multiple capacities for various organizations, including as a consultant on library space planning from 2004 to the present, as a Senior Advisor for the library program of the Council of Independent Colleges from 2001 to 2009, as a member of the Wartburg College Library Advisory Board from 2004 to the present, and as a Visiting Professor at the Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, in the Fall of 2003. I was a founding partner of Prior Art Documentation Services, LLC, in 2015.

5. Over the course of my work as a librarian, professor, researcher, and author of numerous publications, I have had extensive experience with cataloging and online library management systems built around Machine-Readable Cataloging (MARC) standards. As a consultant, I have substantial experience in authenticating documents and establishing the date when they were available to persons exercising reasonable diligence.

6. In the course of more than fifty years of academic life, I have myself been an active researcher. I have collaborated with many individual researchers and, as a librarian, worked in the services of thousands of researchers at four prominent research universities. Members of my family are university researchers. Over the years, I have read some of the voluminous professional literature on the information seeking behaviors of academic researchers. And as an educator, I

have a broad knowledge of the ways in which students in a variety of disciplines learn to master the bibliographic resources used in their disciplines. In all of these ways, I have a general knowledge of the how researchers work.

7. My work in this matter is being billed at my standard consulting rate of $91 per hour. My compensation is not in any way contingent upon the outcome of this or any other *inter partes* review. I have no financial or personal interest in the outcome of this proceeding or any related litigation.

Scope of this Declaration

8. I am not a lawyer and I am not rendering an opinion on the legal question of whether any particular document is, or is not, a "printed publication" under the law.

9. I am, however, rendering my expert opinion on when and how each of the documents addressed herein was disseminated or otherwise made available to the extent that persons interested and ordinarily skilled in the subject matter or art, exercising reasonable diligence, could have located the documents before December 11, 1998.

10. I reserve the right to supplement my opinion in the future to respond to any arguments that the Patent Owner raises and to take into account new information as it becomes available.

Materials Considered in Forming My Opinion

11. In forming the opinions expressed in this declaration, I have reviewed the document and attachments referenced below. These materials are records created in the ordinary course of business by publishers, libraries, indexing services, and others. From my years of experience, I am familiar with the process for creating many of these records, and I know these records are created by people with knowledge of the information in the record. Further, these records are created with the expectation that researchers and other members of the public will use them. All materials cited in this declaration and its attachments are of a type that experts in my field would reasonably rely upon and refer to in forming their opinions.

Document 1. William H. Hsu and Amy E. Zwarico, "Automatic Synthesis of Compression Techniques for Heterogeneous Files," Software: Practice & Experience, 25,10 (October 1995): 1097-1116.

12. The following Attachments are true and accurate representations of library material and online documents and records, as they are identified below. All attachments were secured on 17-21 October 2016. All URLs were available on 19 October 2016.

Attachment 1a: Statewide Illinois Library Catalog record for Software: Practice & Experience

Attachment 1b: Depaul University Library catalog record for Software: Practice & Experience

Attachment 1c: Copy of Hsu from the Depaul University Library

Attachment 1d: Wiley Online Library index record for Hsu

Attachment 1e: Copy of Hsu from the Wiley Online Library

Attachment 1f: Copy of Hsu from the University of Illinois at Urbana-Champaign Library

Attachment 1g: Copy of Hsu from the University of Minnesota Library

Attachment 1h: Copy of Hsu from the Illinois Institute of Technology Library

13. Helen Sullivan is a Managing Partner in Prior Art Documentation Services LLC (see http://www.priorartdocumentation.com/hellen-sullivan/ **)**. Her primarily responsibility in our partnership is to secure the bibliographic documentation used in attachments to our declarations. Ms. Sullivan secured all of the attachments listed above, except Attachment 1h, which I secured. I have carefully reviewed the bibliographic documentation used in my declaration. My signature on the declaration indicates my full confidence in the authenticity, accuracy, and reliability of the bibliographic documentation used.

Background Information

14. *Persons of ordinary skill in the art*. I am told by counsel that the subject matter of this proceeding relates to systems and methods of data compression.

15. I am told by counsel that persons of ordinary skill in this subject matter or art would have had an undergraduate degree in either computer science, computer engineering, electrical and computer engineering, or an equivalent field and one to three years of experience working with data compression, or a graduate degree with course work or research in the field of data compression. Individuals with additional education or additional industrial experience could still be of ordinary skill in the art if that additional education or experience compensates for a deficit in one of the other aspects of the requirements stated above.

16. It is my opinion that such a person would have been engaged in advanced research starting at least in graduate school, learning though study and practice in the field and possibly through formal instruction the bibliographic resources relevant to his or her research. In the 1980s and 1990s such a person would have had access to a vast array of long-established print resources in electrical/computer engineering and computer science as well as to a rich and fast changing set of online resources providing indexing information, abstracts, and full text services for electrical/computer engineering and computer science.

17. *Library catalog records*. WorldCat is the world's largest public online catalog, maintained by the Online Computer Library Center, Inc., or OCLC, and built with the records created by the thousands of libraries that are members of OCLC. WorldCat records appear in many different catalogs, including the Statewide Illinois Library Catalog.

18. *Periodical publications*. A library typically creates a catalog record for a periodical publication when the library receives its first issue. When the institution receives subsequent issues/volumes of the periodical, the issues/volumes are checked in (often using a date stamp), added to the institution's holdings records, and made available very soon thereafter—normally within a few days of receipt or (at most) within a few weeks of receipt.

19. The initial periodicals record will sometimes not reflect all of the subsequent changes in publication details (including minor variations in title, etc.).

20. *Indexing*. An ordinarily skilled researcher may discover material relevant to his or her topic in a variety of ways. One common means of discovery is to search for relevant information in an index of periodical and other publications. Having found relevant material, the researcher will then normally obtain it online, look for it in libraries, or purchase it from the publisher, a bookstore, or other provider.

21. Indexing services commonly provide bibliographic information, abstracts, and full-text copies of the indexed publications, along with a list of the documents cited in the indexed publication. Prominent indexing services include the Wiley Online Library, a multidisciplinary collection of online resources in the life, health and physical sciences, and in the social sciences and humanities (see http://olabout.wiley.com/WileyCDA/Section/id-390001.html).

Consideration of individual documents

Document 1. William H. Hsu and Amy E. Zwarico, "Automatic Synthesis of Compression Techniques for Heterogeneous Files," Software: Practice & Experience, 25,10 (October 1995): 1097-1116.

Authentication

22. Document 1 is a paper written by William Hsu and Amy Zwarico and published in the October 1995 issue of Software: Practice & Experience. This paper is herein referred to Hsu. It is marked as Exhibit 1002 to the petition for *inter partes* review.

23. Attachment 1a is a true and accurate copy of the Statewide Illinois Library Catalog record for Software: Practice & Experience. This record shows that Software: Practice & Experience began publication in 1971 and is held by 598 libraries world-wide. An ordinarily skilled researcher would have no difficulty identifying and locating library copies of this periodical.

24. The DePaul University Library is one library holding this periodical. Attachment 1b is a true and accurate copy of the DePaul University Library catalog record for Software: Practice & Experience, showing the DePaul University Library holdings for Software: Practice & Experience include volume 25, number 10.

25. Attachment 1c is a true and accurate copy, in black and white, of Hsu from the DePaul University Library. Attachment 1c includes the cover for the October issue of Software: Practice & Experience, the contents page, and the Hsu paper on pp. 1097-1116. Attachment 1c is in a condition that creates no suspicion about its authenticity. Specifically, there are no visible alterations to the document, and Attachment 1c was found within the custody of a library – a place where, if authentic, it would likely be found.

26. Attachment 1d is a true and accurate copy of the item record for Hsu in the Wiley Online Library. Attachment 1e is a true and accurate copy of Hsu from the Wiley Online Library. This online version of Hsu is identical to Attachment 1c and includes, on the first page, the bibliographic information about the publication of Hsu. Software: Practice & Experience is a Wiley publication. Attachments 1d and 1e were found in the Wiley Online Library—a place where, if authentic, they would likely be found.

27. Based on finding Hus in a library and online and on finding library catalog and online index records for Hus, I conclude that Hus is an authentic document and that Attachment 1a is an authentic .copy of Hsu.

Public accessibility

28. Attachment 1c includes a library date stamp label indicating that October 1995 issue of Software: Practice & Experience was processed at the DePaul University Library on 25 October 1995. Based on my experience, I affirm this date stamp has the general appearance of date stamps that libraries have long affixed to periodicals in processing them. I do not see any indications or have any reason to believe this date stamp was affixed by anyone other than library personnel on or about the date indicated by the stamp.

29. This date stamp indicates the October 1995 issue of Software: Practice & Experience had been mailed to the DePaul University Library and to other subscribers (including other library subscribers) sometime in October 1995, or earlier, because it takes some time for the item to arrive at and to be processed by the library. I therefore conclude that the October 1995 issue of Software: Practice & Experience would have been received by other subscribers, and that other subscribing libraries would have processed and made this issue available to their readers at about the same time.

30. For example, Attachment 1f is a true and accurate copy, in color, of Hsu from the University of Illinois at Urbana-Champaign Library. Attachment 1f includes the bound volume cover; the covers for the July, August, September, October, November, and December issues of Software: Practice & Experience; and from the October issue a list of editors and other information about the journal, the contents page, and the Hsu paper on pp. 1097-1116.

31. Attachment 1f includes a library date stamp label indicating that July 1995 issue of Software: Practice & Experience was processed at the University of Illinois at Urbana-Champaign Library on 24 July 1995. Similar date stamp labels on the August, November, and December issues indicate they were processed at the library on 22 August, 27 November, and 14 December, respectively. Date stamps can be detected on the covers of the September and October issues of Software: Practice & Experience, but they are hard to read, even under magnification. The September issue appears to have been processed by the University of Illinois at Urbana-Champaign Library on 2? September 1995, while the October issue appears to have been processed on ??October ????. The text here states only the date information that can be made out, as these stamps are difficult to read. This bound volume of Software: Practice & Experience, including the October 1995 issue, also bears on its inside back cover a February

1996 sticker from the Heckman Bindery, Inc., a major provider of periodical binding services.

32. Based on these date markings, it is my opinion that in the second half of 1995 the University of Illinois at Urbana Champaign regularly processed newly received issues of Software: Practice & Experience in the second or third week of the month indicated on the cover of each monthly issue. I also infer that all issues of Software: Practice & Experience published in the second half of 1995 where in hand at the University of Illinois at Urbana-Champaign Library by December 1995 or January 1996, when they were send to the Heckman Bindery to be bound as a volume.

33. Attachment 1g is a third true and accurate copy, in color, of Hsu—this one from the University of Minnesota Library. This copy includes the cover, a list of editors and other information about the journal, the contents page, and the Hsu paper. The cover page in Attachment 1g includes a library date stamp that is hard to read. Under magnification, this date stamp indicates the October 1995 issue of Software: Practice & Experience was processed at the University of Minnesota Library on 20 October [1995], in close conformity to the evidence for processing issues of Software: Practice & Experience at the DePaul University Library and the University of Illinois at Urbana-Champaign Library. The year element in the date stamp is very hard to read, even when magnified, and thus I have listed it here in

brackets. However, Software: Practice & Experience was published monthly (as indicated on the page listing the journal's editors). In my experience, it would have been highly unusual for a library, such as the University of Minnesota Library, to have received this periodical a year later, in 1996. Thus, considering both my experience and the facts outlined in this declaration, it is my opinion that while the date stamp is only partially illegible, this volume was stamped received by the University of Minnesota Library on 20 October 1995.

34. Attachment 1h is a fourth true and accurate copy, in color, of Hsu—this one from the Illinois Institute of Technology Library. This copy includes the spine of volume 25 and the cover of the October 1995 issue, a list of editors and other information about the journal, the contents page, and the Hsu paper. The cover page in Attachment 1h in has a date stamp that indicates the October 1995 issue of Software: Practice & Experience was processed at the Illinois Institute of Technology Library on 3 November 1995, in near conformity to the evidence for processing October 1995 issues of Software: Practice & Experience at the DePaul University Library, the University of Illinois at Urbana-Champaign Library, and the University of Minnesota Library.

35. The copies of Hsu in Attachments 1c and 1f, 1g, and 1h from the DePaul University Library, University of Illinois at Urbana-Champaign Library,

the University of Minnesota Library, and the Illinois Institute of Technology Library, respectively, are substantively identical.

36. The evidence from four academic libraries indicates the October 1995 issue of Software: Practice & Experience was mailed to subscribers in October 1995, or earlier, and processed by these four libraries late in October or early in November 1995. Allowing for some time between the date stamping of the October 1995 issue of Software: Practice and Experience and its appearance on library shelves, where it would be public available, it is my opinion that Hsu was publicly available at least by mid-November 1995.

Conclusion

37. Based on the evidence presented here—publication in an easily identified and widely held periodical, online availability, and library date stamps—**it is my opinion that Document 1 is an authentic document and was available to the public at least by mid-November 1995**.

Attestation

38. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statement were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code

and that such willful false statement may jeopardize the validity of the application or any patent issued thereon.

_____ 1 May 2017

Scott Bennett, Ph.D. Date
Managing Partner
Prior Art Documentation Services LLC

EXHIBIT A: RESUME

SCOTT BENNETT
Yale University Librarian Emeritus

711 South Race
Urbana, Illinois 61801-4132
2scottb@prairienet.org
217-367-9896

EMPLOYMENT

Retired, 2001. Retirement activities include:

- Managing Partner in Prior Art Documentation Services, LLC, 2015-. This firm provides documentation services to patent attorneys; more information is available at http://www.priorartdocumentation.com
- Consultant on library space design, 2004- . This consulting practice is rooted in a research, publication, and public speaking program conducted since I retired from Yale University in 2001. I have served more than 50 colleges and universities in the United States and abroad with projects ranging in likely cost from under $50,000 to over $100 million. More information is available at http://www.libraryspaceplanning.com/
- Senior Advisor for the library program of the **Council of Independent Colleges**, 2001-2009
- Member of the Wartburg College Library Advisory Board, 2004-
- Visiting Professor, Graduate School of Library and Information Science, **University of Illinois at Urbana-Champaign**, Fall 2003

University Librarian, **Yale University**, 1994-2001

Director, The Milton S. Eisenhower Library, **The Johns Hopkins University**, Baltimore, Maryland, 1989-1994

Assistant University Librarian for Collection Management, **Northwestern University**, Evanston, Illinois, 1981-1989

Instructor, Assistant and Associate Professor of Library Administration**, University of Illinois at Urbana-Champaign**, 1974-1981

Assistant Professor of English, **University of Illinois at Urbana-Champaign**, 1967-1974

Woodrow Wilson Teaching Intern, **St. Paul's College**, Lawrenceville, Virginia, 1964-1965

EDUCATION

University of Illinois, M.S., 1976 (Library Science)
Indiana University, M.A., 1966; Ph.D., 1967 (English)
Oberlin College, A.B. magna cum laude, 1960 (English)

HONORS AND AWARDS

Morningside College (Sioux City, IA) Doctor of Humane Letters, 2010

American Council of Learned Societies Fellowship, 1978-1979; Honorary Visiting Research Fellow, Victorian Studies Centre**, University of Leicester**, 1979; **University of Illinois** Summer Faculty Fellowship, 1969

Indiana University Dissertation Year Fellowship and an **Oberlin College** Haskell Fellowship, 1966-1967; **Woodrow Wilson** National Fellow, 1960-1961

PROFESSIONAL ACTIVITIES

American Association for the Advancement of Science: Project on Intellectual Property and Electronic Publishing in Science, 1999-2001

American Association of University Professors: University of Illinois at Urbana-Champaign Chapter Secretary and President, 1975-1978; Illinois Conference Vice President and President, 1978-1984; national Council, 1982-1985, Committee F, 1982-1986, Assembly of State Conferences Executive Committee, 1983-1986, and Committee H, 1997-2001 ; Northwestern University Chapter Secretary/Treasurer, 1985-1986

Association of American Universities: Member of the Research Libraries Task Force on Intellectual Property Rights in an Electronic Environment, 1993-1994, 1995-1996

Association of Research Libraries: Member of the Preservation Committee, 1990-1993; member of the Information Policy Committee, 1993-1995; member of the Working Group on Copyright, 1994-2001; member of the Research Library Leadership and Management Committee, 1999-2001; member of the Board of Directors, 1998-2000

Carnegie Mellon University: Member of the University Libraries Advisory Board, 1994

Center for Research Libraries: Program Committee, 1998-2000

Johns Hopkins University Press: Ex-officio member of the Editorial Board, 1990-1994; Co-director of Project Muse, 1994

Library Administration and Management Association, Public Relations Section, Friends of the Library Committee, 1977-1978

Oberlin College: Member of the Library Visiting Committee, 1990, and of the Steering Committee for the library's capital campaign, 1992-1993; President of the Library Friends, 1992-1993, 2004-2005; member, Friends of the Library Council, 2003-

Research Society for Victorian Periodicals: Executive Board, 1971-1983; Co-chairperson of the Executive Committee on Serials Bibliography, 1976-1982; President, 1977-1982

A Selected Edition of W.D. Howells (one of several editions sponsored by the MLA Center for Editions of American Authors): Associate Textual Editor, 1965-1970; Center for Editions of American Authors panel of textual experts, 1968-1970

Victorian Studies: Editorial Assistant and Managing Editor, 1962-1964

Wartburg College: member, National Advisory Board for the Vogel Library, 2004-

Some other activities: Member of the **Illinois State Library** Statewide Library and Archival Preservation Advisory Panel; member of the **Illinois State Archives** Advisory Board; member of a committee advising the **Illinois Board of Higher Education** on the cooperative management of research collections; chair of a major collaborative research project conducted by the **Research Libraries Group** with support from Conoco, Inc.; active advisor on behalf of the **Illinois Conference AAUP** to faculty and administrators on academic freedom and tenure matters in northern Illinois.

Delegate to **Maryland Governor's Conference on Libraries and Information Service**; principal in initiating state-wide preservation planning in Maryland; principal in an effort to widen the use of mass deacidification for the preservation of library materials through cooperative action by the **Association of Research Libraries** and the **Committee on Institutional Cooperation**; co-instigator of a campus-wide information service for **Johns Hopkins University**; initiated efforts with the **Enoch Pratt Free Library** to provide information services to Baltimore's Empowerment Zones; speaker or panelist on academic publishing, copyright, scholarly communication, national and regional preservation planning, mass deacidification.

Consultant for the **University of British Columbia** (1995), **Princeton University** (1996), **Modern Language Association**, (1995, 1996), **Library of Congress** (1997), **Center for Jewish History** (1998, 2000-), **National Research Council** (1998); Board of Directors for the **Digital Library Federation**, 1996-2001; accreditation visiting team at **Brandeis University** (1997); mentor for **Northern Exposure to Leadership** (1997); instructor and mentor for ARL's **Leadership and Career Development Program** (1999-2000)

At the **Northwestern University Library**, led in the creation of a preservation department and in the renovation of the renovation, for preservation purposes, of the Deering Library book stacks.

At the **Milton S. Eisenhower Library**, led the refocusing and vitalization of client-centered services; strategic planning and organizational restructuring for the library; building renovation planning. Successfully completed a $5 million endowment campaign for the humanities collections and launched a $27 million capital campaign for the library.

At the **Yale University Library**, participated widely in campus-space planning, university budget planning, information technology development, and the promotion of effective teaching and learning; for the library has exercised leadership in space planning and renovation, retrospective conversion of the card catalog, preservation, organizational development, recruitment of minority librarians, intellectual property and copyright issues, scholarly communication, document delivery services among libraries, and instruction in the use of information resources. Oversaw approximately $70 million of library space renovation and construction. Was co-principal investigator for a grant to plan a digital archive for Elsevier Science.

Numerous to invitations speak at regional, national, and other professional meetings and at alumni meetings. Lectured and presented a series of seminars on library management at the **Yunnan University Library**, 2002. Participated in the 2005 International Roundtable for Library and Information Science sponsored by the **Kanazawa Institute of Technology** Library Center and the Council on Library and Information Resources.

PUBLICATIONS

"Putting Learning into Library Planning," *portal: Libraries and the Academy*, 15, 2 (April 2015), 215-231.

"How librarians (and others!) love silos: Three stories from the field " available at the Learning Spaces Collaborary Web site, http://www.pkallsc.org/

"Learning Behaviors and Learning Spaces," *portal: Libraries and the Academy,* 11, 3 (July 2011), 765-789.

"Libraries and Learning: A History of Paradigm Change," *portal: Libraries and the Academy,* 9, 2 (April 2009), 181-197. Judged as the best article published in the 2009 volume of *portal*.

"The Information or the Learning Commons: Which Will We Have?" *Journal of Academic Librarianship*, 34 (May 2008), 183-185. One of the ten most-cited articles published in JAL, 2007-2011.

"Designing for Uncertainty: Three Approaches," *Journal of Academic Librarianship*, 33 (2007), 165–179.

"Campus Cultures Fostering Information Literacy," *portal: Libraries and the Academy*, 7 (2007), 147-167. Included in Library Instruction Round Table Top Twenty library instruction articles published in 2007

"Designing for Uncertainty: Three Approaches," *Journal of Academic Librarianship*, 33 (2007), 165–179.

"First Questions for Designing Higher Education Learning Spaces," *Journal of Academic Librarianship*, 33 (2007), 14-26.

"The Choice for Learning," *Journal of Academic Librarianship*, 32 (2006), 3-13.

With Richard A. O'Connor, "The Power of Place in Learning," *Planning for Higher Education*, 33 (June-August 2005), 28-30

"Righting the Balance," in *Library as Place: Rethinking Roles, Rethinking Space* (Washington, DC: Council on Library and Information Resources, 2005), pp. 10-24

Libraries Designed for Learning (Washington, DC: Council on Library and Information Resources, 2003)

"The Golden Age of Libraries," in *Proceedings of the International Conference on Academic Librarianship in the New Millennium: Roles, Trends, and Global Collaboration*, ed. Haipeng Li (Kunming: Yunnan University Press, 2002), pp. 13-21. This is a slightly different version of the following item.

"The Golden Age of Libraries," *Journal of Academic Librarianship*, 24 (2001), 256-258

"Second Chances. An address . . . at the annual dinner of the Friends of the Oberlin College Library November 13 1999," Friends of the Oberlin College Library, February 2000

"Authors' Rights," *The Journal of Electronic Publishing* (December 1999), http://www.press.umich.edu/jep/05-02/bennett.html

"Information-Based Productivity," in *Technology and Scholarly Communication*, ed. Richard Ekman and Richard E. Quandt (Berkeley, 1999), pp. 73-94

"Just-In-Time Scholarly Monographs: or, Is There a Cavalry Bugle Call for Beleaguered Authors and Publishers?" *The Journal of Electronic Publishing* (September 1998), http://www.press.umich.edu/jep/04-01/bennett.html

"Re-engineering Scholarly Communication: Thoughts Addressed to Authors," *Scholarly Publishing*, 27 (1996), 185-196

"The Copyright Challenge: Strengthening the Public Interest in the Digital Age," *Library Journal*, 15 November 1994, pp. 34-37

"The Management of Intellectual Property," *Computers in Libraries*, 14 (May 1994), 18-20

"Repositioning University Presses in Scholarly Communication," *Journal of Scholarly Publishing*, 25 (1994), 243-248. Reprinted in *The Essential JSP. Critical Insights into the World of Scholarly Publishing. Volume 1: University Presses* (Toronto: University of Toronto Press, 2011), pp. 147-153

"Preservation and the Economic Investment Model," in *Preservation Research and Development. Round Table Proceedings, September 28-29, 1992*, ed. Carrie Beyer (Washington, D.C.: Library of Congress, 1993), pp. 17-18

"Copyright and Innovation in Electronic Publishing: A Commentary," *Journal of Academic Librarianship*, 19 (1993), 87-91; reprinted in condensed form in *Library Issues: Briefings for Faculty and Administrators*, 14 (September 1993)

with Nina Matheson, "Scholarly Articles: Valuable Commodities for Universities," *Chronicle of Higher Education*, 27 May 1992, pp. B1-B3

"Strategies for Increasing [Preservation] Productivity*," Minutes of the [119th] Meeting [of the Association of Research Libraries]* (Washington, D.C., 1992), pp. 39-40

"Management Issues: The Director's Perspective," and "Cooperative Approaches to Mass Deacidification: Mid-Atlantic Region," in *A Roundtable on Mass Deacidification*, ed. Peter G. Sparks (Washington, D.C.: Association of Research Libraries, 1992), pp. 15-18, 54-55

"The Boat that Must Stay Afloat: Academic Libraries in Hard Times," *Scholarly Publishing*, 23 (1992), 131-137

"Buying Time: An Alternative for the Preservation of Library Material," ACLS *Newsletter*, Second Series 3 (Summer, 1991), 10-11

"The Golden Stain of Time: Preserving Victorian Periodicals" in *Investigating Victorian Journalism*, ed. Laurel Brake, Alex Jones, and Lionel Madden (London: Macmillan, 1990), pp. 166-183

"Commentary on the Stephens and Haley Papers" in *Coordinating Cooperative Collection Development: A National Perspective*, an issue of *Resource Sharing and Information Networks*, 2 (1985), 199-201

"The Editorial Character and Readership of *The Penny Magazine*: An Analysis," *Victorian Periodicals Review*, 17 (1984), 127-141

"Current Initiatives and Issues in Collection Management," *Journal of Academic Librarianship*, 10 (1984), 257-261; reprinted in *Library Lit: The Best of 85*

"Revolutions in Thought: Serial Publication and the Mass Market for Reading" in *The Victorian Periodical Press: Samplings and Soundings*, ed. Joanne Shattock and Michael Wolff (Leicester: Leicester University Press, 1982), pp. 225-257

"Victorian Newspaper Advertising: Counting What Counts," *Publishing History*, 8 (1980), 5-18

"Library Friends: A Theoretical History" in *Organizing the Library's Support: Donors, Volunteers, Friends*, ed. D.W. Krummel, Allerton Park Institute Number 25 (Urbana: University of Illinois Graduate School of Library Science, 1980), pp. 23-32

"The Learned Professor: being a brief account of a scholar [Harris Francis Fletcher] who asked for the Moon, and got it," *Non Solus*, 7 (1980), 5-12

"Prolegomenon to Serials Bibliography: A Report to the [Research] Society [for Victorian Periodicals]," *Victorian Periodicals Review*, 12 (1979), 3-15

"The Bibliographic Control of Victorian Periodicals" in *Victorian Periodicals: A Guide to Research*, ed. J. Don Vann and Rosemary T. VanArsdel (New York: Modern Language Association, 1978), pp. 21-51

"John Murray's Family Library and the Cheapening of Books in Early Nineteenth Century Britain," *Studies in Bibliography*, 29 (1976), 139-166. Reprinted in Stephen Colclough and Alexis Weedon, eds., *The History of the Book in the West: 1800-1914*, Vol. 4 (Farnham, Surrey: Ashgate, 2010), pp. 307-334.

with Robert Carringer, "Dreiser to Sandburg: Three Unpublished Letters," *Library Chronicle*, 40 (1976), 252-256

"David Douglas and the British Publication of W. D. Howells' Works," *Studies in Bibliography*, 25 (1972), 107-124

as primary editor, W. D. Howells, *Indian Summer* (Bloomington: Indiana University Press, 1971)

"The Profession of Authorship: Some Problems for Descriptive Bibliography" in *Research Methods in Librarianship: Historical and Bibliographic Methods in Library Research*, ed. Rolland E. Stevens (Urbana: University of Illinois Graduate School of Library Science, 1971), pp. 74-85

edited with Ronald Gottesman, *Art and Error: Modern Textual Editing* (Bloomington: Indiana University Press, 1970)--also published in London by Methuen, 1970

"Catholic Emancipation, the *Quarterly Review*, and Britain's Constitutional Revolution," *Victorian Studies*, 12 (1969), 283-304

as textual editor, W. D. Howells, *The Altrurian Romances* (Bloomington: Indiana University Press, 1968); introduction and annotation by Clara and Rudolf Kirk

as associate textual editor, W. D. Howells, *Their Wedding Journey* (Bloomington: Indiana University Press, 1968); introduction by John Reeves

"A Concealed Printing in W. D. Howells," *Papers of the Bibliographic Society of America*, 61 (1967), 56-60

editor, *Non Solus*, A Publication of the University of Illinois Library Friends, 1974-1981

editor, Robert B. Downs Publication Fund, University of Illinois Library, 1975-1981

reviews, short articles, etc. in *Victorian Studies*, *Journal of English and German Philology*, *Victorian Periodicals Newsletter*, *Collection Management*, *Nineteenth-Century Literature*, *College & Research Libraries*, *Scholarly Publishing Today*, *ARL Newsletter*, *Serials Review*, *Library Issues*, *S[ociety for] S[cholarly] P[ublishing] Newsletter*, and *Victorian Britain: An Encyclopedia*

Software, practice & experience.

1971-
English Serial Publication : Periodical : Monthly Internet Resource volumes ; 26 cm
Chichester : Wiley Interscience.

GET THIS ITEM

Access: http://www3.interscience.wiley.com/cgi-bin/jhome/1752

Availability: **FirstSearch indicates your institution subscribes to this publication.**
- Libraries worldwide that own item: 598 UNIV OF ILLINOIS
- Search the catalog at the Library of University of Illinois at Urbana-Champaign

External Resources:
- Discover full text Discover UIUC Full Text
- Interlibrary Loan Request
- Cite This Item

FIND RELATED

More Like This: Advanced options...

Browse Journal: Available Issues (ArticleFirst)

Title: **Software, practice & experience.**

Publication: Chichester : Wiley Interscience,

Year: 1971-

Frequency: Monthly, 2010-; **Past:** Quarterly, 1971-1976 Monthly, 1977-1997 15 issues a year, 1998-1999 Monthly, plus three additional issues in Apr., July, and Nov., 2000-2008 18 issues a year, 2009

Description: volumes ; 26 cm Vol. 1, no. 1 (Jan.-Mar. 1971)-

Language: English

Standard No: **ISSN:** 0038-0644; **Other format's ISSN:** 1097-024X; **CODEN:** SPEXBL; **National Library:** 010675248; 120252-2; S30400000; 009879556; 013345367; **LCCN:** 75-648615

Access: http://www3.interscience.wiley.com/cgi-bin/jhome/1752

SUBJECT(S)

Descriptor:
Computer software -- Periodicals.
Computer programming -- Periodicals.
Computer programs -- Periodicals.
Programmation (informatique) -- Périodiques.
Langages de programmation -- Périodiques.
Logiciels -- Périodiques.
Software.
Programmierung.
Zeitschrift.
Software.
Computer programming.
Computer programs.
Computer software.
Programmierung.
Software.
Zeitschrift.

Genre/Form: Periodicals.

Note(s): Title from cover./ Published: John Wiley & Sons, Ltd., <2005->

General Info: Some issues for 1995- accompanied by compact discs./ Vols. 1-20, 1971-90. 1 v./ Cumulated by CD-ROM ed. **National bibliography no:** 010675248 **Other format available:** Software (Chichester, England : CD-ROM); Online version:; Software, practice & experience (Online)

Class Descriptors: **LC:** QA76.5; **Dewey:** 001.6/425/05

Other Titles: Softw. pract. exp.; Software, practice & experience (Print); Also known as:; SP & E; Software, practice and experience; Software: Softw. pract. exper.; <2005->

Material Type: Periodical (per); Internet resource (url)

Document Type: Serial; Internet Resource

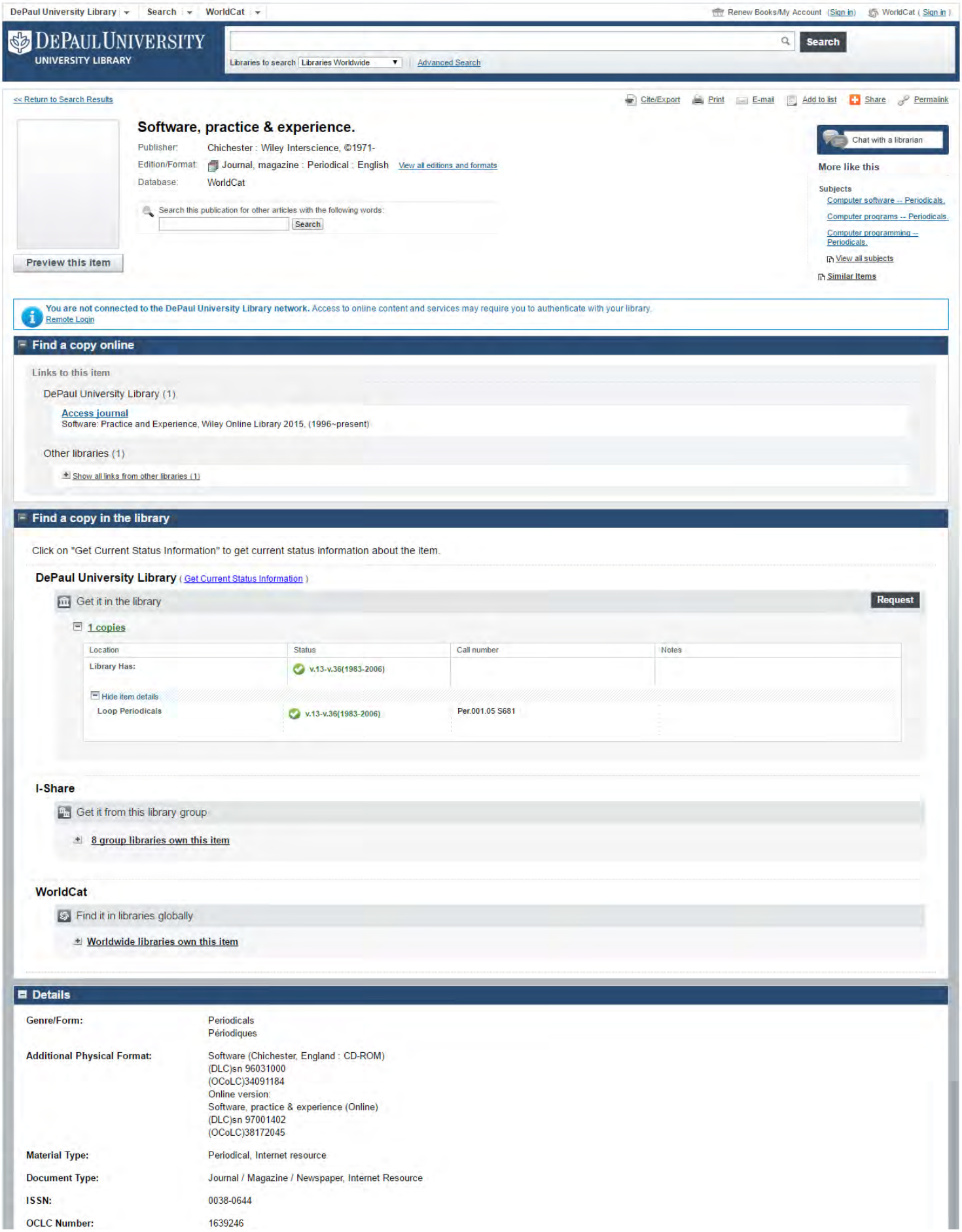
Date of Entry: 19750920

Update: 20150818

Accession No: OCLC: 1639246

Database: WorldCat

OCLC © 1992-2016 OCLC Terms & Conditions


DePaul University Library
Search
WorldCat
Renew Books/My Account (Sign in)
WorldCat (Sign in)
DEPAUL UNIVERSITY
UNIVERSITY LIBRARY
Search
Libraries to search Libraries Worldwide
Advanced Search
<< Return to Search Results
Cite/Export
Print
E-mail
Add to list
Share
Permalink
Software, practice & experience.
Publisher: Chichester : Wiley Interscience, ©1971-
Edition/Format: Journal, magazine : Periodical : English
View all editions and formats
Database: WorldCat
Search this publication for other articles with the following words:
Search
Preview this item
Chat with a librarian
More like this
Subjects
Computer software -- Periodicals.
Computer programs -- Periodicals.
Computer programming -- Periodicals.
View all subjects
Similar Items
You are not connected to the DePaul University Library network. Access to online content and services may require you to authenticate with your library.
Remote Login
Find a copy online
Links to this item
DePaul University Library (1)
Access journal
Software: Practice and Experience, Wiley Online Library 2015, (1996~present)
Other libraries (1)
Show all links from other libraries (1)
Find a copy in the library
Click on "Get Current Status Information" to get current status information about the item.
DePaul University Library (Get Current Status Information)
Get it in the library
Request
1 copies
Location
Status
Call number
Notes
Library Has:
v.13-v.36(1983-2006)
Hide item details
Loop Periodicals
v.13-v.36(1983-2006)
Per.001.05 S681
I-Share
Get it from this library group
8 group libraries own this item
WorldCat
Find it in libraries globally
Worldwide libraries own this item
Details
Genre/Form:
Periodicals
Périodiques
Additional Physical Format:
Software (Chichester, England : CD-ROM)
(DLC)sn 96031000
(OCoLC)34091184
Online version:
Software, practice & experience (Online)
(DLC)sn 97001402
(OCoLC)38172045
Material Type:
Periodical, Internet resource
Document Type:
Journal / Magazine / Newspaper, Internet Resource
ISSN:
0038-0644
OCLC Number:
1639246

Notes:	Title from cover. Published: John Wiley & Sons, Ltd., <2005->
Description:	volumes ; 26 cm
Other Titles:	Software, practice & experience SP & E Software, practice and experience Software Softw. pract. exper.
Local System Bib Number:	353288

Abstract:

Reviews

User-contributed reviews

Tags

Add tags for "Software, practice & experience.". Be the first.

Similar Items

Related Subjects: (12)

- Computer software -- Periodicals.
- Computer programs -- Periodicals.
- Computer programming -- Periodicals.
- Programmation (Informatique) -- Périodiques.
- Langages de programmation -- Périodiques.
- Logiciels -- Périodiques.
- Software.
- Programmierung.
- Zeitschrift.
- Computer programming.
- Computer programs.
- Computer software.

Languages: Česky | Deutsch | English | Español | Français | Italiano | Nederlands | Português | עברית | 한국어 | 日本語 | 中文(繁體) | 中文(简体)
You: Sign In | Register | My WorldCat | My Lists | My Watchlist | My Reviews | My Tags | My Saved Searches
WorldCat: Home | About | Help | Search
Legal: Copyright © 2001-2016 OCLC. All rights reserved. | Privacy Policy | Terms and Conditions

WorldCat is the world's largest library catalog, helping you find library materials online.

COPYING OF ARTICLES
The code and the copyright notice appearing at the bottom of the first page of an article in this journal indicates the copyright owner's consent that copies of the article may be made for personal or internal use, or for the personal or internal use of specific clients, on the condition that the copier pay for copying beyond that permitted by Sections 107 or 108 of the U.S. Copyright Law. The per-copy fee is to be paid through the Copyright Clearance Center Inc. and details of their authorisation schemes can be obtained from them at 27 Congress Street, Salem, MA 01970, U.S.A. Tel: (508) 744-3350, Fax: (508) 745-9379. This consent does not extend to other kinds of copying, such as copying for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. Such permission requests, or other inquiries, should be addressed to the publisher.

0038-0644(199509)25:9;1-D

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 10

OCTOBER 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

Software: practice & experience
25:10
DePaul University Library
Received on: 10-25-95
Per. 001.05 S681
Loop Campus

Toronto · Singapore
ion

SPEXBL
ISSN 0038-06

SOFTWARE—PRACTICE AND EXPERIENCE
(Softw. pract. exp.)

CONTENTS

VOLUME 25, ISSUE No. 10 *October 1995*

Migration in Object-oriented Database Systems—A Practical Approach: C. Huemer, G. Kappel and S. Vieweg 1065

Automatic Synthesis of Compression Techniques for Heterogeneous Files: W. H. Hsu and A. E. Zwarico 1097

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System: P. Ashton and J. Penny 1117

Process Scheduling and UNIX Semaphores: N. Dunstan and I. Fris 1141

Software Maintenance: An Approach to Impact Analysis of Objects Change: S. Ajila 1155

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

Indexed or abstracted by Cambridge Scientific Abstracts, CompuMath Citation Index (ISI), Compuscience Database, Computer Contents, Computer Literature Index, Computing Reviews, Current Contents/Eng, Tech & Applied Sciences, Data Processing Digest, Deadline Newsletter, Educational Technology Abstracts, Engineering Index, Engineering Societies Library, IBZ (International Bibliography of Periodical Literature), Information Science Abstracts (Plenum), INSPEC, Knowledge Engineering Review, Nat Centre for Software Technology, Research Alert (ISI) and SCISEARCH Database (ISI).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1097–1116 (OCTOBER 1995)

Automatic Synthesis of Compression Techniques for Heterogeneous Files

WILLIAM H. HSU
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
(email: bhsu@cs.uiuc.edu, voice: (217) 244-1620)

AND

AMY E. ZWARICO
Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.
(email: amy@cs.jhu.edu, voice: (410) 516-5304)

SUMMARY

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix `compress`, *StuffIt*, and *Compact Pro*, and show that our system provides better space savings.

KEY WORDS: adaptive/selective data compression algorithms; redundancy metrics; heterogeneous files; program synthesis

INTRODUCTION

The primary motivation in studying compression is the savings in space that it provides. Many compression algorithms have been implemented, and with the advent of new hardware standards, more techniques are under development. Historically, research in data compression has been devoted to the development of algorithms that exploit various types of redundancy found in a file. The shortcoming of such algorithms is that they assume, often inaccurately, that files are homogeneous throughout. Consequently, each exploits only a subset of the redundancy found in the file.

Unfortunately, no algorithm is effective in compressing all files.[1] For example, dynamic Huffman coding works best on data files with a high variance in the frequency of individual characters (including some graphics and audio data), achieves mediocre performance on natural language text files, and performs poorly in general on high-redundancy binary data. On the other hand, run length encoding works well on high-redundancy binary data, but performs very poorly on text files. Textual substitution works best when multiple-character strings tend to be repeated, as in English text, but this performance degrades as the average

CCC 0038–0644/95/101097–20
©1995 by John Wiley & Sons, Ltd.
Received 20 April 1994
Revised 5 February 1995

length of these strings decreases. These relative strengths and weaknesses become criti when attempting to compress *heterogeneous* files. Heterogeneous files are those which co tain multiple types of data such as text, images, binary, audio, or animation. Consequent their constituent parts may have different degrees of compressibility. Because most co pression algorithms are either tailored to a few specific classes of data or are designed handle a single type of data at a time, they are not suited to the compression of heterog neous files. In attempting to apply a single method to such files, they forfeit the possibili of greater savings achievable by compressing various segments of the file with differe methods.

To overcome this inherent weakness found in compression algorithms, we have develop a *heterogeneous compressor* that automatically chooses the best compression algorithm use on a given variable-length block of a file, based on both the qualitative and quantit tive properties of that segment. The compressor determines and then applies the selecte algorithms to the blocks separately. Assembling compression procedures to create a speci ically tailored program for each file gives improved performance over using one progra for all files. This system produces better compression results than four commonly availabl compression packages, PKZIP,[2] Unix `compress`,[3] *StuffIt*,[4] and *Compact Pro*[5] for arbitrar heterogeneous files.

The major contributions of this work are twofold. The first is an improved compressio system for heterogeneous files. The second is the development of a method of statisti cal analysis of the compressibility of a file (its redundancy types). Although the concep of redundancy types is not new,[6,7] synthesis of compression techniques using redundanc measurements is largely unprecedented. The approach presented in this paper uses a straight forward program synthesis technique: a *compression plan*, consisting of instructions for eac block of input data, is generated, guided by the statistical properties of the input data. Be cause of its use of algorithms specifically suited to the types of redundancy exhibited by the particular input file, the system achieves consistent average performance throughout the file, as shown by experimental evidence.

As an example of the type of savings our system produces, consider compressing a heterogeneous file (such as a small multimedia data file) consisting of 10K of low redun- dancy (non-natural language) ASCII data, 10K of English text, and 25K of graphics. In this case, a reasonably sophisticated compression program might recognize the increased savings achievable by employing Huffman compression, to better take advantage of the fact that the majority of the data is graphical. However, none of the general-purpose compres- sion methods under consideration are optimal when used alone on this file. This is because the text part of this file is best compressed by textual substitution methods (e.g., Lempel–Ziv) rather than statistical methods, while the low-redundancy data* and graphics parts are best compressed by alphabetic distribution-based methods (e.g., arithmetic or dynamic Huffman coding) rather than Lempel–Ziv or run-length encoding. This particular file totals 45K in length before compression. A compressor using pure dynamic Huffman coding only achieves about 7 per cent savings for a compressed file of length 42.2K. One of the best general-purpose Lempel–Ziv compressors currently available[8,9] achieves 18 per cent sav- ings, producing a compressed file of length 37.4K. Our system uses arithmetic coding on the first and last segments and Lempel–Ziv compression on the text segment in the middle, achieving a 22 per cent savings and producing a compressed file of length 35.6K. This is a 4 per cent improvement over the best commercial system.

* This denotes, in our system, a file with a low rate of repeated strings.

NetApp; Rackspace Exhibit 1014 Page 30

The purpose of our experiments was to verify the conjecture that a selective system for combining methods can improve savings on a significant range of heterogeneous files, especially multimedia data. This combination differs from current adaptive methods in that it switches among compression paradigms designed to remove very different types of redundancy. By contrast, existing adaptive compression programs are sensitive only to changes in particular types of redundancy, such as run-length, which do not require changing the underlying algorithm used in compression. Thus they cannot adapt to changes in the type of redundancy present, such as from high run-length to high character repetition. The superiority of our approach is demonstrated in our experimental section.

This paper begins with a presentation of existing approaches to data compression, including a discussion of hybrid and adaptive compression algorithms and a description of four popular commercial compression packages. These are followed by documentation on the design of the heterogeneous compression system, analysis of experimental results obtained from test runs of the completed system, and comparison of the system's performance against that of commercial systems. Finally, implications of the results and possibilities for future work are presented.

RELATED WORK

It is a fundamental result of information theory that there is no single algorithm that performs optimally in compressing all files.[1] However, much work has been done to develop algorithms and techniques that work nearly optimally on all classes of files. In this section we discuss adaptive algorithms, composite algorithms, and four popular commercial compression packages.

Adaptive compression algorithms and composite techniques

Exploiting the heterogeneity in a file has been addressed in two ways: the development of *adaptive* compression algorithms, and the composition of various algorithms. Adaptive compression algorithms attune themselves gradually to changes in the redundancies within a file by modifying parameters used by the algorithm, such as the dictionary, during execution.

For example, adaptive alphabetic distribution-based algorithms such as dynamic Huffman coding[10] maintain a tree structure to minimize the encoded length of the most frequently occurring characters. This property can be made to change continuously as a file is processed.

An example of an adaptive textual substitution algorithm is Lempel–Ziv compression, a title which refers to two distinct variants of a basic textual substitution scheme. The first variant, known as LZ77 or the *sliding dictionary* or *sliding window* method, selects positional references from a constant range of preceding strings.[1, 11] These 'back-pointers' literally encode position and length of a repeated string. The second variant, known as LZ78 or the *dynamic dictionary* method, uses a dictionary structure with a *paging* heuristic. When the dictionary – a table of strings from the file – is completely filled, the contents are not discarded. Instead, an auxiliary dictionary is created and updated while compression continues using the main dictionary. Each time this auxiliary table is filled, its contents are 'swapped' into the main dictionary and it is cleared. The maintenance of dictionaries for textual substitution is analogous to the semi-space method of garbage collection, in which two pages are used but only one is 'active' – these are exchanged when one fills beyond a preset threshold. Another adaptive variant of this algorithm is the Lempel–Ziv–Welch

NetApp; Rackspace Exhibit 1014 Page 31

(LZW) algorithm, a descendant of LZ78 used in Unix `compress`.[6, 12] Both LZW and LZ78 vary the length of strings used in compression.[6, 12]

Yet another adaptive (alphabetic distribution-based) compression scheme, the Move-To-Front (MTF) method, was developed by Bentley *et al.*[13] In MTF, the 'word code' for a symbol is determined by the position of the word in a sequential list. The word list is ordered so that frequently accessed words are near the front, thus shortening their encodings.

Adaptive compression algorithms are not appropriate to use with heterogeneous files because they are sensitive only to changes in the particular redundancy type with which they are associated, such as a change in the alphabetic distribution. They do not exploit changes across different redundancy types in the files. Therefore a so-called adaptive method typically cannot directly handle drastic changes in file properties, such as an abrupt transition from text to graphics. For example, adaptive Huffman compressors specially optimized for text achieve disproportionately poor performance on certain image files, and vice versa. As the use of multimedia files increases, files exhibiting this sort of transition will become more prevalent.

Our approach differs from adaptive compression because the system chooses each algorithm (as well as the duration of its applicability) before compression begins, rather than modifying the technique for each file during compression. In addition, while adaptive methods make modifications to their compression parameters on the basis of single bytes or fixed length strings of input, our heterogeneous compressor bases its compression upon statistics gathered from larger blocks of five kilobytes. This allows us to handle much larger changes in file redundancy types. This makes our system less sensitive to residual statistical fluctuations from different parts of a file. We note that it is possible to use an adaptive algorithm as a primitive in the system.

Another approach to handling heterogeneous files is the composition of compression algorithms. Composition can either be accomplished by running several algorithms in succession or by combining the basic algorithms and heuristics to create a new technique. For example, recent implementations of 'universal' compression programs execute the Lempel–Ziv algorithm and dynamic Huffman coding in succession, thus improving performance by combining the string repetition-based compression of Lempel–Ziv with the frequency-based compression strategy of dynamic Huffman coding. One commercial implementation is *LHarc*.[14,15] Our system exploits the same savings since it uses the *Freeze* implementation of the Lempel–Ziv algorithm, which filters Lempel–Ziv compressed output through a Huffman coder. An example of a truly composite technique is the compression achieved by using Shannon–Fano tries* in conjunction with the Fiala–Greene algorithm (a variant of Lempel–Ziv)[16] in the PKZIP[2] commercial package. Tries are used to optimally encode strings by character frequency.[17] PKZIP was selected as the representative test program from this group in our experiment due to its superior performance on industrial benchmarks.[9]

Our approach generalizes the ideas of successively executing or combining different compression algorithms by allowing any combination of basic algorithms within a file. This includes switching from among algorithms an arbitrary number of times within a file. The algorithms themselves may be simple or composite and may be adaptive. All are treated as atomic commands to be applied to portions of a file.

* A *trie* is a tree of variable degree ≥ 2 such that (1) each edge is labelled with a character, and the depth of any node represents one more than the number of characters required to identify it; (2) all internal nodes are intermediate and represent prefixes of keys in the trie; (3) keys (strings) may be inserted as leaves using the minimum number of characters which distinguish them uniquely. Thus a generic trie containing the strings *computer* and *compare* would have keys at a depth of five which share a common prefix of length four.

NetApp; Rackspace Exhibit 1014 Page 32

The problem of heterogeneous files was addressed by Toal[18] in a proposal for a naive heterogeneous compression system similar to ours. In such a system, files would be segmented into fixed-length encapsulated blocks; the optimal algorithm would be selected for each block on the basis of their simple taxonomy (qualitative data type) only; and the blocks would be *independently* compressed. Our system, however, performs more in-depth statistical analysis in order to make a more informed selection from the database of algorithms. This entails not only the determination of qualitative data properties but the computation of metrics for an entire block (as opposed to sporadic or random sampling from parts of each block). Furthermore, normalization constants for selection parameters (i.e. the redundancy metrics) are fitted to observed parameters for a test library. Finally, a straightforward but crucial improvement to the naive encapsulated-block method is the implementation of a multi-pass scheme. By determining the complete taxonomy (data type and dominant redundancy type) in advance, any number of contiguous blocks which use the same compression method will be treated as a single segment. Toal observed in preliminary experiments that the overhead of changing compression schemes from one block to another dominated the additional savings that resulted from selection of a superior compression method.[18] This overhead is attributable to the fact that blocks compressed independently (even if the same method is used) are essentially separate files and assume the same startup overhead of the compression algorithm used.* We have determined experimentally that merging contiguous blocks whenever possible obviates the large majority of changes in compression method. This eliminates a sufficient proportion of the overhead to make heterogeneous compression worthwhile.

Commercial products

One of the goals of this research was to develop a compression system which is generally superior to commercially available systems. The four systems we studied are PKZIP, developed for microcomputers running MS-DOS;[2] Unix `compress`;[3] and *StuffIt Classic*[4] and *Compact Pro*,[5] developed for the Apple Macintosh operating system. Each of these products performs its compression in a single pass, with only one method selected per file. Thus, the possibility of heterogeneous files is ignored.

Unix `compress` uses an adaptive version of the Lempel–Ziv algorithm.[6] It operates by substituting a fixed-length code for common substrings. `compress`, like other adaptive textual substitution algorithms, periodically tests its own performance and reinitializes its string table if the amount of compression has decreased.

StuffIt makes use of two sets of algorithms: it first detects special-type files such as image files and processes them with algorithms suited for high-resolution color data; for the remaining files, it queries the operating system for the explicit file type given when the file was created, and uses this information to choose either the LZW variant of Lempel–Ziv,[4, 6] dynamic Huffman coding, or run-length encoding. This is a much more limited selection process than what we have implemented. Additionally, no selection of compression methods is attempted within a file. *Compact Pro* uses the same methodology as *StuffIt* and `compress`, but incorporates an improved Lempel–Ziv derived directly from LZ77.[11] The public-domain version of *StuffIt* is derived from Unix `compress`, as is evident from the similarity of their performance results.

* For purposes of comparison, the block sizes tested by Toal were nearly identical to those used in our system (ranging upwards from 4K).

Compression systems such as *StuffIt* perform simple selection among alternative compression algorithms. The important problem is that they are underequipped for the task of fitting a specific technique to each file (even when the uncompressed data is homogeneous). *StuffIt* uses few heuristics, since its algorithms are intended to be 'multipurpose' . Furthermore, only the file type is considered in selecting the algorithm – that is, no measures of redundancy are computed. Earlier versions of *StuffIt* (which were extremely similar to Unix `compress`) used composite alphabetic and textual compression, but made no selections on the basis of data characteristics. The chief improvements of our heterogeneous compressor over this approach are that it uses a two-dimensional lookup table, indexed by file properties and quantitative redundancy metrics, and – more important – that it treats the file as a collection of heterogeneous data sets.

THE HETEROGENEOUS COMPRESSOR

Our heterogeneous compressor treats a file as a collection of fixed size blocks (5K in the current implementation), each containing a potentially different type of data and thus best compressed using different algorithms. The actual compression is accomplished in two phases. In the first phase, the system determines the type and compressibility of each block. The compressibility of each block of data is determined by the values of three quantitative metrics representing the alphabetic distribution, the average run length and the string repetition ratio in the file. If these metrics are all below a certain threshold, then the block is considered fully compressed (uncompressible) and the program continues on to the next block. Otherwise, using the block type and largest metric, the appropriate compression algorithm (and possible heuristic) are chosen from the compression algorithm database. The compression method for the current block is then recorded in a small array-based map of the file, and the system continues.

The second phase comprises the actual compression and an optimization that maximizes the size of a segment of data to be compressed using a particular algorithm. In this optimization, which is interleaved with the actual compression, adjacent blocks for which exactly the same method have been chosen are merged into a single block. This merge technique maximizes the length of segments requiring a single compression method by greedily scanning ahead until a change of method is detected. Scanning is performed using the array map of the file generated when compression methods were selected from the database. A compression history, needed for decompression, is automatically generated as part of this phase.

The newly compressed segments are written to a buffer by the algorithm, which stores the output data with the compression history. The system then writes out the compressed file and exits with a signal to the operating system that compression was successful.

From this two-pass scheme it is straightforward to see why this system is less susceptible than traditional adaptive systems to biases accrued when the data type changes abruptly during compression. Adaptive compressors perform all operations myopically, sacrificing the ability to see ahead in the file or data stream to detect future fluctuations in the type of data. As a result, adaptive compressors retain the statistical vestiges of the old method until these are 'flushed out' by new data (or balanced out, depending upon the process for paging and aging internal data structures such as dictionaries). Thus adaptive compressors may continue to suffer the effects of bias, achieving suboptimal compression. On the other hand, by abruptly changing compression algorithms, our technique completely discards all remnants of the 'previous' method (i.e. the algorithm used on the preceding segment). This

NetApp; Rackspace Exhibit 1014 Page 34

allows us to immediately capitalize on changes in data. In addition, merging contiguous blocks of the same data type acquires the advantage of incurring all the overhead *at once* for switching to what will be the best compression method for an entire variable-length segment. The primary advantage of adaptive compression techniques over our technique is that the adaptive compression algorithms are 'online' (single-pass). This property increases compression speed and, more important, gives the ability to compress a data stream (for instance, incoming data packets in a network or modem transmission) in addition to files in secondary storage or variable-length buffers.

The remainder of this section presents the system. We begin with a description of the calculation of the block types and the redundancy metrics. We also explain the use of the metrics as absolute indicators of compressibility, and then describe the compression algorithms used and the structure of the database of algorithms. A discussion of implementation details concludes the section.

Property analysis

The compressibility of a block of data and the appropriate algorithm to do so are determined by the type of data contained in a block and the type of redundancy (if any) in the data. These two properties are represented by four parameters: the *block type*, and the three *redundancy metrics*. The block type describes the data in the block – text, binary, graphical, etc. The three redundancy metrics are the degree of variation in character frequency, average run length in the file, and the string repetition ratio of the file. They provide a quantitative measure of how compressible the block is and which type of redundancy is most evident in the block. The use of both quantitative redundancy measures (redundancy metrics) and qualitative characteristics (block types) as indicators for data compressibility is advocated by Held[7] and Salton.[19] We have refined the process for computing those attributes referred to as *datanalysis results* by Held[7] and as *statistical language characteristics* by Salton[19] to obtain an actual guide for compression. The remainder of this section describes how these four parameters are determined for each block.

Block types

The *block type* describes the nature of a segment of input data. There are ten classifications of data in this system: ANSI text, non-natural language text (hexadecimal encodings of binary data), natural language text, computer source code, low redundancy binary, digitized audio, low resolution graphics, high-resolution graphics, high-redundancy binary executable, and binary object data. ANSI text is composed of characters from a superset of the ASCII alphabet. Non-natural language text contains primarily ASCII text but does not follow a distribution of characters like that of human languages. Examples are computer typesetting data, *uuencoded* and *BinHex* encoded data (which has the same character distribution as binary data but is converted to text for ease of transmission). Natural language text includes text written in English as well as other languages which are representable by the Roman (ASCII) alphabet. Most European languages (including the ones using the Cyrillic alphabet), special symbols excluded, fall into this category, as do the Pinyin and Katakana romanizations of the Chinese and Japanese languages (as opposed to their digital encodings). Computer source code uses the ASCII alphabet but characters are distributed with a different frequency than in natural language text. Low-redundancy binaries usually contain compressed data, but may also include data which is merely difficult to compress. Audio

data are very high in redundancy; audio files (and audio segments of multimedia files) are usually extremely large. Low-resolution graphics have long runs of contiguous repeated bits but unlike high-resolution graphics are not suited to lossy compression. High-resolution graphics include color and grayscale and may be compressed with lossy methods. Binary executables, like low-resolution graphics, have long runs of contiguous repeated bits and comprise all compiled programs on a computer system. Finally, object data has slightly shorter runs but is similarly redundant.

To determine the block type we use a procedure `new-file` which is our extension of the Unix `file` command.[20] `file` works by examining the first 512 bytes of a file and comparing the pattern of data contained in it to a collection of known data patterns from Unix and other operating systems. `new-file` works in a similar fashion, with two modifications. First, it examines and compares not only the first 512 bytes of a data set, but also 512 bytes in the middle of the set and the 512 bytes at the end (if they exist). This provides a better indication of the *primary* data type of a file by taking into account the possibility that the properties may change anywhere within the file. Thus, `new-file` decides on the 'most applicable' data type by a majority vote (or the first data type detected in the case of a three-way tie). The other change is that the known patterns of data have been increased by adding three graphics patterns.

Redundancy metrics

The redundancy metrics are quantitative measures that are used to determine the compressibility of a block of data. They are: the *degree of variation in character frequency* or *alphabetic distribution*, M_{AD}; the *average run length* of the block, M_{RL}; and the *string repetition ratio* of the block, M_{SR}. In general, these three manifestations of redundancy are independent. Each of the redundancy types is exploited by different compression algorithms. *Frequency of characters* is exploited by arithmetic or alphabetic encoding algorithms. In arithmetic coding data is represented by an interval that is calculated from the probability distribution of data. With alphabetic coding algorithms such as the Huffman[21] and Shannon–Fano[22] algorithms, more frequently occurring characters are replaced by shorter units than the less frequently occurring characters. *Contiguous strings*, long strings of identical units occurring next to one another, are exploited by run length encoding algorithms.[23] In these algorithms, contiguous strings are replaced by a single occurrence of the string, called a *run*, plus a count of the number of identical strings following. Both alphabetic distribution and average run length are sometimes characterized as statistical redundancy metrics.[24] *Recurrent strings*, which occur repeatedly in the input stream with any number of interleaved symbols, are exploited by *textual substitution algorithms* such as Lempel–Ziv.[6,11,12] In these algorithms, recurrent instances are replaced with positional references (pointers) to the original instance.

Experimental evidence for the efficacy of quantitative redundancy measures is described in texts by Storer[1] and Shannon.[22] Shannon provided an estimate of the entropy of English text, approximately bounding it to be between one and two bits per character.[22] This was determined experimentally by presenting fragments of (unfamiliar) English text to human subjects and recording the frequency with which they guessed unknown letters. The fragments were revealed character by character, so that letters at the end of long or uncommon words were easiest to guess and letters at the beginnings of words were hardest. The observation that binary executables are known to possess high average run lengths is found in Storer.[1] However, this property is rarely exploited or measured.

NetApp; Rackspace Exhibit 1014 Page 36

Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function G to be a number between 0 and 10 so as to simplify comparison among the different metrics. The gamma distribution was chosen because the graph of each of the unscaled redundancy metrics for a test set of 50 files, when plotted on a histogram, approximated a gamma distribution. Normal and χ^2 distributions were also considered, but these proved to be too specific for the application (since they are both specific parametric cases of the gamma distribution). The gamma distribution is defined as follows (cf Ross[25]):

$$\begin{aligned} G_\tau(x_\tau) &= \int_0^{x_\tau} f_\tau(x)\,\mathrm{d}x \\ f_\tau(x) &= \frac{\lambda_\tau \mathrm{e}^{-\lambda_\tau x}(\lambda_\tau x)^{t_\tau - 1}}{\Gamma(t_\tau)} \\ \Gamma(t_\tau) &= \int_0^{\infty} \mathrm{e}^{-y} y^{t_\tau - 1}\,\mathrm{d}y \end{aligned}$$

where f_τ is the density function, Γ is the gamma function, x_τ is the unnormalized measure, t_τ is the shape parameter for the gamma distribution, and λ_τ is the scale parameter for the gamma distribution. The τ subscript simply represents the redundancy type under consideration, i.e. AD, RL, or SR, respectively. The shape and scaling parameters, t_τ and λ_τ respectively, were determined by fitting the best gamma distribution curve to the data set. This was done by performing the preferred compression method for each file and tabulating the induced ratio among normalized metrics to yield the desired parameter values for each segment. These were then averaged to obtain the empirical scaling parameters.

The *alphabetic distribution metric* (the degree of variation in character frequency) of a block is calculated by taking the population (root-mean-square) standard deviation of the ordinal values of characters in the block and dividing it by the block length (in bytes). The M_{AD} metric is calculated by the following formulas:

$$\begin{aligned} M_{\mathrm{AD}} &= 10 * G_{\mathrm{AD}}(x_{\mathrm{AD}}) \\ x_{\mathrm{AD}} &= \frac{\alpha}{\text{block length in bytes}} \\ \alpha &= \sqrt{\frac{\sum_{c \in \text{charset}} (c-\mu)^2}{256}}, \end{aligned}$$

where c is the ordinal value of a character and μ is the average ordinal value of all characters in a block. The normalization uses $t_{\mathrm{AD}} = 1.70$ and $\lambda_{\mathrm{AD}} = 53.0$ as parameters.

The *average run length metric* is obtained by dividing the number of bits in a block by the number of runs. A run is defined to be a repetition of symbols (either bits or bytes). Our implementation takes both bitwise and bytewise run lengths. For example, if $f = 0001111001110000$ is a file of 16 bits, then the number of bit runs is 5, and the number of byte runs is 2. The scaled metric M_{RL} is obtained by:

$$\begin{aligned} M_{\mathrm{RL}} &= 10 * G_{\mathrm{RL}}(x_{\mathrm{RL}}) \\ x_{\mathrm{RL}} &= \frac{\text{file length in bits}}{\text{number of runs}} \end{aligned}$$

with gamma distribution parameters $t_{RL} = 0.50$ and $\lambda_{RL} = 12.0$.

The *string repetition ratio metric* is the total number of n-bit strings in the block divided by the number of distinct n-bit strings (up to 100K). In our implementation, n is 32, the word size of our machine. The normalized metric M_{SR} is obtained by:

$$M_{SR} = 10 * G_{SR}(x_{SR})$$
$$x_{SR} = \frac{\text{number of } n \text{ bit strings}}{\text{number of distinct } n \text{ bit strings}}$$

with gamma distribution parameters $t_{SR} = 0.18$ and $\lambda_{SR} = 0.2$.

The alphabetic distribution and average run length metrics can be calculated in linear time. The string repetition ratio can be computed in $O(n \log n)$ time using a dictionary data structure. For simplicity, and because a (small) constant amount of data is scanned, we use an $O(n^2)$ version. New strings are stored in an array rather than a binary tree, which would require more insertion overhead (and is not worth while for the 5K block length used in the current system). Our routine integrates $f_\tau(x)$ by Simpson's Rule with $n = 10$ intervals.

The largest of the three metrics is assumed to represent the most significant type of redundancy present in the block. It is expected that compression will decrease at least one of the metrics, and experiments conducted on a wide variety of files have proven this convention to be reliable. Experiments have also shown that if all the normalized metrics are smaller than 2.5, the file is considered not compressible, and the system records a verdict of 'uncompressible' on the current block. If at least one of the parameters is greater than 2.5, the file is considered compressible. The maximum of the normalized metrics is then selected and used in conjunction with the file type to select the appropriate compression algorithm from the lookup table described in the following section. A negative compressibility test does not always imply that all three metrics are below the threshold. In some cases, the only redundancy type for which a metric is above the threshold accesses a null entry in the database of compression algorithms. This is interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm.

The algorithm and heuristic database

The compression algorithms and attendant heuristics are organized into the 10 by 3 table illustrated by Table I. The 10 file descriptors are the row indices and the 3 metrics are the column indices. Each entry of the table contains descriptors which are used to access the code for an algorithm-heuristic pair. It should be noted that four of the entries are blank (indicated by an *). A blank entry indicates that the combination of block type and highest metric are very unusual. In this case, the next highest metric is used instead, provided that it is above the threshold. As an example of using this table, consider a high-redundancy binary executable file whose highest metric is the string repetition metric M_{SR}. Together, this pair indicates that the Lempel–Ziv compression algorithm with the Freeze heuristic will be used.

The algorithms

There are four basic algorithms used by the system: arithmetic coding,[26] Lempel–Ziv,[8] run length encoding (RLE),[23] and JPEG for image/graphics compression.[27]

Arithmetic coding algorithms compress data by representing that data by an interval of

NetApp; Rackspace Exhibit 1014 Page 38

Table I. Database of compression algorithms†

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
hexadecimal	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
natural language	arithmetic coding	*	Lempel–Ziv
	*	*	freeze
source code	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
low redundancy	*	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	*
audio	*	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
low resolution	*	run-length encoding	Lempel–Ziv
graphic	*	n-bit run count	freeze
high resolution	JPEG	run-length encoding	JPEG
color graphic	improved Huffman	n-bit run count	improved Huffman
high redundancy	arithmetic coding	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	freeze
object	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze

† Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

real numbers between zero and one. The width of this interval is inversely proportional to the number of symbols encoded, and the decrease in width is directly proportional to the frequency of the original symbols. Thus the interval specifies the encoded message via its bounds, with the precision (distance) of these bounds reflecting the information content of the message. The end result is that arithmetic coding achieves, in practice, much better space savings than Huffman coding and its dynamic implementations because of its higher likelihood of actually achieving the theoretical lower bound.[24, 28] Although early arithmetic coding algorithms performed too slowly to be of practical use,[29] the implementation of the Witten–Neal–Cleary algorithm used here[26] is optimized for speed – at some cost in space savings, but without giving up its advantage over dynamic Huffman coding. The reader is referred to Bell *et al*[24] for a thorough overview of arithmetic coding. We should note that in earlier implementation of the heterogeneous compressor we used a dynamic Huffman algorithm instead of arithmetic coding. We changed our implementation when we found that then Witten–Neal–Cleary algorithm[26] outperformed our implementation of dynamic Huffman coding[10, 30] in both space savings and execution time.

Run length encoding (RLE) algorithms compress data by replacing contiguous occurrences of a single-unit symbol (either bit or byte) by an efficiently coded count of these runs, usually a single occurrence of the symbol and the number of occurrences. We have implemented a straightforward RLE algorithm for our database, based on the description in Sedgewick.[23] In addition, bitwise and bytewise encoding are available as heuristics and the parameters of bitwise RLE are based on the RL metric.

Files with a high degree of string repetition are compressed using the Lempel–Ziv compression algorithm. It compresses data by replacing frequently occurring strings (with min-

NetApp; Rackspace Exhibit 1014 Page 39

imal regard of how far apart they occur) with compact pointers to the position of the first occurrence. Our implementation is a straightforward array-based encoding with constant-length codes. The algorithm maintains a dictionary of recurring strings in order to do the compression. In our system, the Lempel–Ziv algorithm is augmented with the *Freeze* heuristic. This heuristic suppresses paging of strings in the dictionary after it has been filled; that is, it prevents the replacement of previously encountered strings, regardless of how long ago or how infrequently the string has been encountered. *Freeze* is primarily a speed optimization, since it requires less computation than paging heuristics such as least recently used (LRU) or least frequently used (LFU), but it has been shown to work well for all but the least string-redundant files (including both binary executables and most text files). For files with extremely low string-repetition, our system usually selects Huffman compression.

The compression of high-resolution graphics and audio files uses a *lossy* compression scheme. Appropriately used, lossy algorithms guarantee that the decompressed file is similar enough to the original as to be nearly indistinguishable by human perception, and that repeated compression and decompression leads to limited cumulative 'damage' . The primary benefit of lossy compression is that it guarantees much higher compression ratios at a minimal tradeoff. For instance, a very-high-resolution color image can be compressed with much higher savings (possibly 95 per cent) if the user allows a small amount of noise, always less than 1 per cent per compression, to be introduced during each compression. Our system uses the JPEG system[27] for compressing high-resolution color and grayscale images. JPEG, which is divided into lossy and lossless parts, typically achieves compression ratios of between 15-to-1 and 25-to-1. The *potential* for this substantial savings is obtained by the Discrete Cosine Transform portion of the algorithm, a lossy method. This determines a limit on the amount of savings that can then be achieved by any lossless compressor. The actual savings are realized by a lossless portion, known as the *back end* which is applied to the preprocessed image data. The implementation of this module used in our system[27] is a Huffman coder. It is independent of the lossy front end and can be replaced with a run-length or textual-substitution based algorithm, to be selected by the synthesis system. In our implementation, we chose to retain the original Huffman back end, a different algorithm from the general-purpose dynamic Huffman coder which we also studied.[10, 30] This is because the JPEG Huffman coder is especially suited to the redundancy remaining after lossy preprocessing. It is worthy of mention that the JPEG developers have investigated the use of arithmetic coding back ends, which were found to be experimentally superior but were not used because of proprietary considerations.[27]

Implementation

The system consists of a driver module, four block analysis modules, and the synthesis module, which includes the database of compression algorithms. All modules are written in C and were tested on a Unix platform. The program uses a data directed style of implementation for choosing the compression algorithm to apply to a block. Thus, additional block types, compression algorithms and heuristics, and redundancy metrics can be added to the system with minimal modification of the source code. Only the database would have to be updated and the block analysis routines extended; the rest of the program would remain the same.

The driver performs two iterative passes through the file. It first performs block analysis on the file one 5K block at a time. This block size was chosen after experimentation showed that the response of the system to changes in block type became roughly stable as block

NetApp; Rackspace Exhibit 1014 Page 40

size exceeded 5K (i.e., did not significantly increase as block size did), and that a block size of 5K yielded highly accurate metrics (in only 1 of the 20 test files did the heterogeneous compressor select a suboptimal algorithm for any block). Finally, we found that the highest level of adaptivity without a noticeable *decrease* in accuracy was achieved at 5K, hence our choice of 5K as the block size.

For each block, the system invokes the four analysis modules – three for metric computation and normalization and one to determine the file properties – and stores their output. It then performs the metric comparison and combines the results with the file property to complete the table lookup for the current block. An identifying tag for the selected algorithm is written to the 'compression plan' , an array which stores one complete compression instruction per block (if the current block is deemed uncompressible, a 'skip' instruction is recorded).

We pause here to discuss the normalization of the metrics. Originally, we used a naive normalization method: direct algebraic scaling with experimentally determined constants for each metric. This did not, however, accurately reflect the statistical relationship between variance in character frequency and alphabetic redundancy. Also, the behavior of these functions at asymptotes led to poor approximation of the overall distribution of data segments in the test files. The result was that arithmetic coding was too often incorrectly chosen, resulting in inferior compression; and selection approached randomness as metric values for both string repetition and alphabetic distribution tended toward extreme values. Using the gamma normalization method described above resulted in an improvement in the selection of arithmetic coding. Among the 20 benchmark files, arithmetic coding was selected as the compression method in exactly those cases where the other methods performed worse.

The second pass performs the compression of each block. In order to improve performance, this pass includes a simple optimization step which circumvents the overhead of restarting compression after each fixed length block by merging contiguous blocks that are to be compressed using the same compression algorithms.

On this same pass through the file, the system compresses each of the newly merged blocks using the algorithm recorded in the compression plan. The compressed data is written to an output buffer, while the compressed length (which indicates where in the compressed file a compressed block begins and ends) and compression method are recorded in a separate history for reference at decompression time. If negative compression or no compression is achieved, or if the block was already marked uncompressible, then the data is copied directly to the output buffer (the full block length and a code for 'no compression' are recorded in the compression history). Upon reaching the end of the blocks, the system writes out the compressed data from the output buffers and prepends the encoded compression history to produce the final output file.

When decompression is invoked, the driver module opens the compressed file, interprets the history tag and performs the necessary operations. The tags are a stored version of the compression history in compact, encoded form. Since the heterogeneous system generates different compression sequences for each file, and since the length of a compressed block varies with both the length of the original block and the compression method used, these tags are necessary to guide the decompression process. Currently only the compressed lengths of each block and the method of compression are stored, but a checksum for the original (decompressed) block length can be added with negligible overhead. When executed in reverse order on each compressed block, the instructions in the history tags result in the original file. For simplicity and security, they are prepended to the compressed file (and can easily be encrypted).

EXPERIMENTAL RESULTS

Design and construction of the test files

To test the overall performance, the system was run on a set of 20 test files. These files range in length from approximately 39K to 366K, with representative files from each of the ten block types included in the test corpus.

The test files are designed to model certain types of heterogeneous files, including utilities for image viewing, business, or audio processing, and hypothetical multimedia databases and programs. To construct these files, a collection of 30 files from the Unix, Apple Macintosh, and MS-DOS (IBM PC) operating systems was created. These files are listed in Table II. To create the test corpus, they were concatenated in groups of 2 or 3. The resultant series of test files is listed in Table III. All of the source files were used. The goal was to generate as broad a range of permutations as possible (while restricting the generated files to those which are likely to exist in a typical user environment). This was performed manually with consideration toward combinatorial constraints and the criteria of realistic data modeling. Since all of the files in the source collection originate from common commercial sources or from public archives (with the exception of the source and object files, which are from the code for the heterogeneous compressor itself), the latter constraint was considerably simplified.

The assembled files were then ported to the test sites (a Sun workstation for Unix `compress` and our heterogeneous compressor; a Macintosh for *StuffIt* and *Compact Pro*; and an IBM 80486 machine for PKZIP). Binary file transfer mode was used to ensure that the file lengths agreed exactly among all platforms.

Performance

In this section, we review and analyze the performance of the heterogeneous compressor with respect to compression savings, as compared with four of the commercial systems previously discussed; and execution time. Finally, we briefly note the implications of running the experiments and compiling performance data on several different architectures.

Compression savings

The total length of the uncompressed benchmark suite is just under three megabytes. Table IV shows the compressed length achieved by Unix `compress`, PKZIP, *StuffIt*, *Compact Pro* and the heterogeneous compression system. The heterogeneous compressor achieved the greatest compression, with a total compressed length of 1828K. This represents an additional savings of 162K (more than eight per cent) over the best commercial system (*Compact Pro v1.32*), and 339K (nearly 16 per cent) over the average. Compressed lengths for the commercial methods ranged from 1990K to 2375K.

Table V compares the percentage savings obtained by our system to the savings obtained by the commercial programs and the heterogeneous system. The last two columns show the difference in per cent saved between the synthesis system and the best and average of the four commercial packages. The best commercial compressor is marked for each of the files. Note that the heterogeneous compressor does better than all commercial programs in 19 of 20 cases and better than three of the four commercial systems in this one case (file 15). The difference in compression for this file is only 0.02 per cent, whereas for all the other files, the heterogeneous compressor has at least a 1.3 per cent improvement over the best

Table II. Files used to compose the test suite and their respective origins

File designation	File name	File type
audio1	cosby.snd	SoundMaster Macintosh audio file
lowrd1	ticker.txt	ASCII characters from stock ticker
lowrd2	exsound	compressed World Builder sound library
lowrd3	huff	compressed Unix executable
lowrd4	appnote.uue	uuencoded text
text1	phrack.txt	English text
text3	techbook.txt	Unix news article
text4	quanta1.txt	English text
text5	attilla.fluff	English text
text6	shadow.fluff	English text
text7	quanta2.txt	English text
execu1	ad	Unix executable
execu2	sh	Unix executable
execu3	blob	Silicon Graphics executable
execu4	zero	Silicon Graphics executable
execu5	network2.exe	IBM PC executable
execu6	hostname	Unix executable
graph1	compmisc.drw	Lotus Freelance line drawing
graph2	compperi.drw	Lotus Freelance line drawing
graph3	computer.drw	Lotus Freelance line drawing
graph4	lowres.mpt	MacPaint file
graph5	3dbar.drw	Lotus Freelance 3-D bar chart
graph6	image.ppm	PPM (high-resolution image) file
graph7	grp4	MacPaint file
objec1	test1.o	Unix object file
objec2	test2.o	Unix object file
objec3	test3.o	Unix object file
source1	table.c	C source code
source2	freeze.c	C source code

commercial compressor. The average of each column appears in the bottom row; note that the 'percent difference' averages are not weighted by file length, as each file is considered a separate experiment.

Because the quality of compression by the synthesis system depends on that of the algorithms and heuristics used, improvement of the implementations that we use should yield higher performance. This is evidenced by comparing the results of compressing a file dominated by string repetitions by Unix `compress` and *Compact Pro*. Both are implementations of the Lempel–Ziv algorithm. Unix `compress` has no heuristics, whereas *Compact Pro* is a better implementation of LZ77.[5, 11] *Compact Pro* consistently outperforms `compress`. It should be noted that the performance of the *Freeze* variant of Lempel–Ziv[8] used in our sys-

Table III. Combinations of the test files and the resultant simulated data types

File number	File composition	Classification of data modeled
1	text1 — lowrd1	news or stock report
2	graph7 — objec1	object file for a graphics viewer
3	lowrd1 — text3 — graph4	multimedia application (text/graphics)
4	graph7 — execu3	graphics viewer
5	audio1 — graph1	multimedia data file (sound/graphics)
6	text2 — lowrd1 — graph3	multimedia data file (text/graphics)
7	lowrd3 — execu1	commercial utility
8	graph2 — lowrd2 — execu2	multimedia application (graphics/sound/executable)
9	source1 — lowrd3 — graph6	multimedia data or source file (source/compressed binary/image)
10	audio1 — text4	multimedia data file (sound/text)
11	lowrd1 — execu4	statistical application with data
12	graph7 — text5	multimedia data file (text/graphics)
13	lowrd2 — text6	multimedia data file (sound/text)
14	text3 — audio1 — graph5	multimedia data file (text/sound/graphics)
15	lowrd1 — text4 — source2	source file for multimedia program (text/source code)
16	text7 — lowrd2 — graph3	multimedia data file (text/compressed audio/graphics)
17	graph4 — audio1 — execu5	multimedia application (sound/graphics)
18	execu4 — graph7 — text4	multimedia application (graphics/text)
19	objec3 — lowrd3 — execu6	commercial utility
20	objec2 — audio1 — execu2	audio application

tem does consistently better than `compress` and is comparable to *Compact Pro* on standard industrial benchmarks.[9] Improving algorithms and adding or substituting new heuristics would also yield more savings.

Execution times and speed optimizations

In this section we compare, in *approximate units*, the running time of the heterogeneous compressor against those of the four commercial systems the savings rates of which for our test files are documented above. The units are approximate for two reasons. First, because the four test systems are commercial the source code for three of them is not publicly available*, which renders an exact measure of *user* time infeasible. This concern is in part assuaged by the non-multitasked, single-user nature of the microcomputer operating systems on which three (`compress` for Linux notwithstanding) of the commercial systems reside. Second, however, the drastic architectural and organizational differences among the various native machines renders uniform comparisons unreliable. This applies even to normalized execution times because the host machines differ not merely in clock cycle speed, but in instruction set architecture and dynamic instruction frequencies for similar compression algorithms. The *exact* running times reported in this section is only that of the heterogeneous

* As noted, however, the Lempel–Ziv implementation employed by *StuffIt Classic* is nearly identical to that of Unix `compress`.

NetApp; Rackspace Exhibit 1014 Page 44

compressor. These comprise the non-commercial* compression systems for which source code is available for profiling. For the commercial systems we report the observed wall clock time to provide a standard of comparison, but note that the host machines vary in computational power.

Table IV. Results of the four popular commercial programs and the heterogeneous compression system, applied to the 20 test files

File number	Original length	Unix `compress`	PKZIP v1.10	*StuffIt Classic*	*Compact Pro* v1.32	Heterogeneous compressor
1	39,348	20,578	17,119	20,575	16,831	16,315
2	44,202	44,202	39,813	40,412	41,112	37,388
3	46,629	46,629	46,629	43,261	40,367	36,477
4	59,254	52,076	40,571	45,202	41,607	38,007
5	169,108	168,903	151,478	149,701	148,917	134,524
6	100,476	69,771	53,043	65,417	52,349	50,906
7	131,663	131,663	103,544	106,643	109,979	96,429
8	220,644	190,971	137,886	173,677	137,401	127,384
9	301,805	145,993	112,503	137,685	115,096	103,730
10	255,306	204,457	191,378	206,193	183,313	168,675
11	59,305	30,178	22,782	29,701	22,858	21,774
12	51,715	51,715	43,032	46,462	44,107	40,229
13	63,189	63,189	58,247	59,569	59,934	54,481
14	196,789	176,276	196,789	172,486	151,057	137,052
15	148,908	73,555	63,748	75,595	64,618	63,778
16	164,535	141,067	132,992	135,245	110,093	104,175
17	203,912	203,912	184,657	189,398	202,821	170,564
18	200,640	128,675	107,728	125,461	104,711	101,674
19	366,557	265,114	198,727	265,027	198,756	187,659
20	278,152	223,277	193,980	224,943	191,763	181,030
Total	3,102,137	2,432,201	2,096,646	2,312,653	2,037,690	1,872,251

The running times for the commercial systems on the entire test suite documented above appear in Table VI. All of the execution times are measured in wall clock units except for the heterogeneous compressor's, which is a total of user times as reported by `prof`, the C profiler under Unix. The wall clock time was empirically observed not to differ noticeably from this total on an unloaded Unix machine. The commercial systems were similarly tested on unloaded (or single-task) systems.

For Unix `compress`, the mean running time was 26 s, where the average was taken over runs on different Sun workstations of comparable power (documented below). A Unix implementation of PKZIP was also tested on one of these Sun workstations, and achieved an execution time of 56 s – only slightly better than the personal computer version. The running time of 856 s placed the heterogeneous compressor in the middle to high end of the commercial compressors in terms of running time.

* For this purpose we continue to consider Unix `compress` commercial, due to its wide range of versions.

Table V. Percent savings for the test compression systems*

File number	Unix `compress` (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix `compress`	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix `compress`, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

NetApp; Rackspace Exhibit 1014 Page 47

We would like to thank Leonid Broukhis, Graham Toal, and Kenneth Zeger for discussions on some of the research reported here. We also thank Jonathan Eifrig, Bill Goodman, and Tom Lane for guidance on several technical issues. Finally, we thank the anonymous reviewers for their comments and suggestions, especially for introduction to relevant literature in arithmetic coding.

REFERENCES

1. James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.
2. Phillip W. Katz, PKZIP. Commercial compression system, version 1.1, 1990.
3. Sun Microsystems, compress. Commercial compression system, operating system version 5.3, September 1992.
4. Raymond Lau, StuffIt Classic and StuffIt Deluxe. Commercial compression system, 1990.
5. Bill Goodman, Compact Pro. Commercial compression system, v1.32, 1991.
6. Terry A. Welch, 'A technique for high performance data compression', *IEEE Computer*, **17**(6), 8–19 (1984).
7. Gilbert Held and Thomas R. Marshall, *Data Compression: Techniques and Applications: Hardware and Software Considerations*, 3rd edn, John Wiley and Sons, 1991.
8. Leonid Broukhis, Freeze implementation of LZHuf algorithm. comp.sources.misc archives, Internet, 1991.
9. Jean-Loup Gailly, comp.compression benchmark (Calgary test corpus). In comp.compression FAQ list, J. Gailly, (ed.), 1992.
10. Jeffrey S. Vitter, 'Dynamic Huffman Coding', *ACM Transactions on Mathematical Software*, (June 1989).
11. J. Ziv and A. Lempel, 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory*, **23**,(3), 337–343 (1977).
12. J. Ziv and A. Lempel, 'Compression of individual sequences via variable-rate coding', *IEEE Transactions on Information Theory*, **24**(5), 530–546 (1978).
13. Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan and Victor K. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, 320–330 (April 1986).
14. Yooichi Tagawa, Haruhiko Okumura and Haruyasu Yoshizaki, LZHuf: encoding/decoding module for LHarc. Compression system, version 0.03 (Beta), 1989.
15. Haruyasu Yoshizaki, LHA: A high-performance file-compression program. Compression system, version 2.11, 1991.
16. Edward R. Fiala and Daniel H. Greene, 'Data compression with finite windows', *Communications of the ACM*, 490–505 (1989).
17. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, Maryland, second edition, 1987.
18. Graham Toal. Personal communication. Unpublished, 1992.
19. Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
20. Ian F. Darwin, file (program). Berkeley Unix operating system, 1987.
21. David A. Huffman, 'A method for the construction of minimum-redundancy codes', *Proceedings of the IRE*, number 40, 1952, pp. 1098–1101.
22. Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communications*, University of Illinois Press, Urbana and Chicago, 1963.
23. Robert Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
24. Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
25. Sheldon Ross, *A First Course in Probability*, Macmillan Publishing Company, New York, third edition, 1988.
26. Ian H. Witten, Radford Neal and John G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**(6), 520–540 (1987).
27. Independent JPEG Group. 'JPEG image compression system', think.com FTP archives, Internet, 1994.
28. Jean-Loup Gailly. comp.compression/comp.compression.research FAQ list. J. Gailly (ed.), URL `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html`, 1994.
29. James A. Storer, *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
30. Graham Toal. C implementation of dynamic Huffman compressor by J. S. Vitter. comp.source.misc archives, Internet, 1990.

NetApp; Rackspace Exhibit 1014 Page 48

Wiley Online Library

Log in / Register

Advertisement

Go to old article view

PDF

Info

Journal of Software: Practice and Experience

Explore this journal >

Text size Share

Article

Automatic synthesis of compression techniques for heterogeneous files

William H. Hsu, Amy E. Zwarico

First published: October 1995 Full publication history

DOI: 10.1002/spe.4380251003 View/save citation

Cited by: 1 article Citation tools

Am score

View issue TOC
Volume 25, Issue 10
October 1995
Pages 1097–1116

Abstract

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix compress, *Stufflt*, and *Compact Pro*, and show that our system provides better space savings.

» Continue reading full article

Related content

Articles related to the one you are viewing

The articles below have been selected for you based on the article you are currently viewing.

Compression techniques for Chinese text
Phil Vines, Justin Zobel
October 1998

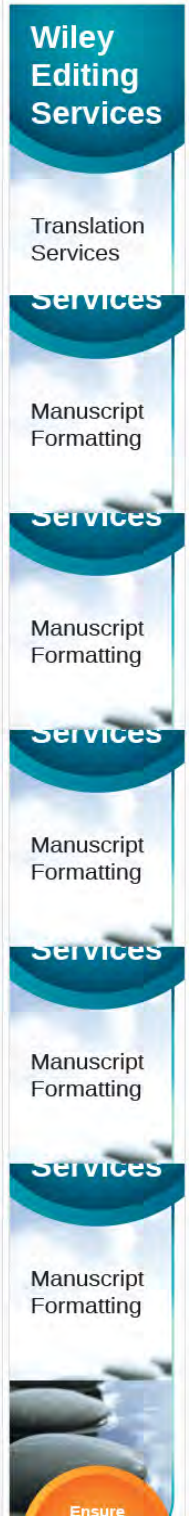
Lempel-Ziv compression of highly structured documents
Joaquin Adiego, Gonzalo Navarro, Pablo de la Fuente
25 January 2007

Word-based text compression
Alistair Moffat
February 1989

LZgrep: a Boyer–Moore string matching tool for Ziv–Lempel compressed text
Gonzalo Navarro, Jorma Tarhio
6 May 2005

Design and implementation of a file system with on-the-fly data compression for GNU/Linux
Praveen B., Deepak Gupta, Rajat Moona

Advertisement



5 August 1999

» Citing Literature

your manuscript is ready for submission

WILEY

Copyright © 1999 - 2016 John Wiley & Sons, Inc. All Rights Reserved

WILEY

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1097–1116 (OCTOBER 1995)

Automatic Synthesis of Compression Techniques for Heterogeneous Files

WILLIAM H. HSU
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
(email: bhsu@cs.uiuc.edu, voice: (217) 244-1620)

AND

AMY E. ZWARICO
Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.
(email: amy@cs.jhu.edu, voice: (410) 516-5304)

SUMMARY

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix `compress`, *StuffIt*, and *Compact Pro*, and show that our system provides better space savings.

KEY WORDS: adaptive/selective data compression algorithms; redundancy metrics; heterogeneous files; program synthesis

INTRODUCTION

The primary motivation in studying compression is the savings in space that it provides. Many compression algorithms have been implemented, and with the advent of new hardware standards, more techniques are under development. Historically, research in data compression has been devoted to the development of algorithms that exploit various types of redundancy found in a file. The shortcoming of such algorithms is that they assume, often inaccurately, that files are homogeneous throughout. Consequently, each exploits only a subset of the redundancy found in the file.

Unfortunately, no algorithm is effective in compressing all files.[1] For example, dynamic Huffman coding works best on data files with a high variance in the frequency of individual characters (including some graphics and audio data), achieves mediocre performance on natural language text files, and performs poorly in general on high-redundancy binary data. On the other hand, run length encoding works well on high-redundancy binary data, but performs very poorly on text files. Textual substitution works best when multiple-character strings tend to be repeated, as in English text, but this performance degrades as the average

CCC 0038–0644/95/101097–20
©1995 by John Wiley & Sons, Ltd.
Received 20 April 1994
Revised 5 February 1995

length of these strings decreases. These relative strengths and weaknesses become critical when attempting to compress *heterogeneous* files. Heterogeneous files are those which contain multiple types of data such as text, images, binary, audio, or animation. Consequently, their constituent parts may have different degrees of compressibility. Because most compression algorithms are either tailored to a few specific classes of data or are designed to handle a single type of data at a time, they are not suited to the compression of heterogeneous files. In attempting to apply a single method to such files, they forfeit the possibility of greater savings achievable by compressing various segments of the file with different methods.

To overcome this inherent weakness found in compression algorithms, we have developed a *heterogeneous compressor* that automatically chooses the best compression algorithm to use on a given variable-length block of a file, based on both the qualitative and quantitative properties of that segment. The compressor determines and then applies the selected algorithms to the blocks separately. Assembling compression procedures to create a specifically tailored program for each file gives improved performance over using one program for all files. This system produces better compression results than four commonly available compression packages, PKZIP,[2] Unix `compress`,[3] *StuffIt*,[4] and *Compact Pro*[5] for arbitrary heterogeneous files.

The major contributions of this work are twofold. The first is an improved compression system for heterogeneous files. The second is the development of a method of statistical analysis of the compressibility of a file (its redundancy types). Although the concept of redundancy types is not new,[6,7] synthesis of compression techniques using redundancy measurements is largely unprecedented. The approach presented in this paper uses a straightforward program synthesis technique: a *compression plan*, consisting of instructions for each block of input data, is generated, guided by the statistical properties of the input data. Because of its use of algorithms specifically suited to the types of redundancy exhibited by the particular input file, the system achieves consistent average performance throughout the file, as shown by experimental evidence.

As an example of the type of savings our system produces, consider compressing a heterogeneous file (such as a small multimedia data file) consisting of 10K of low redundancy (non-natural language) ASCII data, 10K of English text, and 25K of graphics. In this case, a reasonably sophisticated compression program might recognize the increased savings achievable by employing Huffman compression, to better take advantage of the fact that the majority of the data is graphical. However, none of the general-purpose compression methods under consideration are optimal when used alone on this file. This is because the text part of this file is best compressed by textual substitution methods (e.g., Lempel–Ziv) rather than statistical methods, while the low-redundancy data* and graphics parts are best compressed by alphabetic distribution-based methods (e.g., arithmetic or dynamic Huffman coding) rather than Lempel–Ziv or run-length encoding. This particular file totals 45K in length before compression. A compressor using pure dynamic Huffman coding only achieves about 7 per cent savings for a compressed file of length 42.2K. One of the best general-purpose Lempel–Ziv compressors currently available[8,9] achieves 18 per cent savings, producing a compressed file of length 37.4K. Our system uses arithmetic coding on the first and last segments and Lempel–Ziv compression on the text segment in the middle, achieving a 22 per cent savings and producing a compressed file of length 35.6K. This is a 4 per cent improvement over the best commercial system.

* This denotes, in our system, a file with a low rate of repeated strings.

NetApp; Rackspace Exhibit 1014 Page 52

The purpose of our experiments was to verify the conjecture that a selective system for combining methods can improve savings on a significant range of heterogeneous files, especially multimedia data. This combination differs from current adaptive methods in that it switches among compression paradigms designed to remove very different types of redundancy. By contrast, existing adaptive compression programs are sensitive only to changes in particular types of redundancy, such as run-length, which do not require changing the underlying algorithm used in compression. Thus they cannot adapt to changes in the type of redundancy present, such as from high run-length to high character repetition. The superiority of our approach is demonstrated in our experimental section.

This paper begins with a presentation of existing approaches to data compression, including a discussion of hybrid and adaptive compression algorithms and a description of four popular commercial compression packages. These are followed by documentation on the design of the heterogeneous compression system, analysis of experimental results obtained from test runs of the completed system, and comparison of the system's performance against that of commercial systems. Finally, implications of the results and possibilities for future work are presented.

RELATED WORK

It is a fundamental result of information theory that there is no single algorithm that performs optimally in compressing all files.[1] However, much work has been done to develop algorithms and techniques that work nearly optimally on all classes of files. In this section we discuss adaptive algorithms, composite algorithms, and four popular commercial compression packages.

Adaptive compression algorithms and composite techniques

Exploiting the heterogeneity in a file has been addressed in two ways: the development of *adaptive* compression algorithms, and the composition of various algorithms. Adaptive compression algorithms attune themselves gradually to changes in the redundancies within a file by modifying parameters used by the algorithm, such as the dictionary, during execution.

For example, adaptive alphabetic distribution-based algorithms such as dynamic Huffman coding[10] maintain a tree structure to minimize the encoded length of the most frequently occurring characters. This property can be made to change continuously as a file is processed.

An example of an adaptive textual substitution algorithm is Lempel–Ziv compression, a title which refers to two distinct variants of a basic textual substitution scheme. The first variant, known as LZ77 or the *sliding dictionary* or *sliding window* method, selects positional references from a constant range of preceding strings.[1, 11] These 'back-pointers' literally encode position and length of a repeated string. The second variant, known as LZ78 or the *dynamic dictionary* method, uses a dictionary structure with a *paging* heuristic. When the dictionary – a table of strings from the file – is completely filled, the contents are not discarded. Instead, an auxiliary dictionary is created and updated while compression continues using the main dictionary. Each time this auxiliary table is filled, its contents are 'swapped' into the main dictionary and it is cleared. The maintenance of dictionaries for textual substitution is analogous to the semi-space method of garbage collection, in which two pages are used but only one is 'active' – these are exchanged when one fills beyond a preset threshold. Another adaptive variant of this algorithm is the Lempel–Ziv–Welch

NetApp; Rackspace Exhibit 1014 Page 53

(LZW) algorithm, a descendant of LZ78 used in Unix `compress`.[6, 12] Both LZW and LZ78 vary the length of strings used in compression.[6, 12]

Yet another adaptive (alphabetic distribution-based) compression scheme, the Move-To-Front (MTF) method, was developed by Bentley *et al.*[13] In MTF, the 'word code' for a symbol is determined by the position of the word in a sequential list. The word list is ordered so that frequently accessed words are near the front, thus shortening their encodings.

Adaptive compression algorithms are not appropriate to use with heterogeneous files because they are sensitive only to changes in the particular redundancy type with which they are associated, such as a change in the alphabetic distribution. They do not exploit changes across different redundancy types in the files. Therefore a so-called adaptive method typically cannot directly handle drastic changes in file properties, such as an abrupt transition from text to graphics. For example, adaptive Huffman compressors specially optimized for text achieve disproportionately poor performance on certain image files, and vice versa. As the use of multimedia files increases, files exhibiting this sort of transition will become more prevalent.

Our approach differs from adaptive compression because the system chooses each algorithm (as well as the duration of its applicability) before compression begins, rather than modifying the technique for each file during compression. In addition, while adaptive methods make modifications to their compression parameters on the basis of single bytes or fixed length strings of input, our heterogeneous compressor bases its compression upon statistics gathered from larger blocks of five kilobytes. This allows us to handle much larger changes in file redundancy types. This makes our system less sensitive to residual statistical fluctuations from different parts of a file. We note that it is possible to use an adaptive algorithm as a primitive in the system.

Another approach to handling heterogeneous files is the composition of compression algorithms. Composition can either be accomplished by running several algorithms in succession or by combining the basic algorithms and heuristics to create a new technique. For example, recent implementations of 'universal' compression programs execute the Lempel–Ziv algorithm and dynamic Huffman coding in succession, thus improving performance by combining the string repetition-based compression of Lempel–Ziv with the frequency-based compression strategy of dynamic Huffman coding. One commercial implementation is *LHarc*.[14,15] Our system exploits the same savings since it uses the *Freeze* implementation of the Lempel–Ziv algorithm, which filters Lempel–Ziv compressed output through a Huffman coder. An example of a truly composite technique is the compression achieved by using Shannon–Fano tries* in conjunction with the Fiala–Greene algorithm (a variant of Lempel–Ziv)[16] in the PKZIP[2] commercial package. Tries are used to optimally encode strings by character frequency.[17] PKZIP was selected as the representative test program from this group in our experiment due to its superior performance on industrial benchmarks.[9]

Our approach generalizes the ideas of successively executing or combining different compression algorithms by allowing any combination of basic algorithms within a file. This includes switching from among algorithms an arbitrary number of times within a file. The algorithms themselves may be simple or composite and may be adaptive. All are treated as atomic commands to be applied to portions of a file.

* A *trie* is a tree of variable degree ≥ 2 such that (1) each edge is labelled with a character, and the depth of any node represents one more than the number of characters required to identify it; (2) all internal nodes are intermediate and represent prefixes of keys in the trie; (3) keys (strings) may be inserted as leaves using the minimum number of characters which distinguish them uniquely. Thus a generic trie containing the strings *computer* and *compare* would have keys at a depth of five which share a common prefix of length four.

NetApp; Rackspace Exhibit 1014 Page 54

The problem of heterogeneous files was addressed by Toal[18] in a proposal for a naive heterogeneous compression system similar to ours. In such a system, files would be segmented into fixed-length encapsulated blocks; the optimal algorithm would be selected for each block on the basis of their simple taxonomy (qualitative data type) only; and the blocks would be *independently* compressed. Our system, however, performs more in-depth statistical analysis in order to make a more informed selection from the database of algorithms. This entails not only the determination of qualitative data properties but the computation of metrics for an entire block (as opposed to sporadic or random sampling from parts of each block). Furthermore, normalization constants for selection parameters (i.e. the redundancy metrics) are fitted to observed parameters for a test library. Finally, a straightforward but crucial improvement to the naive encapsulated-block method is the implementation of a multi-pass scheme. By determining the complete taxonomy (data type and dominant redundancy type) in advance, any number of contiguous blocks which use the same compression method will be treated as a single segment. Toal observed in preliminary experiments that the overhead of changing compression schemes from one block to another dominated the additional savings that resulted from selection of a superior compression method.[18] This overhead is attributable to the fact that blocks compressed independently (even if the same method is used) are essentially separate files and assume the same startup overhead of the compression algorithm used.* We have determined experimentally that merging contiguous blocks whenever possible obviates the large majority of changes in compression method. This eliminates a sufficient proportion of the overhead to make heterogeneous compression worthwhile.

Commercial products

One of the goals of this research was to develop a compression system which is generally superior to commercially available systems. The four systems we studied are PKZIP, developed for microcomputers running MS-DOS;[2] Unix `compress`;[3] and *StuffIt Classic*[4] and *Compact Pro*,[5] developed for the Apple Macintosh operating system. Each of these products performs its compression in a single pass, with only one method selected per file. Thus, the possibility of heterogeneous files is ignored.

Unix `compress` uses an adaptive version of the Lempel–Ziv algorithm.[6] It operates by substituting a fixed-length code for common substrings. `compress`, like other adaptive textual substitution algorithms, periodically tests its own performance and reinitializes its string table if the amount of compression has decreased.

StuffIt makes use of two sets of algorithms: it first detects special-type files such as image files and processes them with algorithms suited for high-resolution color data; for the remaining files, it queries the operating system for the explicit file type given when the file was created, and uses this information to choose either the LZW variant of Lempel–Ziv,[4, 6] dynamic Huffman coding, or run-length encoding. This is a much more limited selection process than what we have implemented. Additionally, no selection of compression methods is attempted within a file. *Compact Pro* uses the same methodology as *StuffIt* and `compress`, but incorporates an improved Lempel–Ziv derived directly from LZ77.[11] The public-domain version of *StuffIt* is derived from Unix `compress`, as is evident from the similarity of their performance results.

* For purposes of comparison, the block sizes tested by Toal were nearly identical to those used in our system (ranging upwards from 4K).

NetApp; Rackspace Exhibit 1014 Page 55

Compression systems such as *StuffIt* perform simple selection among alternative compression algorithms. The important problem is that they are underequipped for the task of fitting a specific technique to each file (even when the uncompressed data is homogeneous). *StuffIt* uses few heuristics, since its algorithms are intended to be 'multipurpose' . Furthermore, only the file type is considered in selecting the algorithm – that is, no measures of redundancy are computed. Earlier versions of *StuffIt* (which were extremely similar to Unix `compress`) used composite alphabetic and textual compression, but made no selections on the basis of data characteristics. The chief improvements of our heterogeneous compressor over this approach are that it uses a two-dimensional lookup table, indexed by file properties and quantitative redundancy metrics, and – more important – that it treats the file as a collection of heterogeneous data sets.

THE HETEROGENEOUS COMPRESSOR

Our heterogeneous compressor treats a file as a collection of fixed size blocks (5K in the current implementation), each containing a potentially different type of data and thus best compressed using different algorithms. The actual compression is accomplished in two phases. In the first phase, the system determines the type and compressibility of each block. The compressibility of each block of data is determined by the values of three quantitative metrics representing the alphabetic distribution, the average run length and the string repetition ratio in the file. If these metrics are all below a certain threshold, then the block is considered fully compressed (uncompressible) and the program continues on to the next block. Otherwise, using the block type and largest metric, the appropriate compression algorithm (and possible heuristic) are chosen from the compression algorithm database. The compression method for the current block is then recorded in a small array-based map of the file, and the system continues.

The second phase comprises the actual compression and an optimization that maximizes the size of a segment of data to be compressed using a particular algorithm. In this optimization, which is interleaved with the actual compression, adjacent blocks for which exactly the same method have been chosen are merged into a single block. This merge technique maximizes the length of segments requiring a single compression method by greedily scanning ahead until a change of method is detected. Scanning is performed using the array map of the file generated when compression methods were selected from the database. A compression history, needed for decompression, is automatically generated as part of this phase.

The newly compressed segments are written to a buffer by the algorithm, which stores the output data with the compression history. The system then writes out the compressed file and exits with a signal to the operating system that compression was successful.

From this two-pass scheme it is straightforward to see why this system is less susceptible than traditional adaptive systems to biases accrued when the data type changes abruptly during compression. Adaptive compressors perform all operations myopically, sacrificing the ability to see ahead in the file or data stream to detect future fluctuations in the type of data. As a result, adaptive compressors retain the statistical vestiges of the old method until these are 'flushed out' by new data (or balanced out, depending upon the process for paging and aging internal data structures such as dictionaries). Thus adaptive compressors may continue to suffer the effects of bias, achieving suboptimal compression. On the other hand, by abruptly changing compression algorithms, our technique completely discards all remnants of the 'previous' method (i.e. the algorithm used on the preceding segment). This

NetApp; Rackspace Exhibit 1014 Page 56

allows us to immediately capitalize on changes in data. In addition, merging contiguous blocks of the same data type acquires the advantage of incurring all the overhead *at once* for switching to what will be the best compression method for an entire variable-length segment. The primary advantage of adaptive compression techniques over our technique is that the adaptive compression algorithms are 'online' (single-pass). This property increases compression speed and, more important, gives the ability to compress a data stream (for instance, incoming data packets in a network or modem transmission) in addition to files in secondary storage or variable-length buffers.

The remainder of this section presents the system. We begin with a description of the calculation of the block types and the redundancy metrics. We also explain the use of the metrics as absolute indicators of compressibility, and then describe the compression algorithms used and the structure of the database of algorithms. A discussion of implementation details concludes the section.

Property analysis

The compressibility of a block of data and the appropriate algorithm to do so are determined by the type of data contained in a block and the type of redundancy (if any) in the data. These two properties are represented by four parameters: the *block type*, and the three *redundancy metrics*. The block type describes the data in the block – text, binary, graphical, etc. The three redundancy metrics are the degree of variation in character frequency, average run length in the file, and the string repetition ratio of the file. They provide a quantitative measure of how compressible the block is and which type of redundancy is most evident in the block. The use of both quantitative redundancy measures (redundancy metrics) and qualitative characteristics (block types) as indicators for data compressibility is advocated by Held[7] and Salton.[19] We have refined the process for computing those attributes referred to as *datanalysis results* by Held[7] and as *statistical language characteristics* by Salton[19] to obtain an actual guide for compression. The remainder of this section describes how these four parameters are determined for each block.

Block types

The *block type* describes the nature of a segment of input data. There are ten classifications of data in this system: ANSI text, non-natural language text (hexadecimal encodings of binary data), natural language text, computer source code, low redundancy binary, digitized audio, low resolution graphics, high-resolution graphics, high-redundancy binary executable, and binary object data. ANSI text is composed of characters from a superset of the ASCII alphabet. Non-natural language text contains primarily ASCII text but does not follow a distribution of characters like that of human languages. Examples are computer typesetting data, *uuencoded* and *BinHex* encoded data (which has the same character distribution as binary data but is converted to text for ease of transmission). Natural language text includes text written in English as well as other languages which are representable by the Roman (ASCII) alphabet. Most European languages (including the ones using the Cyrillic alphabet), special symbols excluded, fall into this category, as do the Pinyin and Katakana romanizations of the Chinese and Japanese languages (as opposed to their digital encodings). Computer source code uses the ASCII alphabet but characters are distributed with a different frequency than in natural language text. Low-redundancy binaries usually contain compressed data, but may also include data which is merely difficult to compress. Audio

NetApp; Rackspace Exhibit 1014 Page 57

data are very high in redundancy; audio files (and audio segments of multimedia files) are usually extremely large. Low-resolution graphics have long runs of contiguous repeated bits but unlike high-resolution graphics are not suited to lossy compression. High-resolution graphics include color and grayscale and may be compressed with lossy methods. Binary executables, like low-resolution graphics, have long runs of contiguous repeated bits and comprise all compiled programs on a computer system. Finally, object data has slightly shorter runs but is similarly redundant.

To determine the block type we use a procedure `new-file` which is our extension of the Unix `file` command.[20] `file` works by examining the first 512 bytes of a file and comparing the pattern of data contained in it to a collection of known data patterns from Unix and other operating systems. `new-file` works in a similar fashion, with two modifications. First, it examines and compares not only the first 512 bytes of a data set, but also 512 bytes in the middle of the set and the 512 bytes at the end (if they exist). This provides a better indication of the *primary* data type of a file by taking into account the possibility that the properties may change anywhere within the file. Thus, `new-file` decides on the 'most applicable' data type by a majority vote (or the first data type detected in the case of a three-way tie). The other change is that the known patterns of data have been increased by adding three graphics patterns.

Redundancy metrics

The redundancy metrics are quantitative measures that are used to determine the compressibility of a block of data. They are: the *degree of variation in character frequency* or *alphabetic distribution*, M_{AD}; the *average run length* of the block, M_{RL}; and the *string repetition ratio* of the block, M_{SR}. In general, these three manifestations of redundancy are independent. Each of the redundancy types is exploited by different compression algorithms. *Frequency of characters* is exploited by arithmetic or alphabetic encoding algorithms. In arithmetic coding data is represented by an interval that is calculated from the probability distribution of data. With alphabetic coding algorithms such as the Huffman[21] and Shannon–Fano[22] algorithms, more frequently occurring characters are replaced by shorter units than the less frequently occurring characters. *Contiguous strings*, long strings of identical units occurring next to one another, are exploited by run length encoding algorithms.[23] In these algorithms, contiguous strings are replaced by a single occurrence of the string, called a *run*, plus a count of the number of identical strings following. Both alphabetic distribution and average run length are sometimes characterized as statistical redundancy metrics.[24] *Recurrent strings*, which occur repeatedly in the input stream with any number of interleaved symbols, are exploited by *textual substitution algorithms* such as Lempel–Ziv.[6,11,12] In these algorithms, recurrent instances are replaced with positional references (pointers) to the original instance.

Experimental evidence for the efficacy of quantitative redundancy measures is described in texts by Storer[1] and Shannon.[22] Shannon provided an estimate of the entropy of English text, approximately bounding it to be between one and two bits per character.[22] This was determined experimentally by presenting fragments of (unfamiliar) English text to human subjects and recording the frequency with which they guessed unknown letters. The fragments were revealed character by character, so that letters at the end of long or uncommon words were easiest to guess and letters at the beginnings of words were hardest. The observation that binary executables are known to possess high average run lengths is found in Storer.[1] However, this property is rarely exploited or measured.

NetApp; Rackspace Exhibit 1014 Page 58

Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function G to be a number between 0 and 10 so as to simplify comparison among the different metrics. The gamma distribution was chosen because the graph of each of the unscaled redundancy metrics for a test set of 50 files, when plotted on a histogram, approximated a gamma distribution. Normal and χ^2 distributions were also considered, but these proved to be too specific for the application (since they are both specific parametric cases of the gamma distribution). The gamma distribution is defined as follows (cf Ross[25]):

$$\begin{aligned} G_\tau(x_\tau) &= \int_0^{x_\tau} f_\tau(x)\,\mathrm{d}x \\ f_\tau(x) &= \frac{\lambda_\tau \mathrm{e}^{-\lambda_\tau x}(\lambda_\tau x)^{t_\tau - 1}}{\Gamma(t_\tau)} \\ \Gamma(t_\tau) &= \int_0^{\infty} \mathrm{e}^{-y} y^{t_\tau - 1}\,\mathrm{d}y \end{aligned}$$

where f_τ is the density function, Γ is the gamma function, x_τ is the unnormalized measure, t_τ is the shape parameter for the gamma distribution, and λ_τ is the scale parameter for the gamma distribution. The τ subscript simply represents the redundancy type under consideration, i.e. AD, RL, or SR, respectively. The shape and scaling parameters, t_τ and λ_τ respectively, were determined by fitting the best gamma distribution curve to the data set. This was done by performing the preferred compression method for each file and tabulating the induced ratio among normalized metrics to yield the desired parameter values for each segment. These were then averaged to obtain the empirical scaling parameters.

The *alphabetic distribution metric* (the degree of variation in character frequency) of a block is calculated by taking the population (root-mean-square) standard deviation of the ordinal values of characters in the block and dividing it by the block length (in bytes). The M_{AD} metric is calculated by the following formulas:

$$\begin{aligned} M_{\mathrm{AD}} &= 10 * G_{\mathrm{AD}}(x_{\mathrm{AD}}) \\ x_{\mathrm{AD}} &= \frac{\alpha}{\text{block length in bytes}} \\ \alpha &= \sqrt{\frac{\sum_{c \in \text{charset}} (c-\mu)^2}{256}}, \end{aligned}$$

where c is the ordinal value of a character and μ is the average ordinal value of all characters in a block. The normalization uses $t_{\mathrm{AD}} = 1.70$ and $\lambda_{\mathrm{AD}} = 53.0$ as parameters.

The *average run length metric* is obtained by dividing the number of bits in a block by the number of runs. A run is defined to be a repetition of symbols (either bits or bytes). Our implementation takes both bitwise and bytewise run lengths. For example, if $f = 0001111001110000$ is a file of 16 bits, then the number of bit runs is 5, and the number of byte runs is 2. The scaled metric M_{RL} is obtained by:

$$\begin{aligned} M_{\mathrm{RL}} &= 10 * G_{\mathrm{RL}}(x_{\mathrm{RL}}) \\ x_{\mathrm{RL}} &= \frac{\text{file length in bits}}{\text{number of runs}} \end{aligned}$$

with gamma distribution parameters $t_{RL} = 0.50$ and $\lambda_{RL} = 12.0$.

The *string repetition ratio metric* is the total number of n-bit strings in the block divided by the number of distinct n-bit strings (up to 100K). In our implementation, n is 32, the word size of our machine. The normalized metric M_{SR} is obtained by:

$$M_{SR} = 10 * G_{SR}(x_{SR})$$
$$x_{SR} = \frac{\text{number of } n \text{ bit strings}}{\text{number of distinct } n \text{ bit strings}}$$

with gamma distribution parameters $t_{SR} = 0.18$ and $\lambda_{SR} = 0.2$.

The alphabetic distribution and average run length metrics can be calculated in linear time. The string repetition ratio can be computed in $O(n \log n)$ time using a dictionary data structure. For simplicity, and because a (small) constant amount of data is scanned, we use an $O(n^2)$ version. New strings are stored in an array rather than a binary tree, which would require more insertion overhead (and is not worth while for the 5K block length used in the current system). Our routine integrates $f_\tau(x)$ by Simpson's Rule with $n = 10$ intervals.

The largest of the three metrics is assumed to represent the most significant type of redundancy present in the block. It is expected that compression will decrease at least one of the metrics, and experiments conducted on a wide variety of files have proven this convention to be reliable. Experiments have also shown that if all the normalized metrics are smaller than 2.5, the file is considered not compressible, and the system records a verdict of 'uncompressible' on the current block. If at least one of the parameters is greater than 2.5, the file is considered compressible. The maximum of the normalized metrics is then selected and used in conjunction with the file type to select the appropriate compression algorithm from the lookup table described in the following section. A negative compressibility test does not always imply that all three metrics are below the threshold. In some cases, the only redundancy type for which a metric is above the threshold accesses a null entry in the database of compression algorithms. This is interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm.

The algorithm and heuristic database

The compression algorithms and attendant heuristics are organized into the 10 by 3 table illustrated by Table I. The 10 file descriptors are the row indices and the 3 metrics are the column indices. Each entry of the table contains descriptors which are used to access the code for an algorithm-heuristic pair. It should be noted that four of the entries are blank (indicated by an *). A blank entry indicates that the combination of block type and highest metric are very unusual. In this case, the next highest metric is used instead, provided that it is above the threshold. As an example of using this table, consider a high-redundancy binary executable file whose highest metric is the string repetition metric M_{SR}. Together, this pair indicates that the Lempel–Ziv compression algorithm with the Freeze heuristic will be used.

The algorithms

There are four basic algorithms used by the system: arithmetic coding,[26] Lempel–Ziv,[8] run length encoding (RLE),[23] and JPEG for image/graphics compression.[27]

Arithmetic coding algorithms compress data by representing that data by an interval of

NetApp; Rackspace Exhibit 1014 Page 60

Table I. Database of compression algorithms†

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
hexadecimal	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
natural language	arithmetic coding	*	Lempel–Ziv
	*	*	freeze
source code	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
low redundancy	*	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	*
audio	*	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
low resolution	*	run-length encoding	Lempel–Ziv
graphic	*	n-bit run count	freeze
high resolution	JPEG	run-length encoding	JPEG
color graphic	improved Huffman	n-bit run count	improved Huffman
high redundancy	arithmetic coding	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	freeze
object	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze

† Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

real numbers between zero and one. The width of this interval is inversely proportional to the number of symbols encoded, and the decrease in width is directly proportional to the frequency of the original symbols. Thus the interval specifies the encoded message via its bounds, with the precision (distance) of these bounds reflecting the information content of the message. The end result is that arithmetic coding achieves, in practice, much better space savings than Huffman coding and its dynamic implementations because of its higher likelihood of actually achieving the theoretical lower bound.[24, 28] Although early arithmetic coding algorithms performed too slowly to be of practical use,[29] the implementation of the Witten–Neal–Cleary algorithm used here[26] is optimized for speed – at some cost in space savings, but without giving up its advantage over dynamic Huffman coding. The reader is referred to Bell *et al*[24] for a thorough overview of arithmetic coding. We should note that in earlier implementation of the heterogeneous compressor we used a dynamic Huffman algorithm instead of arithmetic coding. We changed our implementation when we found that then Witten–Neal–Cleary algorithm[26] outperformed our implementation of dynamic Huffman coding[10, 30] in both space savings and execution time.

Run length encoding (RLE) algorithms compress data by replacing contiguous occurrences of a single-unit symbol (either bit or byte) by an efficiently coded count of these runs, usually a single occurrence of the symbol and the number of occurrences. We have implemented a straightforward RLE algorithm for our database, based on the description in Sedgewick.[23] In addition, bitwise and bytewise encoding are available as heuristics and the parameters of bitwise RLE are based on the RL metric.

Files with a high degree of string repetition are compressed using the Lempel–Ziv compression algorithm. It compresses data by replacing frequently occurring strings (with min-

imal regard of how far apart they occur) with compact pointers to the position of the first occurrence. Our implementation is a straightforward array-based encoding with constant-length codes. The algorithm maintains a dictionary of recurring strings in order to do the compression. In our system, the Lempel–Ziv algorithm is augmented with the *Freeze* heuristic. This heuristic suppresses paging of strings in the dictionary after it has been filled; that is, it prevents the replacement of previously encountered strings, regardless of how long ago or how infrequently the string has been encountered. *Freeze* is primarily a speed optimization, since it requires less computation than paging heuristics such as least recently used (LRU) or least frequently used (LFU), but it has been shown to work well for all but the least string-redundant files (including both binary executables and most text files). For files with extremely low string-repetition, our system usually selects Huffman compression.

The compression of high-resolution graphics and audio files uses a *lossy* compression scheme. Appropriately used, lossy algorithms guarantee that the decompressed file is similar enough to the original as to be nearly indistinguishable by human perception, and that repeated compression and decompression leads to limited cumulative 'damage' . The primary benefit of lossy compression is that it guarantees much higher compression ratios at a minimal tradeoff. For instance, a very-high-resolution color image can be compressed with much higher savings (possibly 95 per cent) if the user allows a small amount of noise, always less than 1 per cent per compression, to be introduced during each compression. Our system uses the JPEG system[27] for compressing high-resolution color and grayscale images. JPEG, which is divided into lossy and lossless parts, typically achieves compression ratios of between 15-to-1 and 25-to-1. The *potential* for this substantial savings is obtained by the Discrete Cosine Transform portion of the algorithm, a lossy method. This determines a limit on the amount of savings that can then be achieved by any lossless compressor. The actual savings are realized by a lossless portion, known as the *back end* which is applied to the preprocessed image data. The implementation of this module used in our system[27] is a Huffman coder. It is independent of the lossy front end and can be replaced with a run-length or textual-substitution based algorithm, to be selected by the synthesis system. In our implementation, we chose to retain the original Huffman back end, a different algorithm from the general-purpose dynamic Huffman coder which we also studied.[10, 30] This is because the JPEG Huffman coder is especially suited to the redundancy remaining after lossy preprocessing. It is worthy of mention that the JPEG developers have investigated the use of arithmetic coding back ends, which were found to be experimentally superior but were not used because of proprietary considerations.[27]

Implementation

The system consists of a driver module, four block analysis modules, and the synthesis module, which includes the database of compression algorithms. All modules are written in C and were tested on a Unix platform. The program uses a data directed style of implementation for choosing the compression algorithm to apply to a block. Thus, additional block types, compression algorithms and heuristics, and redundancy metrics can be added to the system with minimal modification of the source code. Only the database would have to be updated and the block analysis routines extended; the rest of the program would remain the same.

The driver performs two iterative passes through the file. It first performs block analysis on the file one 5K block at a time. This block size was chosen after experimentation showed that the response of the system to changes in block type became roughly stable as block

NetApp; Rackspace Exhibit 1014 Page 62

size exceeded 5K (i.e., did not significantly increase as block size did), and that a block size of 5K yielded highly accurate metrics (in only 1 of the 20 test files did the heterogeneous compressor select a suboptimal algorithm for any block). Finally, we found that the highest level of adaptivity without a noticeable *decrease* in accuracy was achieved at 5K, hence our choice of 5K as the block size.

For each block, the system invokes the four analysis modules – three for metric computation and normalization and one to determine the file properties – and stores their output. It then performs the metric comparison and combines the results with the file property to complete the table lookup for the current block. An identifying tag for the selected algorithm is written to the 'compression plan' , an array which stores one complete compression instruction per block (if the current block is deemed uncompressible, a 'skip' instruction is recorded).

We pause here to discuss the normalization of the metrics. Originally, we used a naive normalization method: direct algebraic scaling with experimentally determined constants for each metric. This did not, however, accurately reflect the statistical relationship between variance in character frequency and alphabetic redundancy. Also, the behavior of these functions at asymptotes led to poor approximation of the overall distribution of data segments in the test files. The result was that arithmetic coding was too often incorrectly chosen, resulting in inferior compression; and selection approached randomness as metric values for both string repetition and alphabetic distribution tended toward extreme values. Using the gamma normalization method described above resulted in an improvement in the selection of arithmetic coding. Among the 20 benchmark files, arithmetic coding was selected as the compression method in exactly those cases where the other methods performed worse.

The second pass performs the compression of each block. In order to improve performance, this pass includes a simple optimization step which circumvents the overhead of restarting compression after each fixed length block by merging contiguous blocks that are to be compressed using the same compression algorithms.

On this same pass through the file, the system compresses each of the newly merged blocks using the algorithm recorded in the compression plan. The compressed data is written to an output buffer, while the compressed length (which indicates where in the compressed file a compressed block begins and ends) and compression method are recorded in a separate history for reference at decompression time. If negative compression or no compression is achieved, or if the block was already marked uncompressible, then the data is copied directly to the output buffer (the full block length and a code for 'no compression' are recorded in the compression history). Upon reaching the end of the blocks, the system writes out the compressed data from the output buffers and prepends the encoded compression history to produce the final output file.

When decompression is invoked, the driver module opens the compressed file, interprets the history tag and performs the necessary operations. The tags are a stored version of the compression history in compact, encoded form. Since the heterogeneous system generates different compression sequences for each file, and since the length of a compressed block varies with both the length of the original block and the compression method used, these tags are necessary to guide the decompression process. Currently only the compressed lengths of each block and the method of compression are stored, but a checksum for the original (decompressed) block length can be added with negligible overhead. When executed in reverse order on each compressed block, the instructions in the history tags result in the original file. For simplicity and security, they are prepended to the compressed file (and can easily be encrypted).

EXPERIMENTAL RESULTS

Design and construction of the test files

To test the overall performance, the system was run on a set of 20 test files. These files range in length from approximately 39K to 366K, with representative files from each of the ten block types included in the test corpus.

The test files are designed to model certain types of heterogeneous files, including utilities for image viewing, business, or audio processing, and hypothetical multimedia databases and programs. To construct these files, a collection of 30 files from the Unix, Apple Macintosh, and MS-DOS (IBM PC) operating systems was created. These files are listed in Table II. To create the test corpus, they were concatenated in groups of 2 or 3. The resultant series of test files is listed in Table III. All of the source files were used. The goal was to generate as broad a range of permutations as possible (while restricting the generated files to those which are likely to exist in a typical user environment). This was performed manually with consideration toward combinatorial constraints and the criteria of realistic data modeling. Since all of the files in the source collection originate from common commercial sources or from public archives (with the exception of the source and object files, which are from the code for the heterogeneous compressor itself), the latter constraint was considerably simplified.

The assembled files were then ported to the test sites (a Sun workstation for Unix `compress` and our heterogeneous compressor; a Macintosh for *StuffIt* and *Compact Pro*; and an IBM 80486 machine for PKZIP). Binary file transfer mode was used to ensure that the file lengths agreed exactly among all platforms.

Performance

In this section, we review and analyze the performance of the heterogeneous compressor with respect to compression savings, as compared with four of the commercial systems previously discussed; and execution time. Finally, we briefly note the implications of running the experiments and compiling performance data on several different architectures.

Compression savings

The total length of the uncompressed benchmark suite is just under three megabytes. Table IV shows the compressed length achieved by Unix `compress`, PKZIP, *StuffIt*, *Compact Pro* and the heterogeneous compression system. The heterogeneous compressor achieved the greatest compression, with a total compressed length of 1828K. This represents an additional savings of 162K (more than eight per cent) over the best commercial system (*Compact Pro v1.32*), and 339K (nearly 16 per cent) over the average. Compressed lengths for the commercial methods ranged from 1990K to 2375K.

Table V compares the percentage savings obtained by our system to the savings obtained by the commercial programs and the heterogeneous system. The last two columns show the difference in per cent saved between the synthesis system and the best and average of the four commercial packages. The best commercial compressor is marked for each of the files. Note that the heterogeneous compressor does better than all commercial programs in 19 of 20 cases and better than three of the four commercial systems in this one case (file 15). The difference in compression for this file is only 0.02 per cent, whereas for all the other files, the heterogeneous compressor has at least a 1.3 per cent improvement over the best

NetApp; Rackspace Exhibit 1014 Page 64

Table II. Files used to compose the test suite and their respective origins

File designation	File name	File type
audio1	cosby.snd	SoundMaster Macintosh audio file
lowrd1	ticker.txt	ASCII characters from stock ticker
lowrd2	exsound	compressed World Builder sound library
lowrd3	huff	compressed Unix executable
lowrd4	appnote.uue	uuencoded text
text1	phrack.txt	English text
text3	techbook.txt	Unix news article
text4	quanta1.txt	English text
text5	attilla.fluff	English text
text6	shadow.fluff	English text
text7	quanta2.txt	English text
execu1	ad	Unix executable
execu2	sh	Unix executable
execu3	blob	Silicon Graphics executable
execu4	zero	Silicon Graphics executable
execu5	network2.exe	IBM PC executable
execu6	hostname	Unix executable
graph1	compmisc.drw	Lotus Freelance line drawing
graph2	compperi.drw	Lotus Freelance line drawing
graph3	computer.drw	Lotus Freelance line drawing
graph4	lowres.mpt	MacPaint file
graph5	3dbar.drw	Lotus Freelance 3-D bar chart
graph6	image.ppm	PPM (high-resolution image) file
graph7	grp4	MacPaint file
objec1	test1.o	Unix object file
objec2	test2.o	Unix object file
objec3	test3.o	Unix object file
source1	table.c	C source code
source2	freeze.c	C source code

commercial compressor. The average of each column appears in the bottom row; note that the 'percent difference' averages are not weighted by file length, as each file is considered a separate experiment.

Because the quality of compression by the synthesis system depends on that of the algorithms and heuristics used, improvement of the implementations that we use should yield higher performance. This is evidenced by comparing the results of compressing a file dominated by string repetitions by Unix `compress` and *Compact Pro*. Both are implementations of the Lempel–Ziv algorithm. Unix `compress` has no heuristics, whereas *Compact Pro* is a better implementation of LZ77.[5, 11] *Compact Pro* consistently outperforms `compress`. It should be noted that the performance of the *Freeze* variant of Lempel–Ziv[8] used in our sys-

Table III. Combinations of the test files and the resultant simulated data types

File number	File composition	Classification of data modeled
1	text1 — lowrd1	news or stock report
2	graph7 — objec1	object file for a graphics viewer
3	lowrd1 — text3 — graph4	multimedia application (text/graphics)
4	graph7 — execu3	graphics viewer
5	audio1 — graph1	multimedia data file (sound/graphics)
6	text2 — lowrd1 — graph3	multimedia data file (text/graphics)
7	lowrd3 — execu1	commercial utility
8	graph2 — lowrd2 — execu2	multimedia application (graphics/sound/executable)
9	source1 — lowrd3 — graph6	multimedia data or source file (source/compressed binary/image)
10	audio1 — text4	multimedia data file (sound/text)
11	lowrd1 — execu4	statistical application with data
12	graph7 — text5	multimedia data file (text/graphics)
13	lowrd2 — text6	multimedia data file (sound/text)
14	text3 — audio1 — graph5	multimedia data file (text/sound/graphics)
15	lowrd1 — text4 — source2	source file for multimedia program (text/source code)
16	text7 — lowrd2 — graph3	multimedia data file (text/compressed audio/graphics)
17	graph4 — audio1 — execu5	multimedia application (sound/graphics)
18	execu4 — graph7 — text4	multimedia application (graphics/text)
19	objec3 — lowrd3 — execu6	commercial utility
20	objec2 — audio1 — execu2	audio application

tem does consistently better than `compress` and is comparable to *Compact Pro* on standard industrial benchmarks.[9] Improving algorithms and adding or substituting new heuristics would also yield more savings.

Execution times and speed optimizations

In this section we compare, in *approximate units*, the running time of the heterogeneous compressor against those of the four commercial systems the savings rates of which for our test files are documented above. The units are approximate for two reasons. First, because the four test systems are commercial the source code for three of them is not publicly available*, which renders an exact measure of *user* time infeasible. This concern is in part assuaged by the non-multitasked, single-user nature of the microcomputer operating systems on which three (`compress` for Linux notwithstanding) of the commercial systems reside. Second, however, the drastic architectural and organizational differences among the various native machines renders uniform comparisons unreliable. This applies even to normalized execution times because the host machines differ not merely in clock cycle speed, but in instruction set architecture and dynamic instruction frequencies for similar compression algorithms. The *exact* running times reported in this section is only that of the heterogeneous

* As noted, however, the Lempel–Ziv implementation employed by *StuffIt Classic* is nearly identical to that of Unix `compress`.

NetApp; Rackspace Exhibit 1014 Page 66

compressor. These comprise the non-commercial* compression systems for which source code is available for profiling. For the commercial systems we report the observed wall clock time to provide a standard of comparison, but note that the host machines vary in computational power.

Table IV. Results of the four popular commercial programs and the heterogeneous compression system, applied to the 20 test files

File number	Original length	Unix `compress`	PKZIP v1.10	*StuffIt Classic*	*Compact Pro* v1.32	Heterogeneous compressor
1	39,348	20,578	17,119	20,575	16,831	16,315
2	44,202	44,202	39,813	40,412	41,112	37,388
3	46,629	46,629	46,629	43,261	40,367	36,477
4	59,254	52,076	40,571	45,202	41,607	38,007
5	169,108	168,903	151,478	149,701	148,917	134,524
6	100,476	69,771	53,043	65,417	52,349	50,906
7	131,663	131,663	103,544	106,643	109,979	96,429
8	220,644	190,971	137,886	173,677	137,401	127,384
9	301,805	145,993	112,503	137,685	115,096	103,730
10	255,306	204,457	191,378	206,193	183,313	168,675
11	59,305	30,178	22,782	29,701	22,858	21,774
12	51,715	51,715	43,032	46,462	44,107	40,229
13	63,189	63,189	58,247	59,569	59,934	54,481
14	196,789	176,276	196,789	172,486	151,057	137,052
15	148,908	73,555	63,748	75,595	64,618	63,778
16	164,535	141,067	132,992	135,245	110,093	104,175
17	203,912	203,912	184,657	189,398	202,821	170,564
18	200,640	128,675	107,728	125,461	104,711	101,674
19	366,557	265,114	198,727	265,027	198,756	187,659
20	278,152	223,277	193,980	224,943	191,763	181,030
Total	3,102,137	2,432,201	2,096,646	2,312,653	2,037,690	1,872,251

The running times for the commercial systems on the entire test suite documented above appear in Table VI. All of the execution times are measured in wall clock units except for the heterogeneous compressor's, which is a total of user times as reported by `prof`, the C profiler under Unix. The wall clock time was empirically observed not to differ noticeably from this total on an unloaded Unix machine. The commercial systems were similarly tested on unloaded (or single-task) systems.

For Unix `compress`, the mean running time was 26 s, where the average was taken over runs on different Sun workstations of comparable power (documented below). A Unix implementation of PKZIP was also tested on one of these Sun workstations, and achieved an execution time of 56 s – only slightly better than the personal computer version. The running time of 856 s placed the heterogeneous compressor in the middle to high end of the commercial compressors in terms of running time.

* For this purpose we continue to consider Unix `compress` commercial, due to its wide range of versions.

Table V. Percent savings for the test compression systems*

File number	Unix `compress` (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

NetApp; Rackspace Exhibit 1014 Page 68

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix `compress`	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix `compress`, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

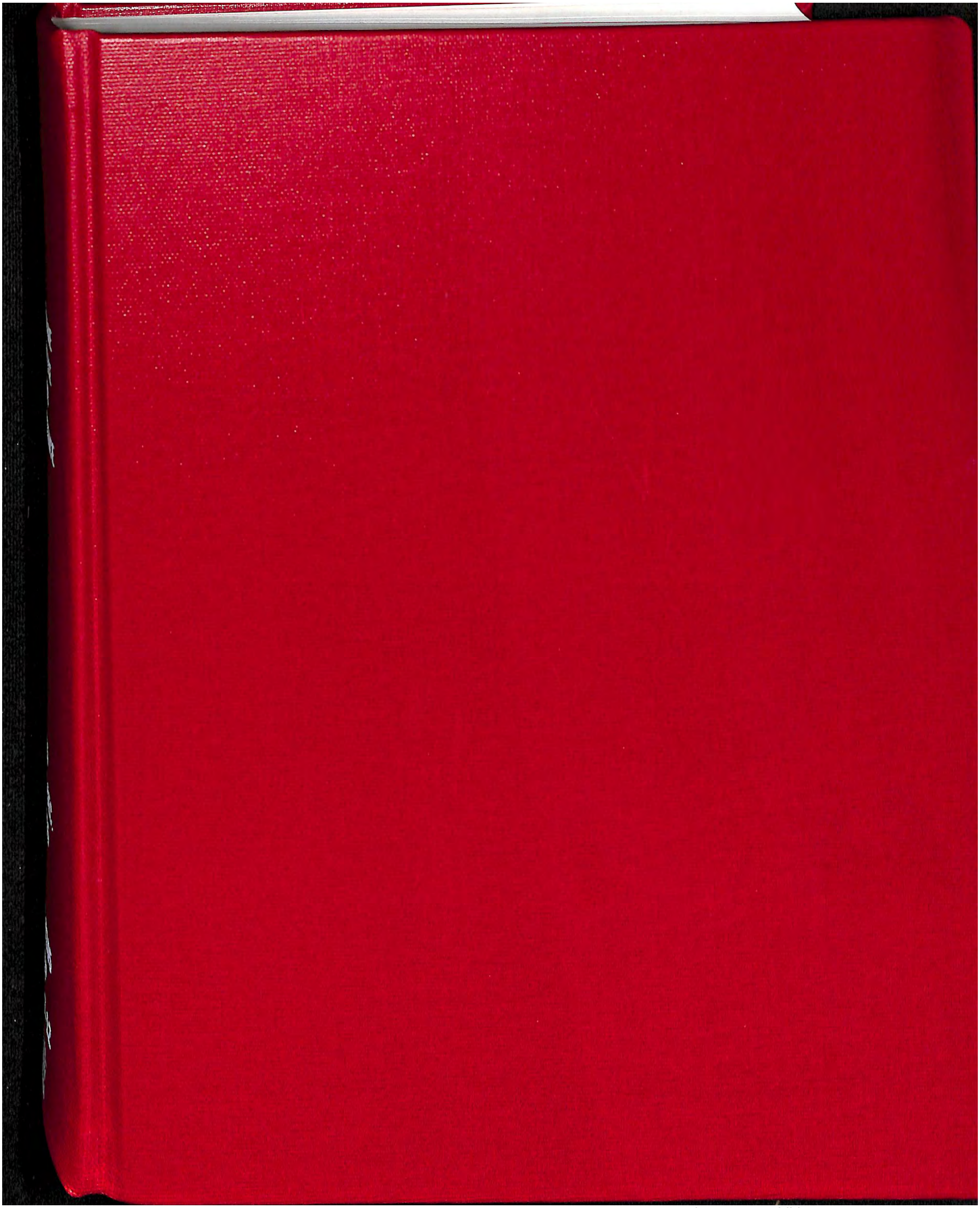
NetApp; Rackspace Exhibit 1014 Page 69

We would like to thank Leonid Broukhis, Graham Toal, and Kenneth Zeger for discussions on some of the research reported here. We also thank Jonathan Eifrig, Bill Goodman, and Tom Lane for guidance on several technical issues. Finally, we thank the anonymous reviewers for their comments and suggestions, especially for introduction to relevant literature in arithmetic coding.

REFERENCES

1. James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.
2. Phillip W. Katz, PKZIP. Commercial compression system, version 1.1, 1990.
3. Sun Microsystems, compress. Commercial compression system, operating system version 5.3, September 1992.
4. Raymond Lau, StuffIt Classic and StuffIt Deluxe. Commercial compression system, 1990.
5. Bill Goodman, Compact Pro. Commercial compression system, v1.32, 1991.
6. Terry A. Welch, 'A technique for high performance data compression', *IEEE Computer*, **17**(6), 8–19 (1984).
7. Gilbert Held and Thomas R. Marshall, *Data Compression: Techniques and Applications: Hardware and Software Considerations*, 3rd edn, John Wiley and Sons, 1991.
8. Leonid Broukhis, Freeze implementation of LZHuf algorithm. comp.sources.misc archives, Internet, 1991.
9. Jean-Loup Gailly, comp.compression benchmark (Calgary test corpus). In comp.compression FAQ list, J. Gailly, (ed.), 1992.
10. Jeffrey S. Vitter, 'Dynamic Huffman Coding', *ACM Transactions on Mathematical Software*, (June 1989).
11. J. Ziv and A. Lempel, 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory*, **23**,(3), 337–343 (1977).
12. J. Ziv and A. Lempel, 'Compression of individual sequences via variable-rate coding', *IEEE Transactions on Information Theory*, **24**(5), 530–546 (1978).
13. Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan and Victor K. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, 320–330 (April 1986).
14. Yooichi Tagawa, Haruhiko Okumura and Haruyasu Yoshizaki, LZHuf: encoding/decoding module for LHarc. Compression system, version 0.03 (Beta), 1989.
15. Haruyasu Yoshizaki, LHA: A high-performance file-compression program. Compression system, version 2.11, 1991.
16. Edward R. Fiala and Daniel H. Greene, 'Data compression with finite windows', *Communications of the ACM*, 490–505 (1989).
17. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, Maryland, second edition, 1987.
18. Graham Toal. Personal communication. Unpublished, 1992.
19. Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
20. Ian F. Darwin, file (program). Berkeley Unix operating system, 1987.
21. David A. Huffman, 'A method for the construction of minimum-redundancy codes', *Proceedings of the IRE*, number 40, 1952, pp. 1098–1101.
22. Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communications*, University of Illinois Press, Urbana and Chicago, 1963.
23. Robert Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
24. Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
25. Sheldon Ross, *A First Course in Probability*, Macmillan Publishing Company, New York, third edition, 1988.
26. Ian H. Witten, Radford Neal and John G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**(6), 520–540 (1987).
27. Independent JPEG Group. 'JPEG image compression system', think.com FTP archives, Internet, 1994.
28. Jean-Loup Gailly. comp.compression/comp.compression.research FAQ list. J. Gailly (ed.), URL `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html`, 1994.
29. James A. Storer, *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
30. Graham Toal. C implementation of dynamic Huffman compressor by J. S. Vitter. comp.source.misc archives, Internet, 1990.

NetApp; Rackspace Exhibit 1014 Page 70



OAK ST. HDSF

UNIVERSITY OF ILLINOIS
LIBRARY
AT URBANA-CHAMPAIGN

SI99205
SOP

JUL 2 6 1995
C62-198

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 7

JULY 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY
Publishers Since 1807
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley-Interscience Publication

SPEXBL 25(7) 705–830 (1995)
ISSN 0038-0644

519.4205
SOF
AUG 2 2 1995
EDK

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 8

AUGUST 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY
Publishers Since 1807
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley–Interscience Publication

SPEXBL 25(8) 831–946 (1995)
ISSN 0038–0644

519.1205 SOF ENX SEP 2 1995

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 9

SEPTEMBER 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY

Publishers Since 1807

Chichester · New York · Brisbane · Toronto · Singapore

A Wiley-Interscience Publication

SPEXBL 25(9) 947–1064 (1995)
ISSN 0038-0644

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 10

OCTOBER 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY

Publishers Since 1807

Chichester · New York · Brisbane · Toronto · Singapore

A Wiley-Interscience Publication

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038-0644

SOFTWARE
PRACTICE & EXPERIENCE

Editors

Professor D. E. Comer, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Charlotte I. Tubis, U.S. Editorial Assistant, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Dr A. J. Wellings, Department of Computer Science, University of York, Heslington, York YO1 5DD

Advisory Editorial Board

Professor D. W. BARRON
Department of Electronics and Computer Science,
University of Southampton,
Southampton SO9 5NH, U.K.

Professor P. J. BROWN
Computing Laboratory, The University,
Canterbury, Kent CT2 7NF, U.K.

Professor J. A. CAMPBELL
Department of Computer Science, University College London,
Gower Street, London WC1E 6BT, U.K.

Professor F. J. CORBATO
Electrical Engineering Department,
Massachusetts Institute of Technology,
545 Technology Square,
Cambridge, Massachusetts 02139, U.S.A.

Dr. Christopher W. FRASER
AT&T Bell Laboratories, 600 Mountain Ave 2C-464,
Murray Hill, NJ 07974-0636, U.S.A.

Professor PER BRINCH HANSEN
School of Computer and Information Science,
4-116 CST, Syracuse University,
Syracuse, New York 13210, U.S.A.

Professor D. R. HANSON
Department of Computer Science,
Princeton University, Princeton,
New Jersey 08544, U.S.A.

Professor J. KATZENELSON
Faculty of Electrical Engineering,
Technion-Israel Institute of Technology,
Haifa, Israel

Dr. B. W. KERNIGHAN
AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, New Jersey 07974, U.S.A.

Professor D. E. KNUTH
Department of Computer Science, Stanford University,
Stanford, California 94305, U.S.A.

Dr. B. W. LAMPSON
180 Lake View Ave,
Cambridge,
MA 02138, U.S.A.

Dr. C. A. LANG
Three Space Ltd,
70 Castle Street,
Cambridge CB3 0AJ, U.K.

Professor B. RANDELL
Computing Laboratory,
University of Newcastle upon Tyne,
Claremont Tower, Claremont Road,
Newcastle-upon-Tyne NE1 7RU, U.K.

Professor J. S. ROHL
Department of Computer Science,
The University of Western Australia,
Nedlands, Western Australia 6009.

D. T. ROSS
Softech Inc., 460 Totten Pond Road,
Waltham, Massachusetts 02154, U.S.A.

B. H. SHEARING
The Software Factory,
28 Padbrook, Limpsfield, Oxted,
Surrey RH8 0DW, U.K.

Professor N. WIRTH
Institut für Computersysteme, ETH-Zentrum,
CH-8092 Zürich, Switzerland.

Aims and Scope

Software—Practice and Experience is an internationally respected and rigorously refereed vehicle for the dissemination and discussion of practical experience with new and established software for both systems and applications. Contributions regularly: (a) describe detailed accounts of completed software-system projects which can serve as 'how-to-do-it' models for future work in the same field; (b) present short reports on programming techniques that can be used in a wide variety of areas; (c) document new techniques and tools that aid in solving software construction problems; and (d) explain methods/techniques that cope with the special demands of large scale software projects. The journal also features timely Short Communications on rapidly developing new topics.

The editors actively encourage papers which result from practical experience with tools and methods developed and used in both academic and industrial environments. The aim is to encourage practitioners to share their experiences with design, implementation and evaluation of techniques and tools for software and software systems.

Papers cover software design and implementation, case studies describing the evolution of system and the thinking behind them, and critical appraisals of software systems. The journal has always welcomed tutorial articles describing well-tried techniques not previously documented in computing literature. The emphasis is on practical experience; articles with theoretical or mathematical content are included only in cases where an understanding of the theory will lead to better practical systems.

Articles range in length from a Short Communication (half to two pages) to the length required to give full treatment to a substantial piece of software (40 or more pages).

Advertising: For details contact—
Michael J. Levermore, Advertisement Sales, John Wiley & Sons Ltd, Baffins Lane, Chichester, Sussex PO19 1UD, England (Telephone 01243 770351, Fax 01243 775878, Telex 86290)

Software—Practice and Experience (ISSN 0038–0644/USPS 890-920) is published monthly, by John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, England. Second class postage paid at Jamaica, N.Y. 11431. Air freight and mailing in the U.S.A. by Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003. © 1995 by John Wiley & Sons Ltd. Printed and bound in Great Britain by Page Bros. Norwich. Printed on acid-free paper.

To subscribe: Orders should be addressed to Subscriptions Department, John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, PO19 1UD, England. 1995 subscription price (13 issues): U.S. $825.00.

U.S.A. POSTMASTER: Send address changes to *Software—Practice and Experience*, c/o Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003, U.S.A.

SOFTWARE—PRACTICE AND EXPERIENCE
(Softw. pract. exp.)

CONTENTS

VOLUME 25, ISSUE No. 10 *October 1995*

Migration in Object-oriented Database Systems—A Practical Approach: C. Huemer, G. Kappel and S. Vieweg 1065

Automatic Synthesis of Compression Techniques for Heterogeneous Files: W. H. Hsu and A. E. Zwarico 1097

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System: P. Ashton and J. Penny 1117

Process Scheduling and UNIX Semaphores: N. Dunstan and I. Fris 1141

Software Maintenance: An Approach to Impact Analysis of Objects Change: S. Ajila 1155

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

Indexed or abstracted by Cambridge Scientific Abstracts, CompuMath Citation Index (ISI), Compuscience Database, Computer Contents, Computer Literature Index, Computing Reviews, Current Contents/Eng, Tech & Applied Sciences, Data Processing Digest, Deadline Newsletter, Educational Technology Abstracts, Engineering Index, Engineering Societies Library, IBZ (International Bibliography of Periodical Literature), Information Science Abstracts (Plenum), INSPEC, Knowledge Engineering Review, Nat Centre for Software Technology, Research Alert (ISI) and SCISEARCH Database (ISI).

20. A. Scheer and A. Hars, 'The Leitstand—a new tool for decentral production control', in G. Fandel and G. Zäpfel (eds), *Modern Production Concepts*, Springer, Berlin, 1991, pp. 370–385.
21. G. Kappel and S. Vieweg, 'Database requirements for CIM applications', *Journal of Integrated Manufacturing Systems*, **5**, (4/5), 48–63 (1994).
22. U. Schreier, 'Database requirements of knowledge-based production scheduling and control: a CIM perspective', in R. Agrawal (ed.), *Proc. 19th International Conference on Very Large Data Bases*, Dublin, August 1993, pp. 710–711.
23. R. Cattell and J. Skeen, 'Object operations benchmark', *ACM Transactions on Database Systems*, **17**, (1), 1–31 (1992).
24. M. Carey, D. DeWitt and J. Naughton, 'The 007 benchmark', *Proc ACM SIGMOD Conf. ACM SIGMOD Record*, **22**, (2), 12–21 (1993).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1097–1116 (OCTOBER 1995)

Automatic Synthesis of Compression Techniques for Heterogeneous Files

WILLIAM H. HSU
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
(email: bhsu@cs.uiuc.edu, voice: (217) 244-1620)

AND

AMY E. ZWARICO
Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.
(email: amy@cs.jhu.edu, voice: (410) 516-5304)

SUMMARY

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix `compress`, *Stuffit*, and *Compact Pro*, and show that our system provides better space savings.

KEY WORDS: adaptive/selective data compression algorithms; redundancy metrics; heterogeneous files; program synthesis

INTRODUCTION

The primary motivation in studying compression is the savings in space that it provides. Many compression algorithms have been implemented, and with the advent of new hardware standards, more techniques are under development. Historically, research in data compression has been devoted to the development of algorithms that exploit various types of redundancy found in a file. The shortcoming of such algorithms is that they assume, often inaccurately, that files are homogeneous throughout. Consequently, each exploits only a subset of the redundancy found in the file.

Unfortunately, no algorithm is effective in compressing all files.[1] For example, dynamic Huffman coding works best on data files with a high variance in the frequency of individual characters (including some graphics and audio data), achieves mediocre performance on natural language text files, and performs poorly in general on high-redundancy binary data. On the other hand, run length encoding works well on high-redundancy binary data, but performs very poorly on text files. Textual substitution works best when multiple-character strings tend to be repeated, as in English text, but this performance degrades as the average

CCC 0038–0644/95/101097–20
©1995 by John Wiley & Sons, Ltd.
Received 20 April 1994
Revised 5 February 1995

length of these strings decreases. These relative strengths and weaknesses become critical when attempting to compress *heterogeneous* files. Heterogeneous files are those which contain multiple types of data such as text, images, binary, audio, or animation. Consequently, their constituent parts may have different degrees of compressibility. Because most compression algorithms are either tailored to a few specific classes of data or are designed to handle a single type of data at a time, they are not suited to the compression of heterogeneous files. In attempting to apply a single method to such files, they forfeit the possibility of greater savings achievable by compressing various segments of the file with different methods.

To overcome this inherent weakness found in compression algorithms, we have developed a *heterogeneous compressor* that automatically chooses the best compression algorithm to use on a given variable-length block of a file, based on both the qualitative and quantitative properties of that segment. The compressor determines and then applies the selected algorithms to the blocks separately. Assembling compression procedures to create a specifically tailored program for each file gives improved performance over using one program for all files. This system produces better compression results than four commonly available compression packages, PKZIP,[2] Unix `compress`,[3] *StuffIt*,[4] and *Compact Pro*[5] for arbitrary heterogeneous files.

The major contributions of this work are twofold. The first is an improved compression system for heterogeneous files. The second is the development of a method of statistical analysis of the compressibility of a file (its redundancy types). Although the concept of redundancy types is not new,[6,7] synthesis of compression techniques using redundancy measurements is largely unprecedented. The approach presented in this paper uses a straightforward program synthesis technique: a *compression plan*, consisting of instructions for each block of input data, is generated, guided by the statistical properties of the input data. Because of its use of algorithms specifically suited to the types of redundancy exhibited by the particular input file, the system achieves consistent average performance throughout the file, as shown by experimental evidence.

As an example of the type of savings our system produces, consider compressing a heterogeneous file (such as a small multimedia data file) consisting of 10K of low redundancy (non-natural language) ASCII data, 10K of English text, and 25K of graphics. In this case, a reasonably sophisticated compression program might recognize the increased savings achievable by employing Huffman compression, to better take advantage of the fact that the majority of the data is graphical. However, none of the general-purpose compression methods under consideration are optimal when used alone on this file. This is because the text part of this file is best compressed by textual substitution methods (e.g., Lempel–Ziv) rather than statistical methods, while the low-redundancy data* and graphics parts are best compressed by alphabetic distribution-based methods (e.g., arithmetic or dynamic Huffman coding) rather than Lempel–Ziv or run-length encoding. This particular file totals 45K in length before compression. A compressor using pure dynamic Huffman coding only achieves about 7 per cent savings for a compressed file of length 42.2K. One of the best general-purpose Lempel–Ziv compressors currently available[8,9] achieves 18 per cent savings, producing a compressed file of length 37.4K. Our system uses arithmetic coding on the first and last segments and Lempel–Ziv compression on the text segment in the middle, achieving a 22 per cent savings and producing a compressed file of length 35.6K. This is a 4 per cent improvement over the best commercial system.

* This denotes, in our system, a file with a low rate of repeated strings.

The purpose of our experiments was to verify the conjecture that a selective system for combining methods can improve savings on a significant range of heterogeneous files, especially multimedia data. This combination differs from current adaptive methods in that it switches among compression paradigms designed to remove very different types of redundancy. By contrast, existing adaptive compression programs are sensitive only to changes in particular types of redundancy, such as run-length, which do not require changing the underlying algorithm used in compression. Thus they cannot adapt to changes in the type of redundancy present, such as from high run-length to high character repetition. The superiority of our approach is demonstrated in our experimental section.

This paper begins with a presentation of existing approaches to data compression, including a discussion of hybrid and adaptive compression algorithms and a description of four popular commercial compression packages. These are followed by documentation on the design of the heterogeneous compression system, analysis of experimental results obtained from test runs of the completed system, and comparison of the system's performance against that of commercial systems. Finally, implications of the results and possibilities for future work are presented.

RELATED WORK

It is a fundamental result of information theory that there is no single algorithm that performs optimally in compressing all files.[1] However, much work has been done to develop algorithms and techniques that work nearly optimally on all classes of files. In this section we discuss adaptive algorithms, composite algorithms, and four popular commercial compression packages.

Adaptive compression algorithms and composite techniques

Exploiting the heterogeneity in a file has been addressed in two ways: the development of *adaptive* compression algorithms, and the composition of various algorithms. Adaptive compression algorithms attune themselves gradually to changes in the redundancies within a file by modifying parameters used by the algorithm, such as the dictionary, during execution.

For example, adaptive alphabetic distribution-based algorithms such as dynamic Huffman coding[10] maintain a tree structure to minimize the encoded length of the most frequently occurring characters. This property can be made to change continuously as a file is processed.

An example of an adaptive textual substitution algorithm is Lempel–Ziv compression, a title which refers to two distinct variants of a basic textual substitution scheme. The first variant, known as LZ77 or the *sliding dictionary* or *sliding window* method, selects positional references from a constant range of preceding strings.[1, 11] These 'back-pointers' literally encode position and length of a repeated string. The second variant, known as LZ78 or the *dynamic dictionary* method, uses a dictionary structure with a *paging* heuristic. When the dictionary – a table of strings from the file – is completely filled, the contents are not discarded. Instead, an auxiliary dictionary is created and updated while compression continues using the main dictionary. Each time this auxiliary table is filled, its contents are 'swapped' into the main dictionary and it is cleared. The maintenance of dictionaries for textual substitution is analogous to the semi-space method of garbage collection, in which two pages are used but only one is 'active' – these are exchanged when one fills beyond a preset threshold. Another adaptive variant of this algorithm is the Lempel–Ziv–Welch

(LZW) algorithm, a descendant of LZ78 used in Unix `compress`.[6, 12] Both LZW and LZ78 vary the length of strings used in compression.[6, 12]

Yet another adaptive (alphabetic distribution-based) compression scheme, the Move-To-Front (MTF) method, was developed by Bentley *et al.*[13] In MTF, the 'word code' for a symbol is determined by the position of the word in a sequential list. The word list is ordered so that frequently accessed words are near the front, thus shortening their encodings.

Adaptive compression algorithms are not appropriate to use with heterogeneous files because they are sensitive only to changes in the particular redundancy type with which they are associated, such as a change in the alphabetic distribution. They do not exploit changes across different redundancy types in the files. Therefore a so-called adaptive method typically cannot directly handle drastic changes in file properties, such as an abrupt transition from text to graphics. For example, adaptive Huffman compressors specially optimized for text achieve disproportionately poor performance on certain image files, and vice versa. As the use of multimedia files increases, files exhibiting this sort of transition will become more prevalent.

Our approach differs from adaptive compression because the system chooses each algorithm (as well as the duration of its applicability) before compression begins, rather than modifying the technique for each file during compression. In addition, while adaptive methods make modifications to their compression parameters on the basis of single bytes or fixed length strings of input, our heterogeneous compressor bases its compression upon statistics gathered from larger blocks of five kilobytes. This allows us to handle much larger changes in file redundancy types. This makes our system less sensitive to residual statistical fluctuations from different parts of a file. We note that it is possible to use an adaptive algorithm as a primitive in the system.

Another approach to handling heterogeneous files is the composition of compression algorithms. Composition can either be accomplished by running several algorithms in succession or by combining the basic algorithms and heuristics to create a new technique. For example, recent implementations of 'universal' compression programs execute the Lempel–Ziv algorithm and dynamic Huffman coding in succession, thus improving performance by combining the string repetition-based compression of Lempel–Ziv with the frequency-based compression strategy of dynamic Huffman coding. One commercial implementation is *LHarc*.[14,15] Our system exploits the same savings since it uses the *Freeze* implementation of the Lempel–Ziv algorithm, which filters Lempel–Ziv compressed output through a Huffman coder. An example of a truly composite technique is the compression achieved by using Shannon–Fano tries* in conjunction with the Fiala–Greene algorithm (a variant of Lempel–Ziv)[16] in the PKZIP[2] commercial package. Tries are used to optimally encode strings by character frequency.[17] PKZIP was selected as the representative test program from this group in our experiment due to its superior performance on industrial benchmarks.[9]

Our approach generalizes the ideas of successively executing or combining different compression algorithms by allowing any combination of basic algorithms within a file. This includes switching from among algorithms an arbitrary number of times within a file. The algorithms themselves may be simple or composite and may be adaptive. All are treated as atomic commands to be applied to portions of a file.

* A *trie* is a tree of variable degree ≥ 2 such that (1) each edge is labelled with a character, and the depth of any node represents one more than the number of characters required to identify it; (2) all internal nodes are intermediate and represent prefixes of keys in the trie; (3) keys (strings) may be inserted as leaves using the minimum number of characters which distinguish them uniquely. Thus a generic trie containing the strings *computer* and *compare* would have keys at a depth of five which share a common prefix of length four.

NetApp; Rackspace Exhibit 1014 Page 83

The problem of heterogeneous files was addressed by Toal[18] in a proposal for a naive heterogeneous compression system similar to ours. In such a system, files would be segmented into fixed-length encapsulated blocks; the optimal algorithm would be selected for each block on the basis of their simple taxonomy (qualitative data type) only; and the blocks would be *independently* compressed. Our system, however, performs more in-depth statistical analysis in order to make a more informed selection from the database of algorithms. This entails not only the determination of qualitative data properties but the computation of metrics for an entire block (as opposed to sporadic or random sampling from parts of each block). Furthermore, normalization constants for selection parameters (i.e. the redundancy metrics) are fitted to observed parameters for a test library. Finally, a straightforward but crucial improvement to the naive encapsulated-block method is the implementation of a multi-pass scheme. By determining the complete taxonomy (data type and dominant redundancy type) in advance, any number of contiguous blocks which use the same compression method will be treated as a single segment. Toal observed in preliminary experiments that the overhead of changing compression schemes from one block to another dominated the additional savings that resulted from selection of a superior compression method.[18] This overhead is attributable to the fact that blocks compressed independently (even if the same method is used) are essentially separate files and assume the same startup overhead of the compression algorithm used.* We have determined experimentally that merging contiguous blocks whenever possible obviates the large majority of changes in compression method. This eliminates a sufficient proportion of the overhead to make heterogeneous compression worthwhile.

Commercial products

One of the goals of this research was to develop a compression system which is generally superior to commercially available systems. The four systems we studied are PKZIP, developed for microcomputers running MS-DOS;[2] Unix `compress`;[3] and *StuffIt Classic*[4] and *Compact Pro*,[5] developed for the Apple Macintosh operating system. Each of these products performs its compression in a single pass, with only one method selected per file. Thus, the possibility of heterogeneous files is ignored.

Unix `compress` uses an adaptive version of the Lempel–Ziv algorithm.[6] It operates by substituting a fixed-length code for common substrings. `compress`, like other adaptive textual substitution algorithms, periodically tests its own performance and reinitializes its string table if the amount of compression has decreased.

StuffIt makes use of two sets of algorithms: it first detects special-type files such as image files and processes them with algorithms suited for high-resolution color data; for the remaining files, it queries the operating system for the explicit file type given when the file was created, and uses this information to choose either the LZW variant of Lempel–Ziv,[4, 6] dynamic Huffman coding, or run-length encoding. This is a much more limited selection process than what we have implemented. Additionally, no selection of compression methods is attempted within a file. *Compact Pro* uses the same methodology as *StuffIt* and `compress`, but incorporates an improved Lempel–Ziv derived directly from LZ77.[11] The public-domain version of *StuffIt* is derived from Unix `compress`, as is evident from the similarity of their performance results.

* For purposes of comparison, the block sizes tested by Toal were nearly identical to those used in our system (ranging upwards from 4K).

NetApp; Rackspace Exhibit 1014 Page 84

Compression systems such as *StuffIt* perform simple selection among alternative compression algorithms. The important problem is that they are underequipped for the task of fitting a specific technique to each file (even when the uncompressed data is homogeneous). *StuffIt* uses few heuristics, since its algorithms are intended to be 'multipurpose' . Furthermore, only the file type is considered in selecting the algorithm – that is, no measures of redundancy are computed. Earlier versions of *StuffIt* (which were extremely similar to Unix `compress`) used composite alphabetic and textual compression, but made no selections on the basis of data characteristics. The chief improvements of our heterogeneous compressor over this approach are that it uses a two-dimensional lookup table, indexed by file properties and quantitative redundancy metrics, and – more important – that it treats the file as a collection of heterogeneous data sets.

THE HETEROGENEOUS COMPRESSOR

Our heterogeneous compressor treats a file as a collection of fixed size blocks (5K in the current implementation), each containing a potentially different type of data and thus best compressed using different algorithms. The actual compression is accomplished in two phases. In the first phase, the system determines the type and compressibility of each block. The compressibility of each block of data is determined by the values of three quantitative metrics representing the alphabetic distribution, the average run length and the string repetition ratio in the file. If these metrics are all below a certain threshold, then the block is considered fully compressed (uncompressible) and the program continues on to the next block. Otherwise, using the block type and largest metric, the appropriate compression algorithm (and possible heuristic) are chosen from the compression algorithm database. The compression method for the current block is then recorded in a small array-based map of the file, and the system continues.

The second phase comprises the actual compression and an optimization that maximizes the size of a segment of data to be compressed using a particular algorithm. In this optimization, which is interleaved with the actual compression, adjacent blocks for which exactly the same method have been chosen are merged into a single block. This merge technique maximizes the length of segments requiring a single compression method by greedily scanning ahead until a change of method is detected. Scanning is performed using the array map of the file generated when compression methods were selected from the database. A compression history, needed for decompression, is automatically generated as part of this phase.

The newly compressed segments are written to a buffer by the algorithm, which stores the output data with the compression history. The system then writes out the compressed file and exits with a signal to the operating system that compression was successful.

From this two-pass scheme it is straightforward to see why this system is less susceptible than traditional adaptive systems to biases accrued when the data type changes abruptly during compression. Adaptive compressors perform all operations myopically, sacrificing the ability to see ahead in the file or data stream to detect future fluctuations in the type of data. As a result, adaptive compressors retain the statistical vestiges of the old method until these are 'flushed out' by new data (or balanced out, depending upon the process for paging and aging internal data structures such as dictionaries). Thus adaptive compressors may continue to suffer the effects of bias, achieving suboptimal compression. On the other hand, by abruptly changing compression algorithms, our technique completely discards all remnants of the 'previous' method (i.e. the algorithm used on the preceding segment). This

NetApp; Rackspace Exhibit 1014 Page 85

allows us to immediately capitalize on changes in data. In addition, merging contiguous blocks of the same data type acquires the advantage of incurring all the overhead *at once* for switching to what will be the best compression method for an entire variable-length segment. The primary advantage of adaptive compression techniques over our technique is that the adaptive compression algorithms are 'online' (single-pass). This property increases compression speed and, more important, gives the ability to compress a data stream (for instance, incoming data packets in a network or modem transmission) in addition to files in secondary storage or variable-length buffers.

The remainder of this section presents the system. We begin with a description of the calculation of the block types and the redundancy metrics. We also explain the use of the metrics as absolute indicators of compressibility, and then describe the compression algorithms used and the structure of the database of algorithms. A discussion of implementation details concludes the section.

Property analysis

The compressibility of a block of data and the appropriate algorithm to do so are determined by the type of data contained in a block and the type of redundancy (if any) in the data. These two properties are represented by four parameters: the *block type*, and the three *redundancy metrics*. The block type describes the data in the block – text, binary, graphical, etc. The three redundancy metrics are the degree of variation in character frequency, average run length in the file, and the string repetition ratio of the file. They provide a quantitative measure of how compressible the block is and which type of redundancy is most evident in the block. The use of both quantitative redundancy measures (redundancy metrics) and qualitative characteristics (block types) as indicators for data compressibility is advocated by Held[7] and Salton.[19] We have refined the process for computing those attributes referred to as *datanalysis results* by Held[7] and as *statistical language characteristics* by Salton[19] to obtain an actual guide for compression. The remainder of this section describes how these four parameters are determined for each block.

Block types

The *block type* describes the nature of a segment of input data. There are ten classifications of data in this system: ANSI text, non-natural language text (hexadecimal encodings of binary data), natural language text, computer source code, low redundancy binary, digitized audio, low resolution graphics, high-resolution graphics, high-redundancy binary executable, and binary object data. ANSI text is composed of characters from a superset of the ASCII alphabet. Non-natural language text contains primarily ASCII text but does not follow a distribution of characters like that of human languages. Examples are computer typesetting data, *uuencoded* and *BinHex* encoded data (which has the same character distribution as binary data but is converted to text for ease of transmission). Natural language text includes text written in English as well as other languages which are representable by the Roman (ASCII) alphabet. Most European languages (including the ones using the Cyrillic alphabet), special symbols excluded, fall into this category, as do the Pinyin and Katakana romanizations of the Chinese and Japanese languages (as opposed to their digital encodings). Computer source code uses the ASCII alphabet but characters are distributed with a different frequency than in natural language text. Low-redundancy binaries usually contain compressed data, but may also include data which is merely difficult to compress. Audio

data are very high in redundancy; audio files (and audio segments of multimedia files) are usually extremely large. Low-resolution graphics have long runs of contiguous repeated bits but unlike high-resolution graphics are not suited to lossy compression. High-resolution graphics include color and grayscale and may be compressed with lossy methods. Binary executables, like low-resolution graphics, have long runs of contiguous repeated bits and comprise all compiled programs on a computer system. Finally, object data has slightly shorter runs but is similarly redundant.

To determine the block type we use a procedure `new-file` which is our extension of the Unix `file` command.[20] `file` works by examining the first 512 bytes of a file and comparing the pattern of data contained in it to a collection of known data patterns from Unix and other operating systems. `new-file` works in a similar fashion, with two modifications. First, it examines and compares not only the first 512 bytes of a data set, but also 512 bytes in the middle of the set and the 512 bytes at the end (if they exist). This provides a better indication of the *primary* data type of a file by taking into account the possibility that the properties may change anywhere within the file. Thus, `new-file` decides on the 'most applicable' data type by a majority vote (or the first data type detected in the case of a three-way tie). The other change is that the known patterns of data have been increased by adding three graphics patterns.

Redundancy metrics

The redundancy metrics are quantitative measures that are used to determine the compressibility of a block of data. They are: the *degree of variation in character frequency* or *alphabetic distribution*, M_{AD}; the *average run length* of the block, M_{RL}; and the *string repetition ratio* of the block, M_{SR}. In general, these three manifestations of redundancy are independent. Each of the redundancy types is exploited by different compression algorithms. *Frequency of characters* is exploited by arithmetic or alphabetic encoding algorithms. In arithmetic coding data is represented by an interval that is calculated from the probability distribution of data. With alphabetic coding algorithms such as the Huffman[21] and Shannon–Fano[22] algorithms, more frequently occurring characters are replaced by shorter units than the less frequently occurring characters. *Contiguous strings*, long strings of identical units occurring next to one another, are exploited by run length encoding algorithms.[23] In these algorithms, contiguous strings are replaced by a single occurrence of the string, called a *run*, plus a count of the number of identical strings following. Both alphabetic distribution and average run length are sometimes characterized as statistical redundancy metrics.[24] *Recurrent strings*, which occur repeatedly in the input stream with any number of interleaved symbols, are exploited by *textual substitution algorithms* such as Lempel–Ziv.[6,11,12] In these algorithms, recurrent instances are replaced with positional references (pointers) to the original instance.

Experimental evidence for the efficacy of quantitative redundancy measures is described in texts by Storer[1] and Shannon.[22] Shannon provided an estimate of the entropy of English text, approximately bounding it to be between one and two bits per character.[22] This was determined experimentally by presenting fragments of (unfamiliar) English text to human subjects and recording the frequency with which they guessed unknown letters. The fragments were revealed character by character, so that letters at the end of long or uncommon words were easiest to guess and letters at the beginnings of words were hardest. The observation that binary executables are known to possess high average run lengths is found in Storer.[1] However, this property is rarely exploited or measured.

NetApp; Rackspace Exhibit 1014 Page 87

Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function G to be a number between 0 and 10 so as to simplify comparison among the different metrics. The gamma distribution was chosen because the graph of each of the unscaled redundancy metrics for a test set of 50 files, when plotted on a histogram, approximated a gamma distribution. Normal and χ^2 distributions were also considered, but these proved to be too specific for the application (since they are both specific parametric cases of the gamma distribution). The gamma distribution is defined as follows (cf Ross[25]):

$$\begin{aligned} G_\tau(x_\tau) &= \int_0^{x_\tau} f_\tau(x)\,\mathrm{d}x \\ f_\tau(x) &= \frac{\lambda_\tau \mathrm{e}^{-\lambda_\tau x}(\lambda_\tau x)^{t_\tau - 1}}{\Gamma(t_\tau)} \\ \Gamma(t_\tau) &= \int_0^{\infty} \mathrm{e}^{-y} y^{t_\tau - 1}\,\mathrm{d}y \end{aligned}$$

where f_τ is the density function, Γ is the gamma function, x_τ is the unnormalized measure, t_τ is the shape parameter for the gamma distribution, and λ_τ is the scale parameter for the gamma distribution. The τ subscript simply represents the redundancy type under consideration, i.e. AD, RL, or SR, respectively. The shape and scaling parameters, t_τ and λ_τ respectively, were determined by fitting the best gamma distribution curve to the data set. This was done by performing the preferred compression method for each file and tabulating the induced ratio among normalized metrics to yield the desired parameter values for each segment. These were then averaged to obtain the empirical scaling parameters.

The *alphabetic distribution metric* (the degree of variation in character frequency) of a block is calculated by taking the population (root-mean-square) standard deviation of the ordinal values of characters in the block and dividing it by the block length (in bytes). The M_{AD} metric is calculated by the following formulas:

$$\begin{aligned} M_{\mathrm{AD}} &= 10 * G_{\mathrm{AD}}(x_{\mathrm{AD}}) \\ x_{\mathrm{AD}} &= \frac{\alpha}{\text{block length in bytes}} \\ \alpha &= \sqrt{\frac{\sum_{c \in \text{charset}} (c-\mu)^2}{256}}, \end{aligned}$$

where c is the ordinal value of a character and μ is the average ordinal value of all characters in a block. The normalization uses $t_{\mathrm{AD}} = 1.70$ and $\lambda_{\mathrm{AD}} = 53.0$ as parameters.

The *average run length metric* is obtained by dividing the number of bits in a block by the number of runs. A run is defined to be a repetition of symbols (either bits or bytes). Our implementation takes both bitwise and bytewise run lengths. For example, if $f = 0001111001110000$ is a file of 16 bits, then the number of bit runs is 5, and the number of byte runs is 2. The scaled metric M_{RL} is obtained by:

$$\begin{aligned} M_{\mathrm{RL}} &= 10 * G_{\mathrm{RL}}(x_{\mathrm{RL}}) \\ x_{\mathrm{RL}} &= \frac{\text{file length in bits}}{\text{number of runs}} \end{aligned}$$

with gamma distribution parameters $t_{RL} = 0.50$ and $\lambda_{RL} = 12.0$.

The *string repetition ratio metric* is the total number of n-bit strings in the block divided by the number of distinct n-bit strings (up to 100K). In our implementation, n is 32, the word size of our machine. The normalized metric M_{SR} is obtained by:

$$M_{SR} = 10 * G_{SR}(x_{SR})$$
$$x_{SR} = \frac{\text{number of } n \text{ bit strings}}{\text{number of distinct } n \text{ bit strings}}$$

with gamma distribution parameters $t_{SR} = 0.18$ and $\lambda_{SR} = 0.2$.

The alphabetic distribution and average run length metrics can be calculated in linear time. The string repetition ratio can be computed in $O(n \log n)$ time using a dictionary data structure. For simplicity, and because a (small) constant amount of data is scanned, we use an $O(n^2)$ version. New strings are stored in an array rather than a binary tree, which would require more insertion overhead (and is not worth while for the 5K block length used in the current system). Our routine integrates $f_\tau(x)$ by Simpson's Rule with $n = 10$ intervals.

The largest of the three metrics is assumed to represent the most significant type of redundancy present in the block. It is expected that compression will decrease at least one of the metrics, and experiments conducted on a wide variety of files have proven this convention to be reliable. Experiments have also shown that if all the normalized metrics are smaller than 2.5, the file is considered not compressible, and the system records a verdict of 'uncompressible' on the current block. If at least one of the parameters is greater than 2.5, the file is considered compressible. The maximum of the normalized metrics is then selected and used in conjunction with the file type to select the appropriate compression algorithm from the lookup table described in the following section. A negative compressibility test does not always imply that all three metrics are below the threshold. In some cases, the only redundancy type for which a metric is above the threshold accesses a null entry in the database of compression algorithms. This is interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm.

The algorithm and heuristic database

The compression algorithms and attendant heuristics are organized into the 10 by 3 table illustrated by Table I. The 10 file descriptors are the row indices and the 3 metrics are the column indices. Each entry of the table contains descriptors which are used to access the code for an algorithm-heuristic pair. It should be noted that four of the entries are blank (indicated by an *). A blank entry indicates that the combination of block type and highest metric are very unusual. In this case, the next highest metric is used instead, provided that it is above the threshold. As an example of using this table, consider a high-redundancy binary executable file whose highest metric is the string repetition metric M_{SR}. Together, this pair indicates that the Lempel–Ziv compression algorithm with the Freeze heuristic will be used.

The algorithms

There are four basic algorithms used by the system: arithmetic coding,[26] Lempel–Ziv,[8] run length encoding (RLE),[23] and JPEG for image/graphics compression.[27]

Arithmetic coding algorithms compress data by representing that data by an interval of

Table I. Database of compression algorithms†

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
hexadecimal	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
natural language	arithmetic coding	*	Lempel–Ziv
	*	*	freeze
source code	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
low redundancy	*	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	*
audio	*	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
low resolution	*	run-length encoding	Lempel–Ziv
graphic	*	n-bit run count	freeze
high resolution	JPEG	run-length encoding	JPEG
color graphic	improved Huffman	n-bit run count	improved Huffman
high redundancy	arithmetic coding	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	freeze
object	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze

† Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

real numbers between zero and one. The width of this interval is inversely proportional to the number of symbols encoded, and the decrease in width is directly proportional to the frequency of the original symbols. Thus the interval specifies the encoded message via its bounds, with the precision (distance) of these bounds reflecting the information content of the message. The end result is that arithmetic coding achieves, in practice, much better space savings than Huffman coding and its dynamic implementations because of its higher likelihood of actually achieving the theoretical lower bound.[24, 28] Although early arithmetic coding algorithms performed too slowly to be of practical use,[29] the implementation of the Witten–Neal–Cleary algorithm used here[26] is optimized for speed – at some cost in space savings, but without giving up its advantage over dynamic Huffman coding. The reader is referred to Bell *et al*[24] for a thorough overview of arithmetic coding. We should note that in earlier implementation of the heterogeneous compressor we used a dynamic Huffman algorithm instead of arithmetic coding. We changed our implementation when we found that then Witten–Neal–Cleary algorithm[26] outperformed our implementation of dynamic Huffman coding[10, 30] in both space savings and execution time.

Run length encoding (RLE) algorithms compress data by replacing contiguous occurrences of a single-unit symbol (either bit or byte) by an efficiently coded count of these runs, usually a single occurrence of the symbol and the number of occurrences. We have implemented a straightforward RLE algorithm for our database, based on the description in Sedgewick.[23] In addition, bitwise and bytewise encoding are available as heuristics and the parameters of bitwise RLE are based on the RL metric.

Files with a high degree of string repetition are compressed using the Lempel–Ziv compression algorithm. It compresses data by replacing frequently occurring strings (with min-

imal regard of how far apart they occur) with compact pointers to the position of the first occurrence. Our implementation is a straightforward array-based encoding with constant-length codes. The algorithm maintains a dictionary of recurring strings in order to do the compression. In our system, the Lempel–Ziv algorithm is augmented with the *Freeze* heuristic. This heuristic suppresses paging of strings in the dictionary after it has been filled; that is, it prevents the replacement of previously encountered strings, regardless of how long ago or how infrequently the string has been encountered. *Freeze* is primarily a speed optimization, since it requires less computation than paging heuristics such as least recently used (LRU) or least frequently used (LFU), but it has been shown to work well for all but the least string-redundant files (including both binary executables and most text files). For files with extremely low string-repetition, our system usually selects Huffman compression.

The compression of high-resolution graphics and audio files uses a *lossy* compression scheme. Appropriately used, lossy algorithms guarantee that the decompressed file is similar enough to the original as to be nearly indistinguishable by human perception, and that repeated compression and decompression leads to limited cumulative 'damage' . The primary benefit of lossy compression is that it guarantees much higher compression ratios at a minimal tradeoff. For instance, a very-high-resolution color image can be compressed with much higher savings (possibly 95 per cent) if the user allows a small amount of noise, always less than 1 per cent per compression, to be introduced during each compression. Our system uses the JPEG system[27] for compressing high-resolution color and grayscale images. JPEG, which is divided into lossy and lossless parts, typically achieves compression ratios of between 15-to-1 and 25-to-1. The *potential* for this substantial savings is obtained by the Discrete Cosine Transform portion of the algorithm, a lossy method. This determines a limit on the amount of savings that can then be achieved by any lossless compressor. The actual savings are realized by a lossless portion, known as the *back end* which is applied to the preprocessed image data. The implementation of this module used in our system[27] is a Huffman coder. It is independent of the lossy front end and can be replaced with a run-length or textual-substitution based algorithm, to be selected by the synthesis system. In our implementation, we chose to retain the original Huffman back end, a different algorithm from the general-purpose dynamic Huffman coder which we also studied.[10, 30] This is because the JPEG Huffman coder is especially suited to the redundancy remaining after lossy preprocessing. It is worthy of mention that the JPEG developers have investigated the use of arithmetic coding back ends, which were found to be experimentally superior but were not used because of proprietary considerations.[27]

Implementation

The system consists of a driver module, four block analysis modules, and the synthesis module, which includes the database of compression algorithms. All modules are written in C and were tested on a Unix platform. The program uses a data directed style of implementation for choosing the compression algorithm to apply to a block. Thus, additional block types, compression algorithms and heuristics, and redundancy metrics can be added to the system with minimal modification of the source code. Only the database would have to be updated and the block analysis routines extended; the rest of the program would remain the same.

The driver performs two iterative passes through the file. It first performs block analysis on the file one 5K block at a time. This block size was chosen after experimentation showed that the response of the system to changes in block type became roughly stable as block

NetApp; Rackspace Exhibit 1014 Page 91

size exceeded 5K (i.e., did not significantly increase as block size did), and that a block size of 5K yielded highly accurate metrics (in only 1 of the 20 test files did the heterogeneous compressor select a suboptimal algorithm for any block). Finally, we found that the highest level of adaptivity without a noticeable *decrease* in accuracy was achieved at 5K, hence our choice of 5K as the block size.

For each block, the system invokes the four analysis modules – three for metric computation and normalization and one to determine the file properties – and stores their output. It then performs the metric comparison and combines the results with the file property to complete the table lookup for the current block. An identifying tag for the selected algorithm is written to the 'compression plan' , an array which stores one complete compression instruction per block (if the current block is deemed uncompressible, a 'skip' instruction is recorded).

We pause here to discuss the normalization of the metrics. Originally, we used a naive normalization method: direct algebraic scaling with experimentally determined constants for each metric. This did not, however, accurately reflect the statistical relationship between variance in character frequency and alphabetic redundancy. Also, the behavior of these functions at asymptotes led to poor approximation of the overall distribution of data segments in the test files. The result was that arithmetic coding was too often incorrectly chosen, resulting in inferior compression; and selection approached randomness as metric values for both string repetition and alphabetic distribution tended toward extreme values. Using the gamma normalization method described above resulted in an improvement in the selection of arithmetic coding. Among the 20 benchmark files, arithmetic coding was selected as the compression method in exactly those cases where the other methods performed worse.

The second pass performs the compression of each block. In order to improve performance, this pass includes a simple optimization step which circumvents the overhead of restarting compression after each fixed length block by merging contiguous blocks that are to be compressed using the same compression algorithms.

On this same pass through the file, the system compresses each of the newly merged blocks using the algorithm recorded in the compression plan. The compressed data is written to an output buffer, while the compressed length (which indicates where in the compressed file a compressed block begins and ends) and compression method are recorded in a separate history for reference at decompression time. If negative compression or no compression is achieved, or if the block was already marked uncompressible, then the data is copied directly to the output buffer (the full block length and a code for 'no compression' are recorded in the compression history). Upon reaching the end of the blocks, the system writes out the compressed data from the output buffers and prepends the encoded compression history to produce the final output file.

When decompression is invoked, the driver module opens the compressed file, interprets the history tag and performs the necessary operations. The tags are a stored version of the compression history in compact, encoded form. Since the heterogeneous system generates different compression sequences for each file, and since the length of a compressed block varies with both the length of the original block and the compression method used, these tags are necessary to guide the decompression process. Currently only the compressed lengths of each block and the method of compression are stored, but a checksum for the original (decompressed) block length can be added with negligible overhead. When executed in reverse order on each compressed block, the instructions in the history tags result in the original file. For simplicity and security, they are prepended to the compressed file (and can easily be encrypted).

EXPERIMENTAL RESULTS

Design and construction of the test files

To test the overall performance, the system was run on a set of 20 test files. These files range in length from approximately 39K to 366K, with representative files from each of the ten block types included in the test corpus.

The test files are designed to model certain types of heterogeneous files, including utilities for image viewing, business, or audio processing, and hypothetical multimedia databases and programs. To construct these files, a collection of 30 files from the Unix, Apple Macintosh, and MS-DOS (IBM PC) operating systems was created. These files are listed in Table II. To create the test corpus, they were concatenated in groups of 2 or 3. The resultant series of test files is listed in Table III. All of the source files were used. The goal was to generate as broad a range of permutations as possible (while restricting the generated files to those which are likely to exist in a typical user environment). This was performed manually with consideration toward combinatorial constraints and the criteria of realistic data modeling. Since all of the files in the source collection originate from common commercial sources or from public archives (with the exception of the source and object files, which are from the code for the heterogeneous compressor itself), the latter constraint was considerably simplified.

The assembled files were then ported to the test sites (a Sun workstation for Unix `compress` and our heterogeneous compressor; a Macintosh for *StuffIt* and *Compact Pro*; and an IBM 80486 machine for PKZIP). Binary file transfer mode was used to ensure that the file lengths agreed exactly among all platforms.

Performance

In this section, we review and analyze the performance of the heterogeneous compressor with respect to compression savings, as compared with four of the commercial systems previously discussed; and execution time. Finally, we briefly note the implications of running the experiments and compiling performance data on several different architectures.

Compression savings

The total length of the uncompressed benchmark suite is just under three megabytes. Table IV shows the compressed length achieved by Unix `compress`, PKZIP, *StuffIt*, *Compact Pro* and the heterogeneous compression system. The heterogeneous compressor achieved the greatest compression, with a total compressed length of 1828K. This represents an additional savings of 162K (more than eight per cent) over the best commercial system (*Compact Pro v1.32*), and 339K (nearly 16 per cent) over the average. Compressed lengths for the commercial methods ranged from 1990K to 2375K.

Table V compares the percentage savings obtained by our system to the savings obtained by the commercial programs and the heterogeneous system. The last two columns show the difference in per cent saved between the synthesis system and the best and average of the four commercial packages. The best commercial compressor is marked for each of the files. Note that the heterogeneous compressor does better than all commercial programs in 19 of 20 cases and better than three of the four commercial systems in this one case (file 15). The difference in compression for this file is only 0.02 per cent, whereas for all the other files, the heterogeneous compressor has at least a 1.3 per cent improvement over the best

NetApp; Rackspace Exhibit 1014 Page 93

Table II. Files used to compose the test suite and their respective origins

File designation	File name	File type
audio1	cosby.snd	SoundMaster Macintosh audio file
lowrd1	ticker.txt	ASCII characters from stock ticker
lowrd2	exsound	compressed World Builder sound library
lowrd3	huff	compressed Unix executable
lowrd4	appnote.uue	uuencoded text
text1	phrack.txt	English text
text3	techbook.txt	Unix news article
text4	quanta1.txt	English text
text5	attilla.fluff	English text
text6	shadow.fluff	English text
text7	quanta2.txt	English text
execu1	ad	Unix executable
execu2	sh	Unix executable
execu3	blob	Silicon Graphics executable
execu4	zero	Silicon Graphics executable
execu5	network2.exe	IBM PC executable
execu6	hostname	Unix executable
graph1	compmisc.drw	Lotus Freelance line drawing
graph2	compperi.drw	Lotus Freelance line drawing
graph3	computer.drw	Lotus Freelance line drawing
graph4	lowres.mpt	MacPaint file
graph5	3dbar.drw	Lotus Freelance 3-D bar chart
graph6	image.ppm	PPM (high-resolution image) file
graph7	grp4	MacPaint file
objec1	test1.o	Unix object file
objec2	test2.o	Unix object file
objec3	test3.o	Unix object file
source1	table.c	C source code
source2	freeze.c	C source code

commercial compressor. The average of each column appears in the bottom row; note that the 'percent difference' averages are not weighted by file length, as each file is considered a separate experiment.

Because the quality of compression by the synthesis system depends on that of the algorithms and heuristics used, improvement of the implementations that we use should yield higher performance. This is evidenced by comparing the results of compressing a file dominated by string repetitions by Unix `compress` and *Compact Pro*. Both are implementations of the Lempel–Ziv algorithm. Unix `compress` has no heuristics, whereas *Compact Pro* is a better implementation of LZ77.[5, 11] *Compact Pro* consistently outperforms `compress`. It should be noted that the performance of the *Freeze* variant of Lempel–Ziv[8] used in our sys-

Table III. Combinations of the test files and the resultant simulated data types

File number	File composition	Classification of data modeled
1	text1 — lowrd1	news or stock report
2	graph7 — objec1	object file for a graphics viewer
3	lowrd1 — text3 — graph4	multimedia application (text/graphics)
4	graph7 — execu3	graphics viewer
5	audio1 — graph1	multimedia data file (sound/graphics)
6	text2 — lowrd1 — graph3	multimedia data file (text/graphics)
7	lowrd3 — execu1	commercial utility
8	graph2 — lowrd2 — execu2	multimedia application (graphics/sound/executable)
9	source1 — lowrd3 — graph6	multimedia data or source file (source/compressed binary/image)
10	audio1 — text4	multimedia data file (sound/text)
11	lowrd1 — execu4	statistical application with data
12	graph7 — text5	multimedia data file (text/graphics)
13	lowrd2 — text6	multimedia data file (sound/text)
14	text3 — audio1 — graph5	multimedia data file (text/sound/graphics)
15	lowrd1 — text4 — source2	source file for multimedia program (text/source code)
16	text7 — lowrd2 — graph3	multimedia data file (text/compressed audio/graphics)
17	graph4 — audio1 — execu5	multimedia application (sound/graphics)
18	execu4 — graph7 — text4	multimedia application (graphics/text)
19	objec3 — lowrd3 — execu6	commercial utility
20	objec2 — audio1 — execu2	audio application

tem does consistently better than `compress` and is comparable to *Compact Pro* on standard industrial benchmarks.[9] Improving algorithms and adding or substituting new heuristics would also yield more savings.

Execution times and speed optimizations

In this section we compare, in *approximate units*, the running time of the heterogeneous compressor against those of the four commercial systems the savings rates of which for our test files are documented above. The units are approximate for two reasons. First, because the four test systems are commercial the source code for three of them is not publicly available*, which renders an exact measure of *user* time infeasible. This concern is in part assuaged by the non-multitasked, single-user nature of the microcomputer operating systems on which three (`compress` for Linux notwithstanding) of the commercial systems reside. Second, however, the drastic architectural and organizational differences among the various native machines renders uniform comparisons unreliable. This applies even to normalized execution times because the host machines differ not merely in clock cycle speed, but in instruction set architecture and dynamic instruction frequencies for similar compression algorithms. The *exact* running times reported in this section is only that of the heterogeneous

* As noted, however, the Lempel–Ziv implementation employed by *StuffIt Classic* is nearly identical to that of Unix `compress`.

compressor. These comprise the non-commercial* compression systems for which source code is available for profiling. For the commercial systems we report the observed wall clock time to provide a standard of comparison, but note that the host machines vary in computational power.

Table IV. Results of the four popular commercial programs and the heterogeneous compression system, applied to the 20 test files

File number	Original length	Unix compress	PKZIP v1.10	*StuffIt Classic*	*Compact Pro* v1.32	Heterogeneous compressor
1	39,348	20,578	17,119	20,575	16,831	16,315
2	44,202	44,202	39,813	40,412	41,112	37,388
3	46,629	46,629	46,629	43,261	40,367	36,477
4	59,254	52,076	40,571	45,202	41,607	38,007
5	169,108	168,903	151,478	149,701	148,917	134,524
6	100,476	69,771	53,043	65,417	52,349	50,906
7	131,663	131,663	103,544	106,643	109,979	96,429
8	220,644	190,971	137,886	173,677	137,401	127,384
9	301,805	145,993	112,503	137,685	115,096	103,730
10	255,306	204,457	191,378	206,193	183,313	168,675
11	59,305	30,178	22,782	29,701	22,858	21,774
12	51,715	51,715	43,032	46,462	44,107	40,229
13	63,189	63,189	58,247	59,569	59,934	54,481
14	196,789	176,276	196,789	172,486	151,057	137,052
15	148,908	73,555	63,748	75,595	64,618	63,778
16	164,535	141,067	132,992	135,245	110,093	104,175
17	203,912	203,912	184,657	189,398	202,821	170,564
18	200,640	128,675	107,728	125,461	104,711	101,674
19	366,557	265,114	198,727	265,027	198,756	187,659
20	278,152	223,277	193,980	224,943	191,763	181,030
Total	3,102,137	2,432,201	2,096,646	2,312,653	2,037,690	1,872,251

The running times for the commercial systems on the entire test suite documented above appear in Table VI. All of the execution times are measured in wall clock units except for the heterogeneous compressor's, which is a total of user times as reported by `prof`, the C profiler under Unix. The wall clock time was empirically observed not to differ noticeably from this total on an unloaded Unix machine. The commercial systems were similarly tested on unloaded (or single-task) systems.

For Unix `compress`, the mean running time was 26 s, where the average was taken over runs on different Sun workstations of comparable power (documented below). A Unix implementation of PKZIP was also tested on one of these Sun workstations, and achieved an execution time of 56 s – only slightly better than the personal computer version. The running time of 856 s placed the heterogeneous compressor in the middle to high end of the commercial compressors in terms of running time.

* For this purpose we continue to consider Unix `compress` commercial, due to its wide range of versions.

Table V. Percent savings for the test compression systems*

File number	Unix compress (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix `compress`	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix `compress`, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

Table V. Percent savings for the test compression systems*

File number	Unix compress (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

NetApp; Rackspace Exhibit 1014 Page 99

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix compress	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix compress, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

We would like to thank Leonid Broukhis, Graham Toal, and Kenneth Zeger for discussions on some of the research reported here. We also thank Jonathan Eifrig, Bill Goodman, and Tom Lane for guidance on several technical issues. Finally, we thank the anonymous reviewers for their comments and suggestions, especially for introduction to relevant literature in arithmetic coding.

REFERENCES

1. James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.
2. Phillip W. Katz, PKZIP. Commercial compression system, version 1.1, 1990.
3. Sun Microsystems, compress. Commercial compression system, operating system version 5.3, September 1992.
4. Raymond Lau, StuffIt Classic and StuffIt Deluxe. Commercial compression system, 1990.
5. Bill Goodman, Compact Pro. Commercial compression system, v1.32, 1991.
6. Terry A. Welch, 'A technique for high performance data compression', *IEEE Computer*, **17**(6), 8–19 (1984).
7. Gilbert Held and Thomas R. Marshall, *Data Compression: Techniques and Applications: Hardware and Software Considerations*, 3rd edn, John Wiley and Sons, 1991.
8. Leonid Broukhis, Freeze implementation of LZHuf algorithm. comp.sources.misc archives, Internet, 1991.
9. Jean-Loup Gailly, comp.compression benchmark (Calgary test corpus). In comp.compression FAQ list, J. Gailly, (ed.), 1992.
10. Jeffrey S. Vitter, 'Dynamic Huffman Coding', *ACM Transactions on Mathematical Software*, (June 1989).
11. J. Ziv and A. Lempel, 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory*, **23**,(3), 337–343 (1977).
12. J. Ziv and A. Lempel, 'Compression of individual sequences via variable-rate coding', *IEEE Transactions on Information Theory*, **24**(5), 530–546 (1978).
13. Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan and Victor K. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, 320–330 (April 1986).
14. Yooichi Tagawa, Haruhiko Okumura and Haruyasu Yoshizaki, LZHuf: encoding/decoding module for LHarc. Compression system, version 0.03 (Beta), 1989.
15. Haruyasu Yoshizaki, LHA: A high-performance file-compression program. Compression system, version 2.11, 1991.
16. Edward R. Fiala and Daniel H. Greene, 'Data compression with finite windows', *Communications of the ACM*, 490–505 (1989).
17. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, Maryland, second edition, 1987.
18. Graham Toal. Personal communication. Unpublished, 1992.
19. Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
20. Ian F. Darwin, file (program). Berkeley Unix operating system, 1987.
21. David A. Huffman, 'A method for the construction of minimum-redundancy codes', *Proceedings of the IRE*, number 40, 1952, pp. 1098–1101.
22. Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communications*, University of Illinois Press, Urbana and Chicago, 1963.
23. Robert Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
24. Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
25. Sheldon Ross, *A First Course in Probability*, Macmillan Publishing Company, New York, third edition, 1988.
26. Ian H. Witten, Radford Neal and John G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**(6), 520–540 (1987).
27. Independent JPEG Group. 'JPEG image compression system', think.com FTP archives, Internet, 1994.
28. Jean-Loup Gailly. comp.compression/comp.compression.research FAQ list. J. Gailly (ed.), URL `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html`, 1994.
29. James A. Storer, *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
30. Graham Toal. C implementation of dynamic Huffman compressor by J. S. Vitter. comp.source.misc archives, Internet, 1990.

NetApp; Rackspace Exhibit 1014 Page 101

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1117–1140 (OCTOBER 1995)

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System

PAUL ASHTON AND JOHN PENNY

Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand
(email: paul@cosc.canterbury.ac.nz)

SUMMARY

The paper describes in detail the implementation of INMON, a prototype monitor for recording and displaying interaction networks. The *interaction network* is designed to show the interrelationships between significant events that occur during the processing of an interaction on a loosely-coupled distributed system. By being directed at analysing what happens in an interaction, this approach is fundamentally different from other graphical representations that show what happens during the execution of a single program on a distributed system. Examples are given of interaction networks recorded by INMON. The approach is based on very general models of a distributed system and of an interaction, and could be widely applied. We conclude by summarizing what is needed to provide facilities within any operating system for recording interaction networks.

KEY WORDS: loosely-coupled distributed systems; interactive performance; visualization tools

INTRODUCTION

Recently, there has been a great deal of interest in visualization techniques for complex systems in science and engineering.[1,2] Visualization techniques have also been developed for computer systems, in particular for distributed and parallel systems. The aim may be to achieve better performance,[3] or simply to get a clearer understanding of what happens in practice during the execution of programs on a complex, modern computer system.[4–6]

In this paper, we present first the idea of the *interaction network*, a directed graph designed to represent the execution of an interaction on a loosely-coupled distributed system. Each interaction network shows the interrelationships between significant events that occur during execution of the interaction. In representing what happens during an *interaction*, the interaction network is significantly different in nature and purpose from other graphical representations[3,6,7] that show what happens during execution of a *program*.

We then discuss in detail the implementation of INMON, a prototype monitor for recording interaction networks for a network of Sun 3 workstations. The implementation required modifications to the kernel of the SunOS operating system, and the finding of a software solution to the problem of clock synchronization. We follow with some examples of interaction networks recorded by INMON, to illustrate what can be discovered from studying interaction networks.

Monitors that collect data to show graphically what happens during execution should, we feel, be built-in to operating system kernels at the time of design. We conclude by

CCC 0038–0644/95/101117–24
©1995 by John Wiley & Sons, Ltd.
Received 1 December 1992
Revised 21 March 1995

NOV 27 1995

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 11

NOVEMBER 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY

Publishers Since 1807

Chichester · New York · Brisbane · Toronto · Singapore

A Wiley–Interscience Publication

SPEXBL 25(11) 1183–1298 (1995)
ISSN 0038-0644

ENX
DEC 14 1995

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 12

DECEMBER 1995

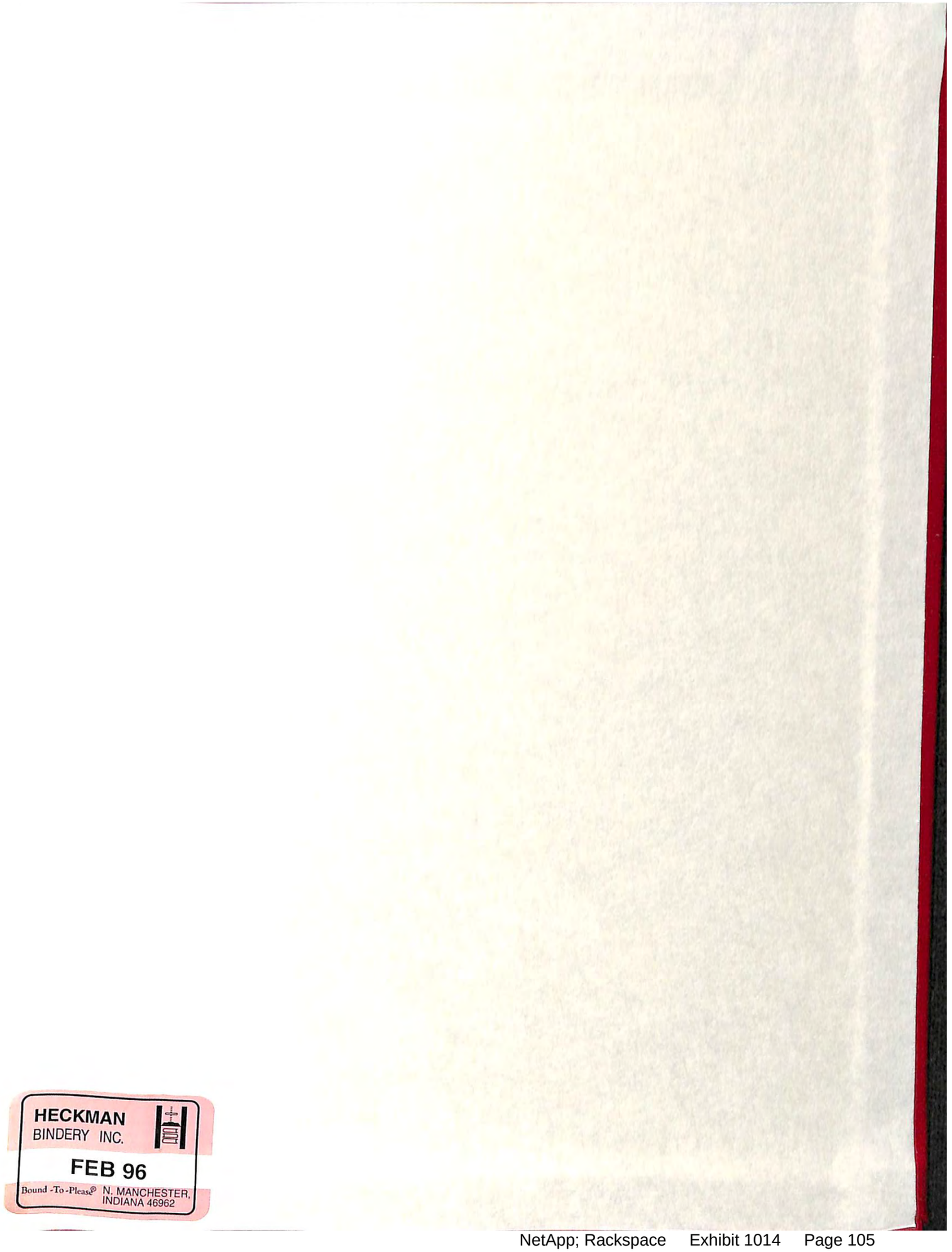
EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY
Publishers Since 1807
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley–Interscience Publication

SPEXBL 25(12) 1299–1408 (1995)
ISSN 0038-0644



HECKMAN
BINDERY INC.
FEB 96
Bound-To-Please® N. MANCHESTER, INDIANA 46962

SOFTWARE
PRACTICE & EXPERIENCE

VOLUME 25, No. 10 OCTOBER 1995

UNIVERSITY OF MINNESOTA
RECEIVED
SEP 25 1995
LIBRARIES

EDITORS

DOUGLAS COMER

ANDY WELLINGS

WILEY
Publishers Since 1807
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley–Interscience Publication

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

NetApp; Rackspace Exhibit 1014 Page 106

SOFTWARE

PRACTICE & EXPERIENCE

Editors

Professor D. E. Comer, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Charlotte I. Tubis, U.S. Editorial Assistant, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Dr A. J. Wellings, Department of Computer Science, University of York, Heslington, York YO1 5DD

Advisory Editorial Board

Professor D. W. BARRON
Department of Electronics and Computer Science,
University of Southampton,
Southampton SO9 5NH, U.K.

Professor P. J. BROWN
Computing Laboratory, The University,
Canterbury, Kent CT2 7NF, U.K.

Professor J. A. CAMPBELL
Department of Computer Science, University College London,
Gower Street, London WC1E 6BT, U.K.

Professor F. J. CORBATO
Electrical Engineering Department,
Massachusetts Institute of Technology,
545 Technology Square,
Cambridge, Massachusetts 02139, U.S.A.

Dr. Christopher W. FRASER
AT&T Bell Laboratories, 600 Mountain Ave 2C-464,
Murray Hill, NJ 07974-0636, U.S.A.

Professor PER BRINCH HANSEN
School of Computer and Information Science,
4-116 CST, Syracuse University,
Syracuse, New York 13210, U.S.A.

Professor D. R. HANSON
Department of Computer Science,
Princeton University, Princeton,
New Jersey 08544, U.S.A.

Professor J. KATZENELSON
Faculty of Electrical Engineering,
Technion-Israel Institute of Technology,
Haifa, Israel

Dr. B. W. KERNIGHAN
AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, New Jersey 07974, U.S.A.

Professor D. E. KNUTH
Department of Computer Science, Stanford University,
Stanford, California 94305, U.S.A.

Dr. B. W. LAMPSON
180 Lake View Ave,
Cambridge,
MA 02138, U.S.A.

Dr. C. A. LANG
Three-Space Ltd,
70 Castle Street,
Cambridge CB3 0AJ, U.K.

Professor B. RANDELL
Computing Laboratory,
University of Newcastle-upon-Tyne,
Claremont Tower, Claremont Road,
Newcastle-upon-Tyne NE1 7RU, U.K.

Professor J. S. ROHL
Department of Computer Science,
The University of Western Australia,
Nedlands, Western Australia 6009.

D. T. ROSS
Softech Inc., 460 Totten Pond Road,
Waltham, Massachusetts 02154, U.S.A.

B. H. SHEARING
The Software Factory,
28 Padbrook, Limpsfield, Oxted,
Surrey RH8 0DW, U.K.

Professor N. WIRTH
Institut für Computersysteme, ETH-Zentrum,
CH-8092 Zürich, Switzerland.

Aims and Scope

Software—Practice and Experience is an internationally respected and rigorously refereed vehicle for the dissemination and discussion of practical experience with new and established software for both systems and applications. Contributions regularly: (a) describe detailed accounts of completed software-system projects which can serve as 'how-to-do-it' models for future work in the same field; (b) present short reports on programming techniques that can be used in a wide variety of areas; (c) document new techniques and tools that aid in solving software construction problems; and (d) explain methods/techniques that cope with the special demands of large scale software projects. The journal also features timely Short Communications on rapidly developing new topics.

The editors actively encourage papers which result from practical experience with tools and methods developed and used in both academic and industrial environments. The aim is to encourage practitioners to share their experiences with design, implementation and evaluation of techniques and tools for software and software systems.

Papers cover software design and implementation, case studies describing the evolution of system and the thinking behind them, and critical appraisals of software systems. The journal has always welcomed tutorial articles describing well-tried techniques not previously documented in computing literature. The emphasis is on practical experience; articles with theoretical or mathematical content are included only in cases where an understanding of the theory will lead to better practical systems.

Articles range in length from a Short Communication (half to two pages) to the length required to give full treatment to a substantial piece of software (40 or more pages).

Advertising: For details contact—
Michael J. Levermore, Advertisement Sales, John Wiley & Sons Ltd, Baffins Lane, Chichester, Sussex PO19 1UD, England (Telephone 01243 770351, Fax 01243 775878, Telex 86290)

Software—Practice and Experience (ISSN 0038-0644/USPS 890-920) is published monthly, by John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, England. Second class postage paid at Jamaica, N.Y. 11431. Air freight and mailing in the U.S.A. by Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003.

© 1995 by John Wiley & Sons Ltd.

Printed and bound in Great Britain by Page Bros, Norwich. Printed on acid-free paper.

To subscribe: Orders should be addressed to Subscriptions Department, John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, PO19 1UD, England. 1995 subscription price (13 issues): U.S. $825.00.

U.S.A. POSTMASTER: Send address changes to *Software—Practice and Experience*, c/o Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003, U.S.A.

SOFTWARE—PRACTICE AND EXPERIENCE
(Softw. pract. exp.)

CONTENTS

VOLUME 25, ISSUE No. 10 *October 1995*

Migration in Object-oriented Database Systems—A Practical Approach: C. Huemer, G. Kappel and S. Vieweg 1065

Automatic Synthesis of Compression Techniques for Heterogeneous Files: W. H. Hsu and A. E. Zwarico 1097

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System: P. Ashton and J. Penny 1117

Process Scheduling and UNIX Semaphores: N. Dunstan and I. Fris 1141

Software Maintenance: An Approach to Impact Analysis of Objects Change: S. Ajila 1155

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

Indexed or abstracted by Cambridge Scientific Abstracts, CompuMath Citation Index (ISI), Compuscience Database, Computer Contents, Computer Literature Index, Computing Reviews, Current Contents/Eng, Tech & Applied Sciences, Data Processing Digest, Deadline Newsletter, Educational Technology Abstracts, Engineering Index, Engineering Societies Library, IBZ (International Bibliography of Periodical Literature), Information Science Abstracts (Plenum), INSPEC, Knowledge Engineering Review, Nat Centre for Software Technology, Research Alert (ISI) and SCISEARCH Database (ISI).

NetApp; Rackspace Exhibit 1014 Page 108

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1097–1116 (OCTOBER 1995)

Automatic Synthesis of Compression Techniques for Heterogeneous Files

WILLIAM H. HSU

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
(email: bhsu@cs.uiuc.edu, voice: (217) 244-1620)

AND

AMY E. ZWARICO

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.
(email: amy@cs.jhu.edu, voice: (410) 516-5304)

SUMMARY

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix `compress`, *StuffIt*, and *Compact Pro*, and show that our system provides better space savings.

KEY WORDS: adaptive/selective data compression algorithms; redundancy metrics; heterogeneous files; program synthesis

INTRODUCTION

The primary motivation in studying compression is the savings in space that it provides. Many compression algorithms have been implemented, and with the advent of new hardware standards, more techniques are under development. Historically, research in data compression has been devoted to the development of algorithms that exploit various types of redundancy found in a file. The shortcoming of such algorithms is that they assume, often inaccurately, that files are homogeneous throughout. Consequently, each exploits only a subset of the redundancy found in the file.

Unfortunately, no algorithm is effective in compressing all files.[1] For example, dynamic Huffman coding works best on data files with a high variance in the frequency of individual characters (including some graphics and audio data), achieves mediocre performance on natural language text files, and performs poorly in general on high-redundancy binary data. On the other hand, run length encoding works well on high-redundancy binary data, but performs very poorly on text files. Textual substitution works best when multiple-character strings tend to be repeated, as in English text, but this performance degrades as the average

CCC 0038–0644/95/101097–20
©1995 by John Wiley & Sons, Ltd.
Received 20 April 1994
Revised 5 February 1995

length of these strings decreases. These relative strengths and weaknesses become critical when attempting to compress *heterogeneous* files. Heterogeneous files are those which contain multiple types of data such as text, images, binary, audio, or animation. Consequently, their constituent parts may have different degrees of compressibility. Because most compression algorithms are either tailored to a few specific classes of data or are designed to handle a single type of data at a time, they are not suited to the compression of heterogeneous files. In attempting to apply a single method to such files, they forfeit the possibility of greater savings achievable by compressing various segments of the file with different methods.

To overcome this inherent weakness found in compression algorithms, we have developed a *heterogeneous compressor* that automatically chooses the best compression algorithm to use on a given variable-length block of a file, based on both the qualitative and quantitative properties of that segment. The compressor determines and then applies the selected algorithms to the blocks separately. Assembling compression procedures to create a specifically tailored program for each file gives improved performance over using one program for all files. This system produces better compression results than four commonly available compression packages, PKZIP,[2] Unix `compress`,[3] *StuffIt*,[4] and *Compact Pro*[5] for arbitrary heterogeneous files.

The major contributions of this work are twofold. The first is an improved compression system for heterogeneous files. The second is the development of a method of statistical analysis of the compressibility of a file (its redundancy types). Although the concept of redundancy types is not new,[6,7] synthesis of compression techniques using redundancy measurements is largely unprecedented. The approach presented in this paper uses a straightforward program synthesis technique: a *compression plan*, consisting of instructions for each block of input data, is generated, guided by the statistical properties of the input data. Because of its use of algorithms specifically suited to the types of redundancy exhibited by the particular input file, the system achieves consistent average performance throughout the file, as shown by experimental evidence.

As an example of the type of savings our system produces, consider compressing a heterogeneous file (such as a small multimedia data file) consisting of 10K of low redundancy (non-natural language) ASCII data, 10K of English text, and 25K of graphics. In this case, a reasonably sophisticated compression program might recognize the increased savings achievable by employing Huffman compression, to better take advantage of the fact that the majority of the data is graphical. However, none of the general-purpose compression methods under consideration are optimal when used alone on this file. This is because the text part of this file is best compressed by textual substitution methods (e.g., Lempel–Ziv) rather than statistical methods, while the low-redundancy data* and graphics parts are best compressed by alphabetic distribution-based methods (e.g., arithmetic or dynamic Huffman coding) rather than Lempel–Ziv or run-length encoding. This particular file totals 45K in length before compression. A compressor using pure dynamic Huffman coding only achieves about 7 per cent savings for a compressed file of length 42.2K. One of the best general-purpose Lempel–Ziv compressors currently available[8,9] achieves 18 per cent savings, producing a compressed file of length 37.4K. Our system uses arithmetic coding on the first and last segments and Lempel–Ziv compression on the text segment in the middle, achieving a 22 per cent savings and producing a compressed file of length 35.6K. This is a 4 per cent improvement over the best commercial system.

* This denotes, in our system, a file with a low rate of repeated strings.

NetApp; Rackspace Exhibit 1014 Page 110

The purpose of our experiments was to verify the conjecture that a selective system for combining methods can improve savings on a significant range of heterogeneous files, especially multimedia data. This combination differs from current adaptive methods in that it switches among compression paradigms designed to remove very different types of redundancy. By contrast, existing adaptive compression programs are sensitive only to changes in particular types of redundancy, such as run-length, which do not require changing the underlying algorithm used in compression. Thus they cannot adapt to changes in the type of redundancy present, such as from high run-length to high character repetition. The superiority of our approach is demonstrated in our experimental section.

This paper begins with a presentation of existing approaches to data compression, including a discussion of hybrid and adaptive compression algorithms and a description of four popular commercial compression packages. These are followed by documentation on the design of the heterogeneous compression system, analysis of experimental results obtained from test runs of the completed system, and comparison of the system's performance against that of commercial systems. Finally, implications of the results and possibilities for future work are presented.

RELATED WORK

It is a fundamental result of information theory that there is no single algorithm that performs optimally in compressing all files.[1] However, much work has been done to develop algorithms and techniques that work nearly optimally on all classes of files. In this section we discuss adaptive algorithms, composite algorithms, and four popular commercial compression packages.

Adaptive compression algorithms and composite techniques

Exploiting the heterogeneity in a file has been addressed in two ways: the development of *adaptive* compression algorithms, and the composition of various algorithms. Adaptive compression algorithms attune themselves gradually to changes in the redundancies within a file by modifying parameters used by the algorithm, such as the dictionary, during execution.

For example, adaptive alphabetic distribution-based algorithms such as dynamic Huffman coding[10] maintain a tree structure to minimize the encoded length of the most frequently occurring characters. This property can be made to change continuously as a file is processed.

An example of an adaptive textual substitution algorithm is Lempel–Ziv compression, a title which refers to two distinct variants of a basic textual substitution scheme. The first variant, known as LZ77 or the *sliding dictionary* or *sliding window* method, selects positional references from a constant range of preceding strings.[1, 11] These 'back-pointers' literally encode position and length of a repeated string. The second variant, known as LZ78 or the *dynamic dictionary* method, uses a dictionary structure with a *paging* heuristic. When the dictionary – a table of strings from the file – is completely filled, the contents are not discarded. Instead, an auxiliary dictionary is created and updated while compression continues using the main dictionary. Each time this auxiliary table is filled, its contents are 'swapped' into the main dictionary and it is cleared. The maintenance of dictionaries for textual substitution is analogous to the semi-space method of garbage collection, in which two pages are used but only one is 'active' – these are exchanged when one fills beyond a preset threshold. Another adaptive variant of this algorithm is the Lempel–Ziv–Welch

(LZW) algorithm, a descendant of LZ78 used in Unix `compress`.[6, 12] Both LZW and LZ78 vary the length of strings used in compression.[6, 12]

Yet another adaptive (alphabetic distribution-based) compression scheme, the Move-To-Front (MTF) method, was developed by Bentley *et al.*[13] In MTF, the 'word code' for a symbol is determined by the position of the word in a sequential list. The word list is ordered so that frequently accessed words are near the front, thus shortening their encodings.

Adaptive compression algorithms are not appropriate to use with heterogeneous files because they are sensitive only to changes in the particular redundancy type with which they are associated, such as a change in the alphabetic distribution. They do not exploit changes across different redundancy types in the files. Therefore a so-called adaptive method typically cannot directly handle drastic changes in file properties, such as an abrupt transition from text to graphics. For example, adaptive Huffman compressors specially optimized for text achieve disproportionately poor performance on certain image files, and vice versa. As the use of multimedia files increases, files exhibiting this sort of transition will become more prevalent.

Our approach differs from adaptive compression because the system chooses each algorithm (as well as the duration of its applicability) before compression begins, rather than modifying the technique for each file during compression. In addition, while adaptive methods make modifications to their compression parameters on the basis of single bytes or fixed length strings of input, our heterogeneous compressor bases its compression upon statistics gathered from larger blocks of five kilobytes. This allows us to handle much larger changes in file redundancy types. This makes our system less sensitive to residual statistical fluctuations from different parts of a file. We note that it is possible to use an adaptive algorithm as a primitive in the system.

Another approach to handling heterogeneous files is the composition of compression algorithms. Composition can either be accomplished by running several algorithms in succession or by combining the basic algorithms and heuristics to create a new technique. For example, recent implementations of 'universal' compression programs execute the Lempel–Ziv algorithm and dynamic Huffman coding in succession, thus improving performance by combining the string repetition-based compression of Lempel–Ziv with the frequency-based compression strategy of dynamic Huffman coding. One commercial implementation is *LHarc*.[14,15] Our system exploits the same savings since it uses the *Freeze* implementation of the Lempel–Ziv algorithm, which filters Lempel–Ziv compressed output through a Huffman coder. An example of a truly composite technique is the compression achieved by using Shannon–Fano tries* in conjunction with the Fiala–Greene algorithm (a variant of Lempel–Ziv)[16] in the PKZIP[2] commercial package. Tries are used to optimally encode strings by character frequency.[17] PKZIP was selected as the representative test program from this group in our experiment due to its superior performance on industrial benchmarks.[9]

Our approach generalizes the ideas of successively executing or combining different compression algorithms by allowing any combination of basic algorithms within a file. This includes switching from among algorithms an arbitrary number of times within a file. The algorithms themselves may be simple or composite and may be adaptive. All are treated as atomic commands to be applied to portions of a file.

* A *trie* is a tree of variable degree ≥ 2 such that (1) each edge is labelled with a character, and the depth of any node represents one more than the number of characters required to identify it; (2) all internal nodes are intermediate and represent prefixes of keys in the trie; (3) keys (strings) may be inserted as leaves using the minimum number of characters which distinguish them uniquely. Thus a generic trie containing the strings *computer* and *compare* would have keys at a depth of five which share a common prefix of length four.

NetApp; Rackspace Exhibit 1014 Page 112

The problem of heterogeneous files was addressed by Toal[18] in a proposal for a naive heterogeneous compression system similar to ours. In such a system, files would be segmented into fixed-length encapsulated blocks; the optimal algorithm would be selected for each block on the basis of their simple taxonomy (qualitative data type) only; and the blocks would be *independently* compressed. Our system, however, performs more in-depth statistical analysis in order to make a more informed selection from the database of algorithms. This entails not only the determination of qualitative data properties but the computation of metrics for an entire block (as opposed to sporadic or random sampling from parts of each block). Furthermore, normalization constants for selection parameters (i.e. the redundancy metrics) are fitted to observed parameters for a test library. Finally, a straightforward but crucial improvement to the naive encapsulated-block method is the implementation of a multi-pass scheme. By determining the complete taxonomy (data type and dominant redundancy type) in advance, any number of contiguous blocks which use the same compression method will be treated as a single segment. Toal observed in preliminary experiments that the overhead of changing compression schemes from one block to another dominated the additional savings that resulted from selection of a superior compression method.[18] This overhead is attributable to the fact that blocks compressed independently (even if the same method is used) are essentially separate files and assume the same startup overhead of the compression algorithm used.* We have determined experimentally that merging contiguous blocks whenever possible obviates the large majority of changes in compression method. This eliminates a sufficient proportion of the overhead to make heterogeneous compression worthwhile.

Commercial products

One of the goals of this research was to develop a compression system which is generally superior to commercially available systems. The four systems we studied are PKZIP, developed for microcomputers running MS-DOS;[2] Unix `compress`;[3] and *StuffIt Classic*[4] and *Compact Pro*,[5] developed for the Apple Macintosh operating system. Each of these products performs its compression in a single pass, with only one method selected per file. Thus, the possibility of heterogeneous files is ignored.

Unix `compress` uses an adaptive version of the Lempel–Ziv algorithm.[6] It operates by substituting a fixed-length code for common substrings. `compress`, like other adaptive textual substitution algorithms, periodically tests its own performance and reinitializes its string table if the amount of compression has decreased.

StuffIt makes use of two sets of algorithms: it first detects special-type files such as image files and processes them with algorithms suited for high-resolution color data; for the remaining files, it queries the operating system for the explicit file type given when the file was created, and uses this information to choose either the LZW variant of Lempel–Ziv,[4, 6] dynamic Huffman coding, or run-length encoding. This is a much more limited selection process than what we have implemented. Additionally, no selection of compression methods is attempted within a file. *Compact Pro* uses the same methodology as *StuffIt* and `compress`, but incorporates an improved Lempel–Ziv derived directly from LZ77.[11] The public-domain version of *StuffIt* is derived from Unix `compress`, as is evident from the similarity of their performance results.

* For purposes of comparison, the block sizes tested by Toal were nearly identical to those used in our system (ranging upwards from 4K).

NetApp; Rackspace Exhibit 1014 Page 113

Compression systems such as *StuffIt* perform simple selection among alternative compression algorithms. The important problem is that they are underequipped for the task of fitting a specific technique to each file (even when the uncompressed data is homogeneous). *StuffIt* uses few heuristics, since its algorithms are intended to be 'multipurpose' . Furthermore, only the file type is considered in selecting the algorithm – that is, no measures of redundancy are computed. Earlier versions of *StuffIt* (which were extremely similar to Unix `compress`) used composite alphabetic and textual compression, but made no selections on the basis of data characteristics. The chief improvements of our heterogeneous compressor over this approach are that it uses a two-dimensional lookup table, indexed by file properties and quantitative redundancy metrics, and – more important – that it treats the file as a collection of heterogeneous data sets.

THE HETEROGENEOUS COMPRESSOR

Our heterogeneous compressor treats a file as a collection of fixed size blocks (5K in the current implementation), each containing a potentially different type of data and thus best compressed using different algorithms. The actual compression is accomplished in two phases. In the first phase, the system determines the type and compressibility of each block. The compressibility of each block of data is determined by the values of three quantitative metrics representing the alphabetic distribution, the average run length and the string repetition ratio in the file. If these metrics are all below a certain threshold, then the block is considered fully compressed (uncompressible) and the program continues on to the next block. Otherwise, using the block type and largest metric, the appropriate compression algorithm (and possible heuristic) are chosen from the compression algorithm database. The compression method for the current block is then recorded in a small array-based map of the file, and the system continues.

The second phase comprises the actual compression and an optimization that maximizes the size of a segment of data to be compressed using a particular algorithm. In this optimization, which is interleaved with the actual compression, adjacent blocks for which exactly the same method have been chosen are merged into a single block. This merge technique maximizes the length of segments requiring a single compression method by greedily scanning ahead until a change of method is detected. Scanning is performed using the array map of the file generated when compression methods were selected from the database. A compression history, needed for decompression, is automatically generated as part of this phase.

The newly compressed segments are written to a buffer by the algorithm, which stores the output data with the compression history. The system then writes out the compressed file and exits with a signal to the operating system that compression was successful.

From this two-pass scheme it is straightforward to see why this system is less susceptible than traditional adaptive systems to biases accrued when the data type changes abruptly during compression. Adaptive compressors perform all operations myopically, sacrificing the ability to see ahead in the file or data stream to detect future fluctuations in the type of data. As a result, adaptive compressors retain the statistical vestiges of the old method until these are 'flushed out' by new data (or balanced out, depending upon the process for paging and aging internal data structures such as dictionaries). Thus adaptive compressors may continue to suffer the effects of bias, achieving suboptimal compression. On the other hand, by abruptly changing compression algorithms, our technique completely discards all remnants of the 'previous' method (i.e. the algorithm used on the preceding segment). This

allows us to immediately capitalize on changes in data. In addition, merging contiguous blocks of the same data type acquires the advantage of incurring all the overhead *at once* for switching to what will be the best compression method for an entire variable-length segment. The primary advantage of adaptive compression techniques over our technique is that the adaptive compression algorithms are 'online' (single-pass). This property increases compression speed and, more important, gives the ability to compress a data stream (for instance, incoming data packets in a network or modem transmission) in addition to files in secondary storage or variable-length buffers.

The remainder of this section presents the system. We begin with a description of the calculation of the block types and the redundancy metrics. We also explain the use of the metrics as absolute indicators of compressibility, and then describe the compression algorithms used and the structure of the database of algorithms. A discussion of implementation details concludes the section.

Property analysis

The compressibility of a block of data and the appropriate algorithm to do so are determined by the type of data contained in a block and the type of redundancy (if any) in the data. These two properties are represented by four parameters: the *block type*, and the three *redundancy metrics*. The block type describes the data in the block – text, binary, graphical, etc. The three redundancy metrics are the degree of variation in character frequency, average run length in the file, and the string repetition ratio of the file. They provide a quantitative measure of how compressible the block is and which type of redundancy is most evident in the block. The use of both quantitative redundancy measures (redundancy metrics) and qualitative characteristics (block types) as indicators for data compressibility is advocated by Held[7] and Salton.[19] We have refined the process for computing those attributes referred to as *datanalysis results* by Held[7] and as *statistical language characteristics* by Salton[19] to obtain an actual guide for compression. The remainder of this section describes how these four parameters are determined for each block.

Block types

The *block type* describes the nature of a segment of input data. There are ten classifications of data in this system: ANSI text, non-natural language text (hexadecimal encodings of binary data), natural language text, computer source code, low redundancy binary, digitized audio, low resolution graphics, high-resolution graphics, high-redundancy binary executable, and binary object data. ANSI text is composed of characters from a superset of the ASCII alphabet. Non-natural language text contains primarily ASCII text but does not follow a distribution of characters like that of human languages. Examples are computer typesetting data, *uuencoded* and *BinHex* encoded data (which has the same character distribution as binary data but is converted to text for ease of transmission). Natural language text includes text written in English as well as other languages which are representable by the Roman (ASCII) alphabet. Most European languages (including the ones using the Cyrillic alphabet), special symbols excluded, fall into this category, as do the Pinyin and Katakana romanizations of the Chinese and Japanese languages (as opposed to their digital encodings). Computer source code uses the ASCII alphabet but characters are distributed with a different frequency than in natural language text. Low-redundancy binaries usually contain compressed data, but may also include data which is merely difficult to compress. Audio

data are very high in redundancy; audio files (and audio segments of multimedia files) are usually extremely large. Low-resolution graphics have long runs of contiguous repeated bits but unlike high-resolution graphics are not suited to lossy compression. High-resolution graphics include color and grayscale and may be compressed with lossy methods. Binary executables, like low-resolution graphics, have long runs of contiguous repeated bits and comprise all compiled programs on a computer system. Finally, object data has slightly shorter runs but is similarly redundant.

To determine the block type we use a procedure `new-file` which is our extension of the Unix `file` command.[20] `file` works by examining the first 512 bytes of a file and comparing the pattern of data contained in it to a collection of known data patterns from Unix and other operating systems. `new-file` works in a similar fashion, with two modifications. First, it examines and compares not only the first 512 bytes of a data set, but also 512 bytes in the middle of the set and the 512 bytes at the end (if they exist). This provides a better indication of the *primary* data type of a file by taking into account the possibility that the properties may change anywhere within the file. Thus, `new-file` decides on the 'most applicable' data type by a majority vote (or the first data type detected in the case of a three-way tie). The other change is that the known patterns of data have been increased by adding three graphics patterns.

Redundancy metrics

The redundancy metrics are quantitative measures that are used to determine the compressibility of a block of data. They are: the *degree of variation in character frequency* or *alphabetic distribution*, M_{AD}; the *average run length* of the block, M_{RL}; and the *string repetition ratio* of the block, M_{SR}. In general, these three manifestations of redundancy are independent. Each of the redundancy types is exploited by different compression algorithms. *Frequency of characters* is exploited by arithmetic or alphabetic encoding algorithms. In arithmetic coding data is represented by an interval that is calculated from the probability distribution of data. With alphabetic coding algorithms such as the Huffman[21] and Shannon–Fano[22] algorithms, more frequently occurring characters are replaced by shorter units than the less frequently occurring characters. *Contiguous strings*, long strings of identical units occurring next to one another, are exploited by run length encoding algorithms.[23] In these algorithms, contiguous strings are replaced by a single occurrence of the string, called a *run*, plus a count of the number of identical strings following. Both alphabetic distribution and average run length are sometimes characterized as statistical redundancy metrics.[24] *Recurrent strings*, which occur repeatedly in the input stream with any number of interleaved symbols, are exploited by *textual substitution algorithms* such as Lempel–Ziv.[6,11,12] In these algorithms, recurrent instances are replaced with positional references (pointers) to the original instance.

Experimental evidence for the efficacy of quantitative redundancy measures is described in texts by Storer[1] and Shannon.[22] Shannon provided an estimate of the entropy of English text, approximately bounding it to be between one and two bits per character.[22] This was determined experimentally by presenting fragments of (unfamiliar) English text to human subjects and recording the frequency with which they guessed unknown letters. The fragments were revealed character by character, so that letters at the end of long or uncommon words were easiest to guess and letters at the beginnings of words were hardest. The observation that binary executables are known to possess high average run lengths is found in Storer.[1] However, this property is rarely exploited or measured.

NetApp; Rackspace Exhibit 1014 Page 116

Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function G to be a number between 0 and 10 so as to simplify comparison among the different metrics. The gamma distribution was chosen because the graph of each of the unscaled redundancy metrics for a test set of 50 files, when plotted on a histogram, approximated a gamma distribution. Normal and χ^2 distributions were also considered, but these proved to be too specific for the application (since they are both specific parametric cases of the gamma distribution). The gamma distribution is defined as follows (cf Ross[25]):

$$\begin{aligned} G_\tau(x_\tau) &= \int_0^{x_\tau} f_\tau(x)\,\mathrm{d}x \\ f_\tau(x) &= \frac{\lambda_\tau \mathrm{e}^{-\lambda_\tau x}(\lambda_\tau x)^{t_\tau - 1}}{\Gamma(t_\tau)} \\ \Gamma(t_\tau) &= \int_0^{\infty} \mathrm{e}^{-y} y^{t_\tau - 1}\,\mathrm{d}y \end{aligned}$$

where f_τ is the density function, Γ is the gamma function, x_τ is the unnormalized measure, t_τ is the shape parameter for the gamma distribution, and λ_τ is the scale parameter for the gamma distribution. The τ subscript simply represents the redundancy type under consideration, i.e. AD, RL, or SR, respectively. The shape and scaling parameters, t_τ and λ_τ respectively, were determined by fitting the best gamma distribution curve to the data set. This was done by performing the preferred compression method for each file and tabulating the induced ratio among normalized metrics to yield the desired parameter values for each segment. These were then averaged to obtain the empirical scaling parameters.

The *alphabetic distribution metric* (the degree of variation in character frequency) of a block is calculated by taking the population (root-mean-square) standard deviation of the ordinal values of characters in the block and dividing it by the block length (in bytes). The M_{AD} metric is calculated by the following formulas:

$$\begin{aligned} M_{\mathrm{AD}} &= 10 * G_{\mathrm{AD}}(x_{\mathrm{AD}}) \\ x_{\mathrm{AD}} &= \frac{\alpha}{\text{block length in bytes}} \\ \alpha &= \sqrt{\frac{\sum_{c \in \text{charset}} (c-\mu)^2}{256}}, \end{aligned}$$

where c is the ordinal value of a character and μ is the average ordinal value of all characters in a block. The normalization uses $t_{\mathrm{AD}} = 1.70$ and $\lambda_{\mathrm{AD}} = 53.0$ as parameters.

The *average run length metric* is obtained by dividing the number of bits in a block by the number of runs. A run is defined to be a repetition of symbols (either bits or bytes). Our implementation takes both bitwise and bytewise run lengths. For example, if $f = 0001111001110000$ is a file of 16 bits, then the number of bit runs is 5, and the number of byte runs is 2. The scaled metric M_{RL} is obtained by:

$$\begin{aligned} M_{\mathrm{RL}} &= 10 * G_{\mathrm{RL}}(x_{\mathrm{RL}}) \\ x_{\mathrm{RL}} &= \frac{\text{file length in bits}}{\text{number of runs}} \end{aligned}$$

NetApp; Rackspace Exhibit 1014 Page 117

with gamma distribution parameters $t_{RL} = 0.50$ and $\lambda_{RL} = 12.0$.

The *string repetition ratio metric* is the total number of n-bit strings in the block divided by the number of distinct n-bit strings (up to 100K). In our implementation, n is 32, the word size of our machine. The normalized metric M_{SR} is obtained by:

$$M_{SR} = 10 * G_{SR}(x_{SR})$$

$$x_{SR} = \frac{\text{number of } n \text{ bit strings}}{\text{number of distinct } n \text{ bit strings}}$$

with gamma distribution parameters $t_{SR} = 0.18$ and $\lambda_{SR} = 0.2$.

The alphabetic distribution and average run length metrics can be calculated in linear time. The string repetition ratio can be computed in $O(n \log n)$ time using a dictionary data structure. For simplicity, and because a (small) constant amount of data is scanned, we use an $O(n^2)$ version. New strings are stored in an array rather than a binary tree, which would require more insertion overhead (and is not worth while for the 5K block length used in the current system). Our routine integrates $f_\tau(x)$ by Simpson's Rule with $n = 10$ intervals.

The largest of the three metrics is assumed to represent the most significant type of redundancy present in the block. It is expected that compression will decrease at least one of the metrics, and experiments conducted on a wide variety of files have proven this convention to be reliable. Experiments have also shown that if all the normalized metrics are smaller than 2.5, the file is considered not compressible, and the system records a verdict of 'uncompressible' on the current block. If at least one of the parameters is greater than 2.5, the file is considered compressible. The maximum of the normalized metrics is then selected and used in conjunction with the file type to select the appropriate compression algorithm from the lookup table described in the following section. A negative compressibility test does not always imply that all three metrics are below the threshold. In some cases, the only redundancy type for which a metric is above the threshold accesses a null entry in the database of compression algorithms. This is interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm.

The algorithm and heuristic database

The compression algorithms and attendant heuristics are organized into the 10 by 3 table illustrated by Table I. The 10 file descriptors are the row indices and the 3 metrics are the column indices. Each entry of the table contains descriptors which are used to access the code for an algorithm-heuristic pair. It should be noted that four of the entries are blank (indicated by an *). A blank entry indicates that the combination of block type and highest metric are very unusual. In this case, the next highest metric is used instead, provided that it is above the threshold. As an example of using this table, consider a high-redundancy binary executable file whose highest metric is the string repetition metric M_{SR}. Together, this pair indicates that the Lempel–Ziv compression algorithm with the Freeze heuristic will be used.

The algorithms

There are four basic algorithms used by the system: arithmetic coding,[26] Lempel–Ziv,[8] run length encoding (RLE),[23] and JPEG for image/graphics compression.[27]

Arithmetic coding algorithms compress data by representing that data by an interval of

Table I. Database of compression algorithms†

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
hexadecimal	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
natural language	arithmetic coding	*	Lempel–Ziv
	*	*	freeze
source code	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
low redundancy	*	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	*
audio	*	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
low resolution	*	run-length encoding	Lempel–Ziv
graphic	*	n-bit run count	freeze
high resolution	JPEG	run-length encoding	JPEG
color graphic	improved Huffman	n-bit run count	improved Huffman
high redundancy	arithmetic coding	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	freeze
object	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze

† Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

real numbers between zero and one. The width of this interval is inversely proportional to the number of symbols encoded, and the decrease in width is directly proportional to the frequency of the original symbols. Thus the interval specifies the encoded message via its bounds, with the precision (distance) of these bounds reflecting the information content of the message. The end result is that arithmetic coding achieves, in practice, much better space savings than Huffman coding and its dynamic implementations because of its higher likelihood of actually achieving the theoretical lower bound.[24, 28] Although early arithmetic coding algorithms performed too slowly to be of practical use,[29] the implementation of the Witten–Neal–Cleary algorithm used here[26] is optimized for speed – at some cost in space savings, but without giving up its advantage over dynamic Huffman coding. The reader is referred to Bell *et al*[24] for a thorough overview of arithmetic coding. We should note that in earlier implementation of the heterogeneous compressor we used a dynamic Huffman algorithm instead of arithmetic coding. We changed our implementation when we found that then Witten–Neal–Cleary algorithm[26] outperformed our implementation of dynamic Huffman coding[10, 30] in both space savings and execution time.

Run length encoding (RLE) algorithms compress data by replacing contiguous occurrences of a single-unit symbol (either bit or byte) by an efficiently coded count of these runs, usually a single occurrence of the symbol and the number of occurrences. We have implemented a straightforward RLE algorithm for our database, based on the description in Sedgewick.[23] In addition, bitwise and bytewise encoding are available as heuristics and the parameters of bitwise RLE are based on the RL metric.

Files with a high degree of string repetition are compressed using the Lempel–Ziv compression algorithm. It compresses data by replacing frequently occurring strings (with min-

NetApp; Rackspace Exhibit 1014 Page 119

imal regard of how far apart they occur) with compact pointers to the position of the first occurrence. Our implementation is a straightforward array-based encoding with constant-length codes. The algorithm maintains a dictionary of recurring strings in order to do the compression. In our system, the Lempel–Ziv algorithm is augmented with the *Freeze* heuristic. This heuristic suppresses paging of strings in the dictionary after it has been filled; that is, it prevents the replacement of previously encountered strings, regardless of how long ago or how infrequently the string has been encountered. *Freeze* is primarily a speed optimization, since it requires less computation than paging heuristics such as least recently used (LRU) or least frequently used (LFU), but it has been shown to work well for all but the least string-redundant files (including both binary executables and most text files). For files with extremely low string-repetition, our system usually selects Huffman compression.

The compression of high-resolution graphics and audio files uses a *lossy* compression scheme. Appropriately used, lossy algorithms guarantee that the decompressed file is similar enough to the original as to be nearly indistinguishable by human perception, and that repeated compression and decompression leads to limited cumulative 'damage' . The primary benefit of lossy compression is that it guarantees much higher compression ratios at a minimal tradeoff. For instance, a very-high-resolution color image can be compressed with much higher savings (possibly 95 per cent) if the user allows a small amount of noise, always less than 1 per cent per compression, to be introduced during each compression. Our system uses the JPEG system[27] for compressing high-resolution color and grayscale images. JPEG, which is divided into lossy and lossless parts, typically achieves compression ratios of between 15-to-1 and 25-to-1. The *potential* for this substantial savings is obtained by the Discrete Cosine Transform portion of the algorithm, a lossy method. This determines a limit on the amount of savings that can then be achieved by any lossless compressor. The actual savings are realized by a lossless portion, known as the *back end* which is applied to the preprocessed image data. The implementation of this module used in our system[27] is a Huffman coder. It is independent of the lossy front end and can be replaced with a run-length or textual-substitution based algorithm, to be selected by the synthesis system. In our implementation, we chose to retain the original Huffman back end, a different algorithm from the general-purpose dynamic Huffman coder which we also studied.[10, 30] This is because the JPEG Huffman coder is especially suited to the redundancy remaining after lossy preprocessing. It is worthy of mention that the JPEG developers have investigated the use of arithmetic coding back ends, which were found to be experimentally superior but were not used because of proprietary considerations.[27]

Implementation

The system consists of a driver module, four block analysis modules, and the synthesis module, which includes the database of compression algorithms. All modules are written in C and were tested on a Unix platform. The program uses a data directed style of implementation for choosing the compression algorithm to apply to a block. Thus, additional block types, compression algorithms and heuristics, and redundancy metrics can be added to the system with minimal modification of the source code. Only the database would have to be updated and the block analysis routines extended; the rest of the program would remain the same.

The driver performs two iterative passes through the file. It first performs block analysis on the file one 5K block at a time. This block size was chosen after experimentation showed that the response of the system to changes in block type became roughly stable as block

NetApp; Rackspace Exhibit 1014 Page 120

size exceeded 5K (i.e., did not significantly increase as block size did), and that a block size of 5K yielded highly accurate metrics (in only 1 of the 20 test files did the heterogeneous compressor select a suboptimal algorithm for any block). Finally, we found that the highest level of adaptivity without a noticeable *decrease* in accuracy was achieved at 5K, hence our choice of 5K as the block size.

For each block, the system invokes the four analysis modules – three for metric computation and normalization and one to determine the file properties – and stores their output. It then performs the metric comparison and combines the results with the file property to complete the table lookup for the current block. An identifying tag for the selected algorithm is written to the 'compression plan' , an array which stores one complete compression instruction per block (if the current block is deemed uncompressible, a 'skip' instruction is recorded).

We pause here to discuss the normalization of the metrics. Originally, we used a naive normalization method: direct algebraic scaling with experimentally determined constants for each metric. This did not, however, accurately reflect the statistical relationship between variance in character frequency and alphabetic redundancy. Also, the behavior of these functions at asymptotes led to poor approximation of the overall distribution of data segments in the test files. The result was that arithmetic coding was too often incorrectly chosen, resulting in inferior compression; and selection approached randomness as metric values for both string repetition and alphabetic distribution tended toward extreme values. Using the gamma normalization method described above resulted in an improvement in the selection of arithmetic coding. Among the 20 benchmark files, arithmetic coding was selected as the compression method in exactly those cases where the other methods performed worse.

The second pass performs the compression of each block. In order to improve performance, this pass includes a simple optimization step which circumvents the overhead of restarting compression after each fixed length block by merging contiguous blocks that are to be compressed using the same compression algorithms.

On this same pass through the file, the system compresses each of the newly merged blocks using the algorithm recorded in the compression plan. The compressed data is written to an output buffer, while the compressed length (which indicates where in the compressed file a compressed block begins and ends) and compression method are recorded in a separate history for reference at decompression time. If negative compression or no compression is achieved, or if the block was already marked uncompressible, then the data is copied directly to the output buffer (the full block length and a code for 'no compression' are recorded in the compression history). Upon reaching the end of the blocks, the system writes out the compressed data from the output buffers and prepends the encoded compression history to produce the final output file.

When decompression is invoked, the driver module opens the compressed file, interprets the history tag and performs the necessary operations. The tags are a stored version of the compression history in compact, encoded form. Since the heterogeneous system generates different compression sequences for each file, and since the length of a compressed block varies with both the length of the original block and the compression method used, these tags are necessary to guide the decompression process. Currently only the compressed lengths of each block and the method of compression are stored, but a checksum for the original (decompressed) block length can be added with negligible overhead. When executed in reverse order on each compressed block, the instructions in the history tags result in the original file. For simplicity and security, they are prepended to the compressed file (and can easily be encrypted).

EXPERIMENTAL RESULTS

Design and construction of the test files

To test the overall performance, the system was run on a set of 20 test files. These files range in length from approximately 39K to 366K, with representative files from each of the ten block types included in the test corpus.

The test files are designed to model certain types of heterogeneous files, including utilities for image viewing, business, or audio processing, and hypothetical multimedia databases and programs. To construct these files, a collection of 30 files from the Unix, Apple Macintosh, and MS-DOS (IBM PC) operating systems was created. These files are listed in Table II. To create the test corpus, they were concatenated in groups of 2 or 3. The resultant series of test files is listed in Table III. All of the source files were used. The goal was to generate as broad a range of permutations as possible (while restricting the generated files to those which are likely to exist in a typical user environment). This was performed manually with consideration toward combinatorial constraints and the criteria of realistic data modeling. Since all of the files in the source collection originate from common commercial sources or from public archives (with the exception of the source and object files, which are from the code for the heterogeneous compressor itself), the latter constraint was considerably simplified.

The assembled files were then ported to the test sites (a Sun workstation for Unix `compress` and our heterogeneous compressor; a Macintosh for *StuffIt* and *Compact Pro*; and an IBM 80486 machine for PKZIP). Binary file transfer mode was used to ensure that the file lengths agreed exactly among all platforms.

Performance

In this section, we review and analyze the performance of the heterogeneous compressor with respect to compression savings, as compared with four of the commercial systems previously discussed; and execution time. Finally, we briefly note the implications of running the experiments and compiling performance data on several different architectures.

Compression savings

The total length of the uncompressed benchmark suite is just under three megabytes. Table IV shows the compressed length achieved by Unix `compress`, PKZIP, *StuffIt*, *Compact Pro* and the heterogeneous compression system. The heterogeneous compressor achieved the greatest compression, with a total compressed length of 1828K. This represents an additional savings of 162K (more than eight per cent) over the best commercial system (*Compact Pro v1.32*), and 339K (nearly 16 per cent) over the average. Compressed lengths for the commercial methods ranged from 1990K to 2375K.

Table V compares the percentage savings obtained by our system to the savings obtained by the commercial programs and the heterogeneous system. The last two columns show the difference in per cent saved between the synthesis system and the best and average of the four commercial packages. The best commercial compressor is marked for each of the files. Note that the heterogeneous compressor does better than all commercial programs in 19 of 20 cases and better than three of the four commercial systems in this one case (file 15). The difference in compression for this file is only 0.02 per cent, whereas for all the other files, the heterogeneous compressor has at least a 1.3 per cent improvement over the best

NetApp; Rackspace Exhibit 1014 Page 122

Table II. Files used to compose the test suite and their respective origins

File designation	File name	File type
audio1	cosby.snd	SoundMaster Macintosh audio file
lowrd1	ticker.txt	ASCII characters from stock ticker
lowrd2	exsound	compressed World Builder sound library
lowrd3	huff	compressed Unix executable
lowrd4	appnote.uue	uuencoded text
text1	phrack.txt	English text
text3	techbook.txt	Unix news article
text4	quanta1.txt	English text
text5	attilla.fluff	English text
text6	shadow.fluff	English text
text7	quanta2.txt	English text
execu1	ad	Unix executable
execu2	sh	Unix executable
execu3	blob	Silicon Graphics executable
execu4	zero	Silicon Graphics executable
execu5	network2.exe	IBM PC executable
execu6	hostname	Unix executable
graph1	compmisc.drw	Lotus Freelance line drawing
graph2	compperi.drw	Lotus Freelance line drawing
graph3	computer.drw	Lotus Freelance line drawing
graph4	lowres.mpt	MacPaint file
graph5	3dbar.drw	Lotus Freelance 3-D bar chart
graph6	image.ppm	PPM (high-resolution image) file
graph7	grp4	MacPaint file
objec1	test1.o	Unix object file
objec2	test2.o	Unix object file
objec3	test3.o	Unix object file
source1	table.c	C source code
source2	freeze.c	C source code

commercial compressor. The average of each column appears in the bottom row; note that the 'percent difference' averages are not weighted by file length, as each file is considered a separate experiment.

Because the quality of compression by the synthesis system depends on that of the algorithms and heuristics used, improvement of the implementations that we use should yield higher performance. This is evidenced by comparing the results of compressing a file dominated by string repetitions by Unix `compress` and *Compact Pro*. Both are implementations of the Lempel–Ziv algorithm. Unix `compress` has no heuristics, whereas *Compact Pro* is a better implementation of LZ77.[5, 11] *Compact Pro* consistently outperforms `compress`. It should be noted that the performance of the *Freeze* variant of Lempel–Ziv[8] used in our sys-

Table III. Combinations of the test files and the resultant simulated data types

File number	File composition	Classification of data modeled
1	text1 — lowrd1	news or stock report
2	graph7 — objec1	object file for a graphics viewer
3	lowrd1 — text3 — graph4	multimedia application (text/graphics)
4	graph7 — execu3	graphics viewer
5	audio1 — graph1	multimedia data file (sound/graphics)
6	text2 — lowrd1 — graph3	multimedia data file (text/graphics)
7	lowrd3 — execu1	commercial utility
8	graph2 — lowrd2 — execu2	multimedia application (graphics/sound/executable)
9	source1 — lowrd3 — graph6	multimedia data or source file (source/compressed binary/image)
10	audio1 — text4	multimedia data file (sound/text)
11	lowrd1 — execu4	statistical application with data
12	graph7 — text5	multimedia data file (text/graphics)
13	lowrd2 — text6	multimedia data file (sound/text)
14	text3 — audio1 — graph5	multimedia data file (text/sound/graphics)
15	lowrd1 — text4 — source2	source file for multimedia program (text/source code)
16	text7 — lowrd2 — graph3	multimedia data file (text/compressed audio/graphics)
17	graph4 — audio1 — execu5	multimedia application (sound/graphics)
18	execu4 — graph7 — text4	multimedia application (graphics/text)
19	objec3 — lowrd3 — execu6	commercial utility
20	objec2 — audio1 — execu2	audio application

tem does consistently better than `compress` and is comparable to *Compact Pro* on standard industrial benchmarks.[9] Improving algorithms and adding or substituting new heuristics would also yield more savings.

Execution times and speed optimizations

In this section we compare, in *approximate units*, the running time of the heterogeneous compressor against those of the four commercial systems the savings rates of which for our test files are documented above. The units are approximate for two reasons. First, because the four test systems are commercial the source code for three of them is not publicly available*, which renders an exact measure of *user* time infeasible. This concern is in part assuaged by the non-multitasked, single-user nature of the microcomputer operating systems on which three (`compress` for Linux notwithstanding) of the commercial systems reside. Second, however, the drastic architectural and organizational differences among the various native machines renders uniform comparisons unreliable. This applies even to normalized execution times because the host machines differ not merely in clock cycle speed, but in instruction set architecture and dynamic instruction frequencies for similar compression algorithms. The *exact* running times reported in this section is only that of the heterogeneous

* As noted, however, the Lempel–Ziv implementation employed by *StuffIt Classic* is nearly identical to that of Unix `compress`.

compressor. These comprise the non-commercial* compression systems for which source code is available for profiling. For the commercial systems we report the observed wall clock time to provide a standard of comparison, but note that the host machines vary in computational power.

Table IV. Results of the four popular commercial programs and the heterogeneous compression system, applied to the 20 test files

File number	Original length	Unix `compress`	PKZIP v1.10	*StuffIt Classic*	*Compact Pro* v1.32	Heterogeneous compressor
1	39,348	20,578	17,119	20,575	16,831	16,315
2	44,202	44,202	39,813	40,412	41,112	37,388
3	46,629	46,629	46,629	43,261	40,367	36,477
4	59,254	52,076	40,571	45,202	41,607	38,007
5	169,108	168,903	151,478	149,701	148,917	134,524
6	100,476	69,771	53,043	65,417	52,349	50,906
7	131,663	131,663	103,544	106,643	109,979	96,429
8	220,644	190,971	137,886	173,677	137,401	127,384
9	301,805	145,993	112,503	137,685	115,096	103,730
10	255,306	204,457	191,378	206,193	183,313	168,675
11	59,305	30,178	22,782	29,701	22,858	21,774
12	51,715	51,715	43,032	46,462	44,107	40,229
13	63,189	63,189	58,247	59,569	59,934	54,481
14	196,789	176,276	196,789	172,486	151,057	137,052
15	148,908	73,555	63,748	75,595	64,618	63,778
16	164,535	141,067	132,992	135,245	110,093	104,175
17	203,912	203,912	184,657	189,398	202,821	170,564
18	200,640	128,675	107,728	125,461	104,711	101,674
19	366,557	265,114	198,727	265,027	198,756	187,659
20	278,152	223,277	193,980	224,943	191,763	181,030
Total	3,102,137	2,432,201	2,096,646	2,312,653	2,037,690	1,872,251

The running times for the commercial systems on the entire test suite documented above appear in Table VI. All of the execution times are measured in wall clock units except for the heterogeneous compressor's, which is a total of user times as reported by `prof`, the C profiler under Unix. The wall clock time was empirically observed not to differ noticeably from this total on an unloaded Unix machine. The commercial systems were similarly tested on unloaded (or single-task) systems.

For Unix `compress`, the mean running time was 26 s, where the average was taken over runs on different Sun workstations of comparable power (documented below). A Unix implementation of PKZIP was also tested on one of these Sun workstations, and achieved an execution time of 56 s – only slightly better than the personal computer version. The running time of 856 s placed the heterogeneous compressor in the middle to high end of the commercial compressors in terms of running time.

* For this purpose we continue to consider Unix `compress` commercial, due to its wide range of versions.

Table V. Percent savings for the test compression systems*

File number	Unix compress (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

NetApp; Rackspace Exhibit 1014 Page 126

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix `compress`	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix `compress`, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

We would like to thank Leonid Broukhis, Graham Toal, and Kenneth Zeger for discussions on some of the research reported here. We also thank Jonathan Eifrig, Bill Goodman, and Tom Lane for guidance on several technical issues. Finally, we thank the anonymous reviewers for their comments and suggestions, especially for introduction to relevant literature in arithmetic coding.

REFERENCES

1. James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.
2. Phillip W. Katz, PKZIP. Commercial compression system, version 1.1, 1990.
3. Sun Microsystems, compress. Commercial compression system, operating system version 5.3, September 1992.
4. Raymond Lau, StuffIt Classic and StuffIt Deluxe. Commercial compression system, 1990.
5. Bill Goodman, Compact Pro. Commercial compression system, v1.32, 1991.
6. Terry A. Welch, 'A technique for high performance data compression', *IEEE Computer*, **17**(6), 8–19 (1984).
7. Gilbert Held and Thomas R. Marshall, *Data Compression: Techniques and Applications: Hardware and Software Considerations*, 3rd edn, John Wiley and Sons, 1991.
8. Leonid Broukhis, Freeze implementation of LZHuf algorithm. comp.sources.misc archives, Internet, 1991.
9. Jean-Loup Gailly, comp.compression benchmark (Calgary test corpus). In comp.compression FAQ list, J. Gailly, (ed.), 1992.
10. Jeffrey S. Vitter, 'Dynamic Huffman Coding', *ACM Transactions on Mathematical Software*, (June 1989).
11. J. Ziv and A. Lempel, 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory*, **23**,(3), 337–343 (1977).
12. J. Ziv and A. Lempel, 'Compression of individual sequences via variable-rate coding', *IEEE Transactions on Information Theory*, **24**(5), 530–546 (1978).
13. Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan and Victor K. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, 320–330 (April 1986).
14. Yooichi Tagawa, Haruhiko Okumura and Haruyasu Yoshizaki, LZHuf: encoding/decoding module for LHarc. Compression system, version 0.03 (Beta), 1989.
15. Haruyasu Yoshizaki, LHA: A high-performance file-compression program. Compression system, version 2.11, 1991.
16. Edward R. Fiala and Daniel H. Greene, 'Data compression with finite windows', *Communications of the ACM*, 490–505 (1989).
17. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, Maryland, second edition, 1987.
18. Graham Toal. Personal communication. Unpublished, 1992.
19. Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
20. Ian F. Darwin, file (program). Berkeley Unix operating system, 1987.
21. David A. Huffman, 'A method for the construction of minimum-redundancy codes', *Proceedings of the IRE*, number 40, 1952, pp. 1098–1101.
22. Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communications*, University of Illinois Press, Urbana and Chicago, 1963.
23. Robert Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
24. Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
25. Sheldon Ross, *A First Course in Probability*, Macmillan Publishing Company, New York, third edition, 1988.
26. Ian H. Witten, Radford Neal and John G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**(6), 520–540 (1987).
27. Independent JPEG Group. 'JPEG image compression system', think.com FTP archives, Internet, 1994.
28. Jean-Loup Gailly. comp.compression/comp.compression.research FAQ list. J. Gailly (ed.), URL `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html`, 1994.
29. James A. Storer, *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
30. Graham Toal. C implementation of dynamic Huffman compressor by J. S. Vitter. comp.source.misc archives, Internet, 1990.

NetApp; Rackspace Exhibit 1014 Page 128

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1117–1140 (OCTOBER 1995)

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System

PAUL ASHTON AND JOHN PENNY
Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand
(email: paul@cosc.canterbury.ac.nz)

SUMMARY

The paper describes in detail the implementation of INMON, a prototype monitor for recording and displaying interaction networks. The *interaction network* is designed to show the interrelationships between significant events that occur during the processing of an interaction on a loosely-coupled distributed system. By being directed at analysing what happens in an interaction, this approach is fundamentally different from other graphical representations that show what happens during the execution of a single program on a distributed system. Examples are given of interaction networks recorded by INMON. The approach is based on very general models of a distributed system and of an interaction, and could be widely applied. We conclude by summarizing what is needed to provide facilities within any operating system for recording interaction networks.

KEY WORDS: loosely-coupled distributed systems; interactive performance; visualization tools

INTRODUCTION

Recently, there has been a great deal of interest in visualization techniques for complex systems in science and engineering.[1,2] Visualization techniques have also been developed for computer systems, in particular for distributed and parallel systems. The aim may be to achieve better performance,[3] or simply to get a clearer understanding of what happens in practice during the execution of programs on a complex, modern computer system.[4–6]

In this paper, we present first the idea of the *interaction network*, a directed graph designed to represent the execution of an interaction on a loosely-coupled distributed system. Each interaction network shows the interrelationships between significant events that occur during execution of the interaction. In representing what happens during an *interaction*, the interaction network is significantly different in nature and purpose from other graphical representations[3,6,7] that show what happens during execution of a *program*.

We then discuss in detail the implementation of INMON, a prototype monitor for recording interaction networks for a network of Sun 3 workstations. The implementation required modifications to the kernel of the SunOS operating system, and the finding of a software solution to the problem of clock synchronization. We follow with some examples of interaction networks recorded by INMON, to illustrate what can be discovered from studying interaction networks.

Monitors that collect data to show graphically what happens during execution should, we feel, be built-in to operating system kernels at the time of design. We conclude by

CCC 0038–0644/95/101117–24
©1995 by John Wiley & Sons, Ltd.
Received 1 December 1992
Revised 21 March 1995

SOFTWARE
PRACTICE
AND
EXPERIENCE

25
JULY-DEC.
1995

OFTWARE
ACTICE & EXPERIENCE

No. 10

OCTOBER 1995

EDITORS

DOUGLAS COMER

ANDY WELLINGS

Software: practice & experience
Received on: 11-03-95
Illinois Institute of Technology Library

WILEY
Publishers Since 1807
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley–Interscience Publication

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

SOFTWARE

PRACTICE & EXPERIENCE

Editors

Professor D. E. Comer, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Charlotte I. Tubis, U.S. Editorial Assistant, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Dr A. J. Wellings, Department of Computer Science, University of York, Heslington, York YO1 5DD

Advisory Editorial Board

Professor D. W. BARRON
Department of Electronics and Computer Science,
University of Southampton,
Southampton SO9 5NH, U.K.

Professor P. J. BROWN
Computing Laboratory, The University,
Canterbury, Kent CT2 7NF, U.K.

Professor J. A. CAMPBELL
Department of Computer Science, University College London,
Gower Street, London WC1E 6BT, U.K.

Professor F. J. CORBATO
Electrical Engineering Department,
Massachusetts Institute of Technology,
545 Technology Square,
Cambridge, Massachusetts 02139, U.S.A.

Dr. Christopher W. FRASER
AT&T Bell Laboratories, 600 Mountain Ave 2C-464,
Murray Hill, NJ 07974-0636, U.S.A.

Professor PER BRINCH HANSEN
School of Computer and Information Science,
4-116 CST, Syracuse University,
Syracuse, New York 13210, U.S.A.

Professor D. R. HANSON
Department of Computer Science,
Princeton University, Princeton,
New Jersey 08544, U.S.A.

Professor J. KATZENELSON
Faculty of Electrical Engineering,
Technion-Israel Institute of Technology,
Haifa, Israel

Dr. B. W. KERNIGHAN
AT&T Bell Laboratories, 600 Mountain Avenue,
Murray Hill, New Jersey 07974, U.S.A.

Professor D. E. KNUTH
Department of Computer Science, Stanford University,
Stanford, California 94305, U.S.A.

Dr. B. W. LAMPSON
180 Lake View Ave,
Cambridge,
MA 02138, U.S.A.

Dr. C. A. LANG
Three-Space Ltd,
70 Castle Street,
Cambridge CB3 0AJ, U.K.

Professor B. RANDELL
Computing Laboratory,
University of Newcastle-upon-Tyne,
Claremont Tower, Claremont Road,
Newcastle-upon-Tyne NE1 7RU, U.K.

Professor J. S. ROHL
Department of Computer Science,
The University of Western Australia,
Nedlands, Western Australia 6009.

D. T. ROSS
Softech Inc., 460 Totten Pond Road,
Waltham, Massachusetts 02154, U.S.A.

B. H. SHEARING
The Software Factory,
28 Padbrook, Limpsfield, Oxted,
Surrey RH8 0DW, U.K.

Professor N. WIRTH
Institut für Computersysteme, ETH-Zentrum,
CH-8092 Zürich, Switzerland.

Aims and Scope

Software—Practice and Experience is an internationally respected and rigorously refereed vehicle for the dissemination and discussion of practical experience with new and established software for both systems and applications. Contributions regularly: (a) describe detailed accounts of completed software-system projects which can serve as 'how-to-do-it' models for future work in the same field; (b) present short reports on programming techniques that can be used in a wide variety of areas; (c) document new techniques and tools that aid in solving software construction problems; and (d) explain methods/techniques that cope with the special demands of large scale software projects. The journal also features timely Short Communications on rapidly developing new topics.

The editors actively encourage papers which result from practical experience with tools and methods developed and used in both academic and industrial environments. The aim is to encourage practitioners to share their experiences with design, implementation and evaluation of techniques and tools for software and software systems.

Papers cover software design and implementation, case studies describing the evolution of system and the thinking behind them, and critical appraisals of software systems. The journal has always welcomed tutorial articles describing well-tried techniques not previously documented in computing literature. The emphasis is on practical experience; articles with theoretical or mathematical content are included only in cases where an understanding of the theory will lead to better practical systems.

Articles range in length from a Short Communication (half to two pages) to the length required to give full treatment to a substantial piece of software (40 or more pages).

Advertising: For details contact—
Michael J. Levermore, Advertisement Sales, John Wiley & Sons Ltd, Baffins Lane, Chichester, Sussex PO19 1UD, England (Telephone 01243 770351, Fax 01243 775878, Telex 86290)

Software—Practice and Experience (ISSN 0038-0644/USPS 890-920) is published monthly, by John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, England. Second class postage paid at Jamaica, N.Y. 11431. Air freight and mailing in the U.S.A. by Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003.

© 1995 by John Wiley & Sons Ltd.

Printed and bound in Great Britain by Page Bros, Norwich. Printed on acid-free paper.

To subscribe: Orders should be addressed to Subscriptions Department, John Wiley & Sons Limited, Baffins Lane, Chichester, Sussex, PO19 1UD, England. 1995 subscription price (13 issues): U.S. $825.00.

U.S.A. POSTMASTER: Send address changes to *Software—Practice and Experience*, c/o Publications Expediting Services Inc., 200 Meacham Avenue, Elmont, N.Y. 11003, U.S.A.

SOFTWARE—PRACTICE AND EXPERIENCE
(Softw. pract. exp.)

CONTENTS

VOLUME 25, ISSUE No. 10 *October 1995*

Migration in Object-oriented Database Systems—A Practical Approach: C. Huemer, G. Kappel and S. Vieweg 1065

Automatic Synthesis of Compression Techniques for Heterogeneous Files: W. H. Hsu and A. E. Zwarico 1097

A Tool for Visualizing the Execution of Interactions on a Loosely-coupled Distributed System: P. Ashton and J. Penny 1117

Process Scheduling and UNIX Semaphores: N. Dunstan and I. Fris 1141

Software Maintenance: An Approach to Impact Analysis of Objects Change: S. Ajila 1155

SPEXBL 25(10) 1065–1182 (1995)
ISSN 0038–0644

Indexed or abstracted by Cambridge Scientific Abstracts, CompuMath Citation Index (ISI), Compuscience Database, Computer Contents, Computer Literature Index, Computing Reviews, Current Contents/Eng, Tech & Applied Sciences, Data Processing Digest, Deadline Newsletter, Educational Technology Abstracts, Engineering Index, Engineering Societies Library, IBZ (International Bibliography of Periodical Literature), Information Science Abstracts (Plenum), INSPEC, Knowledge Engineering Review, Nat Centre for Software Technology, Research Alert (ISI) and SCISEARCH Database (ISI).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(10), 1097–1116 (OCTOBER 1995)

Automatic Synthesis of Compression Techniques for Heterogeneous Files

WILLIAM H. HSU

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
(email: bhsu@cs.uiuc.edu, voice: (217) 244-1620)

AND

AMY E. ZWARICO

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.
(email: amy@cs.jhu.edu, voice: (410) 516-5304)

SUMMARY

We present a compression technique for heterogeneous files, those files which contain multiple types of data such as text, images, binary, audio, or animation. The system uses statistical methods to determine the best algorithm to use in compressing each block of data in a file (possibly a different algorithm for each block). The file is then compressed by applying the appropriate algorithm to each block. We obtain better savings than possible by using a single algorithm for compressing the file. The implementation of a working version of this heterogeneous compressor is described, along with examples of its value toward improving compression both in theoretical and applied contexts. We compare our results with those obtained using four commercially available compression programs, PKZIP, Unix `compress`, *StuffIt*, and *Compact Pro*, and show that our system provides better space savings.

KEY WORDS: adaptive/selective data compression algorithms; redundancy metrics; heterogeneous files; program synthesis

INTRODUCTION

The primary motivation in studying compression is the savings in space that it provides. Many compression algorithms have been implemented, and with the advent of new hardware standards, more techniques are under development. Historically, research in data compression has been devoted to the development of algorithms that exploit various types of redundancy found in a file. The shortcoming of such algorithms is that they assume, often inaccurately, that files are homogeneous throughout. Consequently, each exploits only a subset of the redundancy found in the file.

Unfortunately, no algorithm is effective in compressing all files.[1] For example, dynamic Huffman coding works best on data files with a high variance in the frequency of individual characters (including some graphics and audio data), achieves mediocre performance on natural language text files, and performs poorly in general on high-redundancy binary data. On the other hand, run length encoding works well on high-redundancy binary data, but performs very poorly on text files. Textual substitution works best when multiple-character strings tend to be repeated, as in English text, but this performance degrades as the average

CCC 0038–0644/95/101097–20
©1995 by John Wiley & Sons, Ltd.
Received 20 April 1994
Revised 5 February 1995

length of these strings decreases. These relative strengths and weaknesses become critica when attempting to compress *heterogeneous* files. Heterogeneous files are those which con tain multiple types of data such as text, images, binary, audio, or animation. Consequently their constituent parts may have different degrees of compressibility. Because most com pression algorithms are either tailored to a few specific classes of data or are designed t handle a single type of data at a time, they are not suited to the compression of heteroge neous files. In attempting to apply a single method to such files, they forfeit the possibilit of greater savings achievable by compressing various segments of the file with differen methods.

To overcome this inherent weakness found in compression algorithms, we have develope a *heterogeneous compressor* that automatically chooses the best compression algorithm t use on a given variable-length block of a file, based on both the qualitative and quantita tive properties of that segment. The compressor determines and then applies the selecte algorithms to the blocks separately. Assembling compression procedures to create a speci ically tailored program for each file gives improved performance over using one progra for all files. This system produces better compression results than four commonly availabl compression packages, PKZIP,[2] Unix `compress`,[3] *StuffIt*,[4] and *Compact Pro*[5] for arbitra heterogeneous files.

The major contributions of this work are twofold. The first is an improved compressio system for heterogeneous files. The second is the development of a method of statist cal analysis of the compressibility of a file (its redundancy types). Although the conce of redundancy types is not new,[6,7] synthesis of compression techniques using redundanc measurements is largely unprecedented. The approach presented in this paper uses a straigh forward program synthesis technique: a *compression plan*, consisting of instructions for ea block of input data, is generated, guided by the statistical properties of the input data. B cause of its use of algorithms specifically suited to the types of redundancy exhibited b the particular input file, the system achieves consistent average performance throughout t file, as shown by experimental evidence.

As an example of the type of savings our system produces, consider compressing heterogeneous file (such as a small multimedia data file) consisting of 10K of low redu dancy (non-natural language) ASCII data, 10K of English text, and 25K of graphics. this case, a reasonably sophisticated compression program might recognize the increas savings achievable by employing Huffman compression, to better take advantage of the fa that the majority of the data is graphical. However, none of the general-purpose compre sion methods under consideration are optimal when used alone on this file. This is becau the text part of this file is best compressed by textual substitution methods (e.g., Lempe Ziv) rather than statistical methods, while the low-redundancy data* and graphics pa are best compressed by alphabetic distribution-based methods (e.g., arithmetic or dynam Huffman coding) rather than Lempel–Ziv or run-length encoding. This particular file tot 45K in length before compression. A compressor using pure dynamic Huffman coding o achieves about 7 per cent savings for a compressed file of length 42.2K. One of the b general-purpose Lempel–Ziv compressors currently available[8,9] achieves 18 per cent sa ings, producing a compressed file of length 37.4K. Our system uses arithmetic coding the first and last segments and Lempel–Ziv compression on the text segment in the midd achieving a 22 per cent savings and producing a compressed file of length 35.6K. This a 4 per cent improvement over the best commercial system.

* This denotes, in our system, a file with a low rate of repeated strings.

NetApp; Rackspace Exhibit 1014 Page 135

The purpose of our experiments was to verify the conjecture that a selective system for combining methods can improve savings on a significant range of heterogeneous files, especially multimedia data. This combination differs from current adaptive methods in that it switches among compression paradigms designed to remove very different types of redundancy. By contrast, existing adaptive compression programs are sensitive only to changes in particular types of redundancy, such as run-length, which do not require changing the underlying algorithm used in compression. Thus they cannot adapt to changes in the type of redundancy present, such as from high run-length to high character repetition. The superiority of our approach is demonstrated in our experimental section.

This paper begins with a presentation of existing approaches to data compression, including a discussion of hybrid and adaptive compression algorithms and a description of four popular commercial compression packages. These are followed by documentation on the design of the heterogeneous compression system, analysis of experimental results obtained from test runs of the completed system, and comparison of the system's performance against that of commercial systems. Finally, implications of the results and possibilities for future work are presented.

RELATED WORK

It is a fundamental result of information theory that there is no single algorithm that performs optimally in compressing all files.[1] However, much work has been done to develop algorithms and techniques that work nearly optimally on all classes of files. In this section we discuss adaptive algorithms, composite algorithms, and four popular commercial compression packages.

Adaptive compression algorithms and composite techniques

Exploiting the heterogeneity in a file has been addressed in two ways: the development of *adaptive* compression algorithms, and the composition of various algorithms. Adaptive compression algorithms attune themselves gradually to changes in the redundancies within a file by modifying parameters used by the algorithm, such as the dictionary, during execution.

For example, adaptive alphabetic distribution-based algorithms such as dynamic Huffman coding[10] maintain a tree structure to minimize the encoded length of the most frequently occurring characters. This property can be made to change continuously as a file is processed.

An example of an adaptive textual substitution algorithm is Lempel–Ziv compression, a title which refers to two distinct variants of a basic textual substitution scheme. The first variant, known as LZ77 or the *sliding dictionary* or *sliding window* method, selects positional references from a constant range of preceding strings.[1, 11] These 'back-pointers' literally encode position and length of a repeated string. The second variant, known as LZ78 or the *dynamic dictionary* method, uses a dictionary structure with a *paging* heuristic. When the dictionary – a table of strings from the file – is completely filled, the contents are not discarded. Instead, an auxiliary dictionary is created and updated while compression continues using the main dictionary. Each time this auxiliary table is filled, its contents are 'swapped' into the main dictionary and it is cleared. The maintenance of dictionaries for textual substitution is analogous to the semi-space method of garbage collection, in which two pages are used but only one is 'active' – these are exchanged when one fills beyond a preset threshold. Another adaptive variant of this algorithm is the Lempel–Ziv–Welch

NetApp; Rackspace Exhibit 1014 Page 136

(LZW) algorithm, a descendant of LZ78 used in Unix `compress`.[6, 12] Both LZW and LZ78 vary the length of strings used in compression.[6, 12]

Yet another adaptive (alphabetic distribution-based) compression scheme, the Move-To-Front (MTF) method, was developed by Bentley *et al.*[13] In MTF, the 'word code' for a symbol is determined by the position of the word in a sequential list. The word list is ordered so that frequently accessed words are near the front, thus shortening their encodings.

Adaptive compression algorithms are not appropriate to use with heterogeneous files because they are sensitive only to changes in the particular redundancy type with which they are associated, such as a change in the alphabetic distribution. They do not exploit changes across different redundancy types in the files. Therefore a so-called adaptive method typically cannot directly handle drastic changes in file properties, such as an abrupt transition from text to graphics. For example, adaptive Huffman compressors specially optimized for text achieve disproportionately poor performance on certain image files, and vice versa. As the use of multimedia files increases, files exhibiting this sort of transition will become more prevalent.

Our approach differs from adaptive compression because the system chooses each algorithm (as well as the duration of its applicability) before compression begins, rather than modifying the technique for each file during compression. In addition, while adaptive methods make modifications to their compression parameters on the basis of single bytes or fixed length strings of input, our heterogeneous compressor bases its compression upon statistics gathered from larger blocks of five kilobytes. This allows us to handle much larger changes in file redundancy types. This makes our system less sensitive to residual statistical fluctuations from different parts of a file. We note that it is possible to use an adaptive algorithm as a primitive in the system.

Another approach to handling heterogeneous files is the composition of compression algorithms. Composition can either be accomplished by running several algorithms in succession or by combining the basic algorithms and heuristics to create a new technique. For example, recent implementations of 'universal' compression programs execute the Lempel–Ziv algorithm and dynamic Huffman coding in succession, thus improving performance by combining the string repetition-based compression of Lempel–Ziv with the frequency-based compression strategy of dynamic Huffman coding. One commercial implementation is *LHarc*.[14,15] Our system exploits the same savings since it uses the *Freeze* implementation of the Lempel–Ziv algorithm, which filters Lempel–Ziv compressed output through a Huffman coder. An example of a truly composite technique is the compression achieved by using Shannon–Fano tries* in conjunction with the Fiala–Greene algorithm (a variant of Lempel–Ziv)[16] in the PKZIP[2] commercial package. Tries are used to optimally encode strings by character frequency.[17] PKZIP was selected as the representative test program from this group in our experiment due to its superior performance on industrial benchmarks.[9]

Our approach generalizes the ideas of successively executing or combining different compression algorithms by allowing any combination of basic algorithms within a file. This includes switching from among algorithms an arbitrary number of times within a file. The algorithms themselves may be simple or composite and may be adaptive. All are treated as atomic commands to be applied to portions of a file.

* A *trie* is a tree of variable degree ≥ 2 such that (1) each edge is labelled with a character, and the depth of any node represents one more than the number of characters required to identify it; (2) all internal nodes are intermediate and represent prefixes of keys in the trie; (3) keys (strings) may be inserted as leaves using the minimum number of characters which distinguish them uniquely. Thus a generic trie containing the strings *computer* and *compare* would have keys at a depth of five which share a common prefix of length four.

NetApp; Rackspace Exhibit 1014 Page 137

The problem of heterogeneous files was addressed by Toal[18] in a proposal for a naive heterogeneous compression system similar to ours. In such a system, files would be segmented into fixed-length encapsulated blocks; the optimal algorithm would be selected for each block on the basis of their simple taxonomy (qualitative data type) only; and the blocks would be *independently* compressed. Our system, however, performs more in-depth statistical analysis in order to make a more informed selection from the database of algorithms. This entails not only the determination of qualitative data properties but the computation of metrics for an entire block (as opposed to sporadic or random sampling from parts of each block). Furthermore, normalization constants for selection parameters (i.e. the redundancy metrics) are fitted to observed parameters for a test library. Finally, a straightforward but crucial improvement to the naive encapsulated-block method is the implementation of a multi-pass scheme. By determining the complete taxonomy (data type and dominant redundancy type) in advance, any number of contiguous blocks which use the same compression method will be treated as a single segment. Toal observed in preliminary experiments that the overhead of changing compression schemes from one block to another dominated the additional savings that resulted from selection of a superior compression method.[18] This overhead is attributable to the fact that blocks compressed independently (even if the same method is used) are essentially separate files and assume the same startup overhead of the compression algorithm used.* We have determined experimentally that merging contiguous blocks whenever possible obviates the large majority of changes in compression method. This eliminates a sufficient proportion of the overhead to make heterogeneous compression worthwhile.

Commercial products

One of the goals of this research was to develop a compression system which is generally superior to commercially available systems. The four systems we studied are PKZIP, developed for microcomputers running MS-DOS;[2] Unix `compress`;[3] and *StuffIt Classic*[4] and *Compact Pro*,[5] developed for the Apple Macintosh operating system. Each of these products performs its compression in a single pass, with only one method selected per file. Thus, the possibility of heterogeneous files is ignored.

Unix `compress` uses an adaptive version of the Lempel–Ziv algorithm.[6] It operates by substituting a fixed-length code for common substrings. `compress`, like other adaptive textual substitution algorithms, periodically tests its own performance and reinitializes its string table if the amount of compression has decreased.

StuffIt makes use of two sets of algorithms: it first detects special-type files such as image files and processes them with algorithms suited for high-resolution color data; for the remaining files, it queries the operating system for the explicit file type given when the file was created, and uses this information to choose either the LZW variant of Lempel–Ziv,[4, 6] dynamic Huffman coding, or run-length encoding. This is a much more limited selection process than what we have implemented. Additionally, no selection of compression methods is attempted within a file. *Compact Pro* uses the same methodology as *StuffIt* and `compress`, but incorporates an improved Lempel–Ziv derived directly from LZ77.[11] The public-domain version of *StuffIt* is derived from Unix `compress`, as is evident from the similarity of their performance results.

* For purposes of comparison, the block sizes tested by Toal were nearly identical to those used in our system (ranging upwards from 4K).

NetApp; Rackspace Exhibit 1014 Page 138

Compression systems such as *StuffIt* perform simple selection among alternative compression algorithms. The important problem is that they are underequipped for the task of fitting a specific technique to each file (even when the uncompressed data is homogeneous). *StuffIt* uses few heuristics, since its algorithms are intended to be 'multipurpose' . Furthermore, only the file type is considered in selecting the algorithm – that is, no measures of redundancy are computed. Earlier versions of *StuffIt* (which were extremely similar to Unix `compress`) used composite alphabetic and textual compression, but made no selections on the basis of data characteristics. The chief improvements of our heterogeneous compressor over this approach are that it uses a two-dimensional lookup table, indexed by file properties and quantitative redundancy metrics, and – more important – that it treats the file as a collection of heterogeneous data sets.

THE HETEROGENEOUS COMPRESSOR

Our heterogeneous compressor treats a file as a collection of fixed size blocks (5K in the current implementation), each containing a potentially different type of data and thus best compressed using different algorithms. The actual compression is accomplished in two phases. In the first phase, the system determines the type and compressibility of each block. The compressibility of each block of data is determined by the values of three quantitative metrics representing the alphabetic distribution, the average run length and the string repetition ratio in the file. If these metrics are all below a certain threshold, then the block is considered fully compressed (uncompressible) and the program continues on to the next block. Otherwise, using the block type and largest metric, the appropriate compression algorithm (and possible heuristic) are chosen from the compression algorithm database. The compression method for the current block is then recorded in a small array-based map of the file, and the system continues.

The second phase comprises the actual compression and an optimization that maximizes the size of a segment of data to be compressed using a particular algorithm. In this optimization, which is interleaved with the actual compression, adjacent blocks for which exactly the same method have been chosen are merged into a single block. This merge technique maximizes the length of segments requiring a single compression method by greedily scanning ahead until a change of method is detected. Scanning is performed using the array map of the file generated when compression methods were selected from the database. A compression history, needed for decompression, is automatically generated as part of this phase.

The newly compressed segments are written to a buffer by the algorithm, which stores the output data with the compression history. The system then writes out the compressed file and exits with a signal to the operating system that compression was successful.

From this two-pass scheme it is straightforward to see why this system is less susceptible than traditional adaptive systems to biases accrued when the data type changes abruptly during compression. Adaptive compressors perform all operations myopically, sacrificing the ability to see ahead in the file or data stream to detect future fluctuations in the type of data. As a result, adaptive compressors retain the statistical vestiges of the old method until these are 'flushed out' by new data (or balanced out, depending upon the process for paging and aging internal data structures such as dictionaries). Thus adaptive compressors may continue to suffer the effects of bias, achieving suboptimal compression. On the other hand, by abruptly changing compression algorithms, our technique completely discards all remnants of the 'previous' method (i.e. the algorithm used on the preceding segment). This

NetApp; Rackspace Exhibit 1014 Page 139

allows us to immediately capitalize on changes in data. In addition, merging contiguous blocks of the same data type acquires the advantage of incurring all the overhead *at once* for switching to what will be the best compression method for an entire variable-length segment. The primary advantage of adaptive compression techniques over our technique is that the adaptive compression algorithms are 'online' (single-pass). This property increases compression speed and, more important, gives the ability to compress a data stream (for instance, incoming data packets in a network or modem transmission) in addition to files in secondary storage or variable-length buffers.

The remainder of this section presents the system. We begin with a description of the calculation of the block types and the redundancy metrics. We also explain the use of the metrics as absolute indicators of compressibility, and then describe the compression algorithms used and the structure of the database of algorithms. A discussion of implementation details concludes the section.

Property analysis

The compressibility of a block of data and the appropriate algorithm to do so are determined by the type of data contained in a block and the type of redundancy (if any) in the data. These two properties are represented by four parameters: the *block type*, and the three *redundancy metrics*. The block type describes the data in the block – text, binary, graphical, etc. The three redundancy metrics are the degree of variation in character frequency, average run length in the file, and the string repetition ratio of the file. They provide a quantitative measure of how compressible the block is and which type of redundancy is most evident in the block. The use of both quantitative redundancy measures (redundancy metrics) and qualitative characteristics (block types) as indicators for data compressibility is advocated by Held[7] and Salton.[19] We have refined the process for computing those attributes referred to as *datanalysis results* by Held[7] and as *statistical language characteristics* by Salton[19] to obtain an actual guide for compression. The remainder of this section describes how these four parameters are determined for each block.

Block types

The *block type* describes the nature of a segment of input data. There are ten classifications of data in this system: ANSI text, non-natural language text (hexadecimal encodings of binary data), natural language text, computer source code, low redundancy binary, digitized audio, low resolution graphics, high-resolution graphics, high-redundancy binary executable, and binary object data. ANSI text is composed of characters from a superset of the ASCII alphabet. Non-natural language text contains primarily ASCII text but does not follow a distribution of characters like that of human languages. Examples are computer typesetting data, *uuencoded* and *BinHex* encoded data (which has the same character distribution as binary data but is converted to text for ease of transmission). Natural language text includes text written in English as well as other languages which are representable by the Roman (ASCII) alphabet. Most European languages (including the ones using the Cyrillic alphabet), special symbols excluded, fall into this category, as do the Pinyin and Katakana romanizations of the Chinese and Japanese languages (as opposed to their digital encodings). Computer source code uses the ASCII alphabet but characters are distributed with a different frequency than in natural language text. Low-redundancy binaries usually contain compressed data, but may also include data which is merely difficult to compress. Audio

NetApp; Rackspace Exhibit 1014 Page 140

data are very high in redundancy; audio files (and audio segments of multimedia files) are usually extremely large. Low-resolution graphics have long runs of contiguous repeated bits but unlike high-resolution graphics are not suited to lossy compression. High-resolution graphics include color and grayscale and may be compressed with lossy methods. Binary executables, like low-resolution graphics, have long runs of contiguous repeated bits and comprise all compiled programs on a computer system. Finally, object data has slightly shorter runs but is similarly redundant.

To determine the block type we use a procedure `new-file` which is our extension of the Unix `file` command.[20] `file` works by examining the first 512 bytes of a file and comparing the pattern of data contained in it to a collection of known data patterns from Unix and other operating systems. `new-file` works in a similar fashion, with two modifications. First, it examines and compares not only the first 512 bytes of a data set, but also 512 bytes in the middle of the set and the 512 bytes at the end (if they exist). This provides a better indication of the *primary* data type of a file by taking into account the possibility that the properties may change anywhere within the file. Thus, `new-file` decides on the 'most applicable' data type by a majority vote (or the first data type detected in the case of a three-way tie). The other change is that the known patterns of data have been increased by adding three graphics patterns.

Redundancy metrics

The redundancy metrics are quantitative measures that are used to determine the compressibility of a block of data. They are: the *degree of variation in character frequency* or *alphabetic distribution*, M_{AD}; the *average run length* of the block, M_{RL}; and the *string repetition ratio* of the block, M_{SR}. In general, these three manifestations of redundancy are independent. Each of the redundancy types is exploited by different compression algorithms. *Frequency of characters* is exploited by arithmetic or alphabetic encoding algorithms. In arithmetic coding data is represented by an interval that is calculated from the probability distribution of data. With alphabetic coding algorithms such as the Huffman[21] and Shannon–Fano[22] algorithms, more frequently occurring characters are replaced by shorter units than the less frequently occurring characters. *Contiguous strings*, long strings of identical units occurring next to one another, are exploited by run length encoding algorithms.[23] In these algorithms, contiguous strings are replaced by a single occurrence of the string, called a *run*, plus a count of the number of identical strings following. Both alphabetic distribution and average run length are sometimes characterized as statistical redundancy metrics.[24] *Recurrent strings*, which occur repeatedly in the input stream with any number of interleaved symbols, are exploited by *textual substitution algorithms* such as Lempel–Ziv.[6,11,12] In these algorithms, recurrent instances are replaced with positional references (pointers) to the original instance.

Experimental evidence for the efficacy of quantitative redundancy measures is described in texts by Storer[1] and Shannon.[22] Shannon provided an estimate of the entropy of English text, approximately bounding it to be between one and two bits per character.[22] This was determined experimentally by presenting fragments of (unfamiliar) English text to human subjects and recording the frequency with which they guessed unknown letters. The fragments were revealed character by character, so that letters at the end of long or uncommon words were easiest to guess and letters at the beginnings of words were hardest. The observation that binary executables are known to possess high average run lengths is found in Storer.[1] However, this property is rarely exploited or measured.

NetApp; Rackspace Exhibit 1014 Page 141

Each redundancy metric is calculated by a separate statistical sampling routine and normalized using a gamma distribution function G to be a number between 0 and 10 so as to simplify comparison among the different metrics. The gamma distribution was chosen because the graph of each of the unscaled redundancy metrics for a test set of 50 files, when plotted on a histogram, approximated a gamma distribution. Normal and χ^2 distributions were also considered, but these proved to be too specific for the application (since they are both specific parametric cases of the gamma distribution). The gamma distribution is defined as follows (cf Ross[25]):

$$\begin{aligned} G_\tau(x_\tau) &= \int_0^{x_\tau} f_\tau(x)\,\mathrm{d}x \\ f_\tau(x) &= \frac{\lambda_\tau \mathrm{e}^{-\lambda_\tau x}(\lambda_\tau x)^{t_\tau - 1}}{\Gamma(t_\tau)} \\ \Gamma(t_\tau) &= \int_0^{\infty} \mathrm{e}^{-y} y^{t_\tau - 1}\,\mathrm{d}y \end{aligned}$$

where f_τ is the density function, Γ is the gamma function, x_τ is the unnormalized measure, t_τ is the shape parameter for the gamma distribution, and λ_τ is the scale parameter for the gamma distribution. The τ subscript simply represents the redundancy type under consideration, i.e. AD, RL, or SR, respectively. The shape and scaling parameters, t_τ and λ_τ respectively, were determined by fitting the best gamma distribution curve to the data set. This was done by performing the preferred compression method for each file and tabulating the induced ratio among normalized metrics to yield the desired parameter values for each segment. These were then averaged to obtain the empirical scaling parameters.

The *alphabetic distribution metric* (the degree of variation in character frequency) of a block is calculated by taking the population (root-mean-square) standard deviation of the ordinal values of characters in the block and dividing it by the block length (in bytes). The M_{AD} metric is calculated by the following formulas:

$$\begin{aligned} M_{\mathrm{AD}} &= 10 * G_{\mathrm{AD}}(x_{\mathrm{AD}}) \\ x_{\mathrm{AD}} &= \frac{\alpha}{\text{block length in bytes}} \\ \alpha &= \sqrt{\frac{\sum_{c \in \text{charset}} (c - \mu)^2}{256}}, \end{aligned}$$

where c is the ordinal value of a character and μ is the average ordinal value of all characters in a block. The normalization uses $t_{\mathrm{AD}} = 1.70$ and $\lambda_{\mathrm{AD}} = 53.0$ as parameters.

The *average run length metric* is obtained by dividing the number of bits in a block by the number of runs. A run is defined to be a repetition of symbols (either bits or bytes). Our implementation takes both bitwise and bytewise run lengths. For example, if $f = 0001111001110000$ is a file of 16 bits, then the number of bit runs is 5, and the number of byte runs is 2. The scaled metric M_{RL} is obtained by:

$$\begin{aligned} M_{\mathrm{RL}} &= 10 * G_{\mathrm{RL}}(x_{\mathrm{RL}}) \\ x_{\mathrm{RL}} &= \frac{\text{file length in bits}}{\text{number of runs}} \end{aligned}$$

NetApp; Rackspace Exhibit 1014 Page 142

with gamma distribution parameters $t_{RL} = 0.50$ and $\lambda_{RL} = 12.0$.

The *string repetition ratio metric* is the total number of n-bit strings in the block divided by the number of distinct n-bit strings (up to 100K). In our implementation, n is 32, the word size of our machine. The normalized metric M_{SR} is obtained by:

$$M_{SR} = 10 * G_{SR}(x_{SR})$$

$$x_{SR} = \frac{\text{number of } n \text{ bit strings}}{\text{number of distinct } n \text{ bit strings}}$$

with gamma distribution parameters $t_{SR} = 0.18$ and $\lambda_{SR} = 0.2$.

The alphabetic distribution and average run length metrics can be calculated in linear time. The string repetition ratio can be computed in $O(n \log n)$ time using a dictionary data structure. For simplicity, and because a (small) constant amount of data is scanned, we use an $O(n^2)$ version. New strings are stored in an array rather than a binary tree, which would require more insertion overhead (and is not worth while for the 5K block length used in the current system). Our routine integrates $f_\tau(x)$ by Simpson's Rule with $n = 10$ intervals.

The largest of the three metrics is assumed to represent the most significant type of redundancy present in the block. It is expected that compression will decrease at least one of the metrics, and experiments conducted on a wide variety of files have proven this convention to be reliable. Experiments have also shown that if all the normalized metrics are smaller than 2.5, the file is considered not compressible, and the system records a verdict of 'uncompressible' on the current block. If at least one of the parameters is greater than 2.5, the file is considered compressible. The maximum of the normalized metrics is then selected and used in conjunction with the file type to select the appropriate compression algorithm from the lookup table described in the following section. A negative compressibility test does not always imply that all three metrics are below the threshold. In some cases, the only redundancy type for which a metric is above the threshold accesses a null entry in the database of compression algorithms. This is interpreted as a decision that the (poor) potential for compression is outweighed by the overhead of executing the compression algorithm.

The algorithm and heuristic database

The compression algorithms and attendant heuristics are organized into the 10 by 3 table illustrated by Table I. The 10 file descriptors are the row indices and the 3 metrics are the column indices. Each entry of the table contains descriptors which are used to access the code for an algorithm-heuristic pair. It should be noted that four of the entries are blank (indicated by an *). A blank entry indicates that the combination of block type and highest metric are very unusual. In this case, the next highest metric is used instead, provided that it is above the threshold. As an example of using this table, consider a high-redundancy binary executable file whose highest metric is the string repetition metric M_{SR}. Together, this pair indicates that the Lempel–Ziv compression algorithm with the Freeze heuristic will be used.

The algorithms

There are four basic algorithms used by the system: arithmetic coding,[26] Lempel–Ziv,[8] run length encoding (RLE),[23] and JPEG for image/graphics compression.[27]

Arithmetic coding algorithms compress data by representing that data by an interval of

NetApp; Rackspace Exhibit 1014 Page 143

Table I. Database of compression algorithms†

	M_{AD}	M_{RL}	M_{SR}
ANSI	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
hexadecimal	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
natural language	arithmetic coding	*	Lempel–Ziv
	*	*	freeze
source code	arithmetic coding	run-length encoding	Lempel–Ziv
	*	n-bit run count	freeze
low redundancy	*	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	*
audio	*	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze
low resolution	*	run-length encoding	Lempel–Ziv
graphic	*	n-bit run count	freeze
high resolution	JPEG	run-length encoding	JPEG
color graphic	improved Huffman	n-bit run count	improved Huffman
high redundancy	arithmetic coding	run-length encoding	Lempel–Ziv
binary	*	n-bit run count	freeze
object	arithmetic coding	run-length encoding	Lempel–Ziv
	*	byte-wise encoding	freeze

† Note: the first line of each entry is the basic algorithm and the second line is the heuristic. An * as the heuristic indicates that no heuristic is used. Two * indicates no entry.

real numbers between zero and one. The width of this interval is inversely proportional to the number of symbols encoded, and the decrease in width is directly proportional to the frequency of the original symbols. Thus the interval specifies the encoded message via its bounds, with the precision (distance) of these bounds reflecting the information content of the message. The end result is that arithmetic coding achieves, in practice, much better space savings than Huffman coding and its dynamic implementations because of its higher likelihood of actually achieving the theoretical lower bound.[24, 28] Although early arithmetic coding algorithms performed too slowly to be of practical use,[29] the implementation of the Witten–Neal–Cleary algorithm used here[26] is optimized for speed – at some cost in space savings, but without giving up its advantage over dynamic Huffman coding. The reader is referred to Bell *et al*[24] for a thorough overview of arithmetic coding. We should note that in earlier implementation of the heterogeneous compressor we used a dynamic Huffman algorithm instead of arithmetic coding. We changed our implementation when we found that then Witten–Neal–Cleary algorithm[26] outperformed our implementation of dynamic Huffman coding[10, 30] in both space savings and execution time.

Run length encoding (RLE) algorithms compress data by replacing contiguous occurrences of a single-unit symbol (either bit or byte) by an efficiently coded count of these runs, usually a single occurrence of the symbol and the number of occurrences. We have implemented a straightforward RLE algorithm for our database, based on the description in Sedgewick.[23] In addition, bitwise and bytewise encoding are available as heuristics and the parameters of bitwise RLE are based on the RL metric.

Files with a high degree of string repetition are compressed using the Lempel–Ziv compression algorithm. It compresses data by replacing frequently occurring strings (with min-

NetApp; Rackspace Exhibit 1014 Page 144

imal regard of how far apart they occur) with compact pointers to the position of the first occurrence. Our implementation is a straightforward array-based encoding with constant-length codes. The algorithm maintains a dictionary of recurring strings in order to do the compression. In our system, the Lempel–Ziv algorithm is augmented with the *Freeze* heuristic. This heuristic suppresses paging of strings in the dictionary after it has been filled; that is, it prevents the replacement of previously encountered strings, regardless of how long ago or how infrequently the string has been encountered. *Freeze* is primarily a speed optimization, since it requires less computation than paging heuristics such as least recently used (LRU) or least frequently used (LFU), but it has been shown to work well for all but the least string-redundant files (including both binary executables and most text files). For files with extremely low string-repetition, our system usually selects Huffman compression.

The compression of high-resolution graphics and audio files uses a *lossy* compression scheme. Appropriately used, lossy algorithms guarantee that the decompressed file is similar enough to the original as to be nearly indistinguishable by human perception, and that repeated compression and decompression leads to limited cumulative 'damage' . The primary benefit of lossy compression is that it guarantees much higher compression ratios at a minimal tradeoff. For instance, a very-high-resolution color image can be compressed with much higher savings (possibly 95 per cent) if the user allows a small amount of noise, always less than 1 per cent per compression, to be introduced during each compression. Our system uses the JPEG system[27] for compressing high-resolution color and grayscale images. JPEG, which is divided into lossy and lossless parts, typically achieves compression ratios of between 15-to-1 and 25-to-1. The *potential* for this substantial savings is obtained by the Discrete Cosine Transform portion of the algorithm, a lossy method. This determines a limit on the amount of savings that can then be achieved by any lossless compressor. The actual savings are realized by a lossless portion, known as the *back end* which is applied to the preprocessed image data. The implementation of this module used in our system[27] is a Huffman coder. It is independent of the lossy front end and can be replaced with a run-length or textual-substitution based algorithm, to be selected by the synthesis system. In our implementation, we chose to retain the original Huffman back end, a different algorithm from the general-purpose dynamic Huffman coder which we also studied.[10, 30] This is because the JPEG Huffman coder is especially suited to the redundancy remaining after lossy preprocessing. It is worthy of mention that the JPEG developers have investigated the use of arithmetic coding back ends, which were found to be experimentally superior but were not used because of proprietary considerations.[27]

Implementation

The system consists of a driver module, four block analysis modules, and the synthesis module, which includes the database of compression algorithms. All modules are written in C and were tested on a Unix platform. The program uses a data directed style of implementation for choosing the compression algorithm to apply to a block. Thus, additional block types, compression algorithms and heuristics, and redundancy metrics can be added to the system with minimal modification of the source code. Only the database would have to be updated and the block analysis routines extended; the rest of the program would remain the same.

The driver performs two iterative passes through the file. It first performs block analysis on the file one 5K block at a time. This block size was chosen after experimentation showed that the response of the system to changes in block type became roughly stable as block

NetApp; Rackspace Exhibit 1014 Page 145

size exceeded 5K (i.e., did not significantly increase as block size did), and that a block size of 5K yielded highly accurate metrics (in only 1 of the 20 test files did the heterogeneous compressor select a suboptimal algorithm for any block). Finally, we found that the highest level of adaptivity without a noticeable *decrease* in accuracy was achieved at 5K, hence our choice of 5K as the block size.

For each block, the system invokes the four analysis modules – three for metric computation and normalization and one to determine the file properties – and stores their output. It then performs the metric comparison and combines the results with the file property to complete the table lookup for the current block. An identifying tag for the selected algorithm is written to the 'compression plan' , an array which stores one complete compression instruction per block (if the current block is deemed uncompressible, a 'skip' instruction is recorded).

We pause here to discuss the normalization of the metrics. Originally, we used a naive normalization method: direct algebraic scaling with experimentally determined constants for each metric. This did not, however, accurately reflect the statistical relationship between variance in character frequency and alphabetic redundancy. Also, the behavior of these functions at asymptotes led to poor approximation of the overall distribution of data segments in the test files. The result was that arithmetic coding was too often incorrectly chosen, resulting in inferior compression; and selection approached randomness as metric values for both string repetition and alphabetic distribution tended toward extreme values. Using the gamma normalization method described above resulted in an improvement in the selection of arithmetic coding. Among the 20 benchmark files, arithmetic coding was selected as the compression method in exactly those cases where the other methods performed worse.

The second pass performs the compression of each block. In order to improve performance, this pass includes a simple optimization step which circumvents the overhead of restarting compression after each fixed length block by merging contiguous blocks that are to be compressed using the same compression algorithms.

On this same pass through the file, the system compresses each of the newly merged blocks using the algorithm recorded in the compression plan. The compressed data is written to an output buffer, while the compressed length (which indicates where in the compressed file a compressed block begins and ends) and compression method are recorded in a separate history for reference at decompression time. If negative compression or no compression is achieved, or if the block was already marked uncompressible, then the data is copied directly to the output buffer (the full block length and a code for 'no compression' are recorded in the compression history). Upon reaching the end of the blocks, the system writes out the compressed data from the output buffers and prepends the encoded compression history to produce the final output file.

When decompression is invoked, the driver module opens the compressed file, interprets the history tag and performs the necessary operations. The tags are a stored version of the compression history in compact, encoded form. Since the heterogeneous system generates different compression sequences for each file, and since the length of a compressed block varies with both the length of the original block and the compression method used, these tags are necessary to guide the decompression process. Currently only the compressed lengths of each block and the method of compression are stored, but a checksum for the original (decompressed) block length can be added with negligible overhead. When executed in reverse order on each compressed block, the instructions in the history tags result in the original file. For simplicity and security, they are prepended to the compressed file (and can easily be encrypted).

NetApp; Rackspace Exhibit 1014 Page 146

EXPERIMENTAL RESULTS

Design and construction of the test files

To test the overall performance, the system was run on a set of 20 test files. These files range in length from approximately 39K to 366K, with representative files from each of the ten block types included in the test corpus.

The test files are designed to model certain types of heterogeneous files, including utilities for image viewing, business, or audio processing, and hypothetical multimedia databases and programs. To construct these files, a collection of 30 files from the Unix, Apple Macintosh, and MS-DOS (IBM PC) operating systems was created. These files are listed in Table II. To create the test corpus, they were concatenated in groups of 2 or 3. The resultant series of test files is listed in Table III. All of the source files were used. The goal was to generate as broad a range of permutations as possible (while restricting the generated files to those which are likely to exist in a typical user environment). This was performed manually with consideration toward combinatorial constraints and the criteria of realistic data modeling. Since all of the files in the source collection originate from common commercial sources or from public archives (with the exception of the source and object files, which are from the code for the heterogeneous compressor itself), the latter constraint was considerably simplified.

The assembled files were then ported to the test sites (a Sun workstation for Unix `compress` and our heterogeneous compressor; a Macintosh for *StuffIt* and *Compact Pro*; and an IBM 80486 machine for PKZIP). Binary file transfer mode was used to ensure that the file lengths agreed exactly among all platforms.

Performance

In this section, we review and analyze the performance of the heterogeneous compressor with respect to compression savings, as compared with four of the commercial systems previously discussed; and execution time. Finally, we briefly note the implications of running the experiments and compiling performance data on several different architectures.

Compression savings

The total length of the uncompressed benchmark suite is just under three megabytes. Table IV shows the compressed length achieved by Unix `compress`, PKZIP, *StuffIt*, *Compact Pro* and the heterogeneous compression system. The heterogeneous compressor achieved the greatest compression, with a total compressed length of 1828K. This represents an additional savings of 162K (more than eight per cent) over the best commercial system (*Compact Pro v1.32*), and 339K (nearly 16 per cent) over the average. Compressed lengths for the commercial methods ranged from 1990K to 2375K.

Table V compares the percentage savings obtained by our system to the savings obtained by the commercial programs and the heterogeneous system. The last two columns show the difference in per cent saved between the synthesis system and the best and average of the four commercial packages. The best commercial compressor is marked for each of the files. Note that the heterogeneous compressor does better than all commercial programs in 19 of 20 cases and better than three of the four commercial systems in this one case (file 15). The difference in compression for this file is only 0.02 per cent, whereas for all the other files, the heterogeneous compressor has at least a 1.3 per cent improvement over the best

NetApp; Rackspace Exhibit 1014 Page 147

Table II. Files used to compose the test suite and their respective origins

File designation	File name	File type
audio1	cosby.snd	SoundMaster Macintosh audio file
lowrd1	ticker.txt	ASCII characters from stock ticker
lowrd2	exsound	compressed World Builder sound library
lowrd3	huff	compressed Unix executable
lowrd4	appnote.uue	uuencoded text
text1	phrack.txt	English text
text3	techbook.txt	Unix news article
text4	quanta1.txt	English text
text5	attilla.fluff	English text
text6	shadow.fluff	English text
text7	quanta2.txt	English text
execu1	ad	Unix executable
execu2	sh	Unix executable
execu3	blob	Silicon Graphics executable
execu4	zero	Silicon Graphics executable
execu5	network2.exe	IBM PC executable
execu6	hostname	Unix executable
graph1	compmisc.drw	Lotus Freelance line drawing
graph2	compperi.drw	Lotus Freelance line drawing
graph3	computer.drw	Lotus Freelance line drawing
graph4	lowres.mpt	MacPaint file
graph5	3dbar.drw	Lotus Freelance 3-D bar chart
graph6	image.ppm	PPM (high-resolution image) file
graph7	grp4	MacPaint file
objec1	test1.o	Unix object file
objec2	test2.o	Unix object file
objec3	test3.o	Unix object file
source1	table.c	C source code
source2	freeze.c	C source code

commercial compressor. The average of each column appears in the bottom row; note that the 'percent difference' averages are not weighted by file length, as each file is considered a separate experiment.

Because the quality of compression by the synthesis system depends on that of the algorithms and heuristics used, improvement of the implementations that we use should yield higher performance. This is evidenced by comparing the results of compressing a file dominated by string repetitions by Unix `compress` and *Compact Pro*. Both are implementations of the Lempel–Ziv algorithm. Unix `compress` has no heuristics, whereas *Compact Pro* is a better implementation of LZ77.[5, 11] *Compact Pro* consistently outperforms `compress`. It should be noted that the performance of the *Freeze* variant of Lempel–Ziv[8] used in our sys-

NetApp; Rackspace Exhibit 1014 Page 148

Table III. Combinations of the test files and the resultant simulated data types

File number	File composition	Classification of data modeled
1	text1 — lowrd1	news or stock report
2	graph7 — objec1	object file for a graphics viewer
3	lowrd1 — text3 — graph4	multimedia application (text/graphics)
4	graph7 — execu3	graphics viewer
5	audio1 — graph1	multimedia data file (sound/graphics)
6	text2 — lowrd1 — graph3	multimedia data file (text/graphics)
7	lowrd3 — execu1	commercial utility
8	graph2 — lowrd2 — execu2	multimedia application (graphics/sound/executable)
9	source1 — lowrd3 — graph6	multimedia data or source file (source/compressed binary/image)
10	audio1 — text4	multimedia data file (sound/text)
11	lowrd1 — execu4	statistical application with data
12	graph7 — text5	multimedia data file (text/graphics)
13	lowrd2 — text6	multimedia data file (sound/text)
14	text3 — audio1 — graph5	multimedia data file (text/sound/graphics)
15	lowrd1 — text4 — source2	source file for multimedia program (text/source code)
16	text7 — lowrd2 — graph3	multimedia data file (text/compressed audio/graphics)
17	graph4 — audio1 — execu5	multimedia application (sound/graphics)
18	execu4 — graph7 — text4	multimedia application (graphics/text)
19	objec3 — lowrd3 — execu6	commercial utility
20	objec2 — audio1 — execu2	audio application

tem does consistently better than `compress` and is comparable to *Compact Pro* on standard industrial benchmarks.[9] Improving algorithms and adding or substituting new heuristics would also yield more savings.

Execution times and speed optimizations

In this section we compare, in *approximate units*, the running time of the heterogeneous compressor against those of the four commercial systems the savings rates of which for our test files are documented above. The units are approximate for two reasons. First, because the four test systems are commercial the source code for three of them is not publicly available*, which renders an exact measure of *user* time infeasible. This concern is in part assuaged by the non-multitasked, single-user nature of the microcomputer operating systems on which three (`compress` for Linux notwithstanding) of the commercial systems reside. Second, however, the drastic architectural and organizational differences among the various native machines renders uniform comparisons unreliable. This applies even to normalized execution times because the host machines differ not merely in clock cycle speed, but in instruction set architecture and dynamic instruction frequencies for similar compression algorithms. The *exact* running times reported in this section is only that of the heterogeneous

* As noted, however, the Lempel–Ziv implementation employed by *StuffIt Classic* is nearly identical to that of Unix `compress`.

NetApp; Rackspace Exhibit 1014 Page 149

compressor. These comprise the non-commercial* compression systems for which source code is available for profiling. For the commercial systems we report the observed wall clock time to provide a standard of comparison, but note that the host machines vary in computational power.

Table IV. Results of the four popular commercial programs and the heterogeneous compression system, applied to the 20 test files

File number	Original length	Unix `compress`	PKZIP v1.10	*StuffIt Classic*	*Compact Pro* v1.32	Heterogeneous compressor
1	39,348	20,578	17,119	20,575	16,831	16,315
2	44,202	44,202	39,813	40,412	41,112	37,388
3	46,629	46,629	46,629	43,261	40,367	36,477
4	59,254	52,076	40,571	45,202	41,607	38,007
5	169,108	168,903	151,478	149,701	148,917	134,524
6	100,476	69,771	53,043	65,417	52,349	50,906
7	131,663	131,663	103,544	106,643	109,979	96,429
8	220,644	190,971	137,886	173,677	137,401	127,384
9	301,805	145,993	112,503	137,685	115,096	103,730
10	255,306	204,457	191,378	206,193	183,313	168,675
11	59,305	30,178	22,782	29,701	22,858	21,774
12	51,715	51,715	43,032	46,462	44,107	40,229
13	63,189	63,189	58,247	59,569	59,934	54,481
14	196,789	176,276	196,789	172,486	151,057	137,052
15	148,908	73,555	63,748	75,595	64,618	63,778
16	164,535	141,067	132,992	135,245	110,093	104,175
17	203,912	203,912	184,657	189,398	202,821	170,564
18	200,640	128,675	107,728	125,461	104,711	101,674
19	366,557	265,114	198,727	265,027	198,756	187,659
20	278,152	223,277	193,980	224,943	191,763	181,030
Total	3,102,137	2,432,201	2,096,646	2,312,653	2,037,690	1,872,251

The running times for the commercial systems on the entire test suite documented above appear in Table VI. All of the execution times are measured in wall clock units except for the heterogeneous compressor's, which is a total of user times as reported by `prof`, the C profiler under Unix. The wall clock time was empirically observed not to differ noticeably from this total on an unloaded Unix machine. The commercial systems were similarly tested on unloaded (or single-task) systems.

For Unix `compress`, the mean running time was 26 s, where the average was taken over runs on different Sun workstations of comparable power (documented below). A Unix implementation of PKZIP was also tested on one of these Sun workstations, and achieved an execution time of 56 s – only slightly better than the personal computer version. The running time of 856 s placed the heterogeneous compressor in the middle to high end of the commercial compressors in terms of running time.

* For this purpose we continue to consider Unix `compress` commercial, due to its wide range of versions.

NetApp; Rackspace Exhibit 1014 Page 150

Table V. Percent savings for the test compression systems*

File number	Unix `compress` (% saved)	PKZIP v1.10 (% saved)	*StuffIt Classic* (% saved)	*Compact Pro* v1.32 (% saved)	Heterogeneous compressor (% saved)	Best win (% diff.)	Average win (% diff.)
1	47·70	56·49	47·71	57·23*	58·54	1·31	6·25
2	0·00	9·93*	8·57	6·99	15·42	5·49	9·04
3	0·00	0·00	7·22	13·43*	21·77	8·34	16·61
4	12·11	31·53*	23·71	29·78	35·86	4·33	11·57
5	0·12	10·43	11·48	11·94*	20·45	8·51	11·96
6	30·56	47·21	34·89	47·90*	49·34	1·44	9·20
7	0·00	21·36*	19·00	16·47	26·76	5·40	12·55
8	13·45	37·51	21·29	37·73*	42·27	4·54	14·77
9	51·63	62·72*	54·38	61·86	65·63	2·91	7·98
10	19·92	25·04	19·24	28·20*	33·93	5·73	10·83
11	49·11	61·59*	49·92	61·46	63·28	1·70	7·77
12	0·00	16·79*	10·16	14·71	22·21	5·42	11·80
13	0·00	7·82*	5·73	5·15	13·78	5·96	9·11
14	10·42	0·00	12·35	23·24*	30·36	7·12	18·85
15	50·60	57·19*	49·23	56·61	57·17	−0·02	3·76
16	14·26	19·17	17·80	33·09*	36·69	3·60	15·60
17	0·00	9·44*	7·12	0·54	16·35	6·91	12·08
18	35·87	46·31	37·47	47·81*	49·33	1·51	7·46
19	27·67	45·79*	27·70	45·78	48·80	3·02	12·07
20	19·73	30·26	19·13	31·06*	34·92	3·86	9·87
Average	19·16	29·83	24·21	31·55*	37·14	4·35	10·96

* The starred entry in each row is the best commercial system.

CONCLUSIONS

Analysis of results

This project was successful on several levels. First, the feasibility of synthesizing compression plans from encapsulated primitives for heterogeneous files was illustrated. The use of property analysis and redundancy metrics was experimentally successful, the latter verifying the applicability of statistical data analysis to automatic programming in this domain. The positive test results obtained with the primitive database currently available would probably be even better with improved implementations of the algorithms and heuristics. The statistical foundations of the heterogeneous system proved strong enough to be of definite relevance to the operating systems community, and might be useful in an information theoretic context. The benefits of data compression are ubiquitous in that savings through compression are independent of hardware and storage capabilities; selective techniques increase these savings by a significant factor for heterogeneous files.

Future work

The sampling method may be improved in future implementations by randomization. The increase in analysis accuracy that this would bring would demand more primitives and heuristics – such need would arise in any case with the continuing development of new files types, such as high-resolution animation and three-dimensional images.

NetApp; Rackspace Exhibit 1014 Page 151

Table VI. Execution times of the heterogeneous and commercial compressors

Compression system	Execution time (s)	Execution time (min)
Unix `compress`	≈ 26	0:26
PKZIP v1.10	67	1:07
StuffIt Classic	1152	19:12
Compact Pro v1.32	1594	26:34
Heterogeneous compressor	856	14:56

In the current system, lossy compression methods can be applied only if an entire file is found to be of a lossily compressible data type. Typically, these include high-resolution images (for JPEG) and speech, general high-definition audio, and high-resolution animation files. A special case could be implemented specifying that when an entire file matching a single lossily compressible data type (i.e. a homogeneous loss-permissible file) is found, the lossy algorithm may be applied.

The difficulty is that without explicit information on where loss-permissible portions of a heterogeneous (e.g. multimedia) file begin and end, the compressor cannot absolutely guarantee that no data will be distorted which the user is not willing to have distorted. Thus no lossy methods can be safely applied to any *segment* in the block-based system. Thus a heterogeneous system would require either full interactive guidance from a user who could inspect the file or knew its contents, or would require improved magic numbers which encoded the lengths of loss-permissible segments. The heterogeneous system could then scan for these codes during the property analysis phase and preempt or modify metric-based selection if a lossy algorithm is warranted. The latter approach seems far superior to interactive compression, which places an intolerable burden of responsibility on users (consider a multimedia file with hundreds of interspersed digitized photographs).

Another improvement worth considering is the use of a ratings system for specialized (especially lossy) compression algorithms such as JPEG and MPEG. For example, by designating RLE compression '0 per cent alphabetic distribution, 100 per cent run length, 0 per cent string repetition' and by defining its single-type counterparts similarly, a standard can be established. Unix `compress`, for instance, *might* rate '40 per cent AD, 0 per cent RL, 60 per cent SR' and a hypothetical algorithm X might rate '25 per cent AD, 50 per cent RL, 25 per cent SR' . The rating standard would correspond to the metric rating system for files which our system uses, and would help in analysis of the performance of composite compression techniques (which handle multiple redundancy types). Non-synthesized composite techniques exist, both adaptive and non-adaptive, though results are not as promising as those of automatically generated techniques.

Finally, it is clear from the frequency of duplicate entries in the algorithm lookup table that the database of primitives used in this heterogeneous system may not be as well-stocked as it optimally could be. Storer[1] lists a plethora of optional heuristics which are applicable to Lempel–Ziv compression, specifically in augmenting and deleting from the dictionary.

ACKNOWLEDGEMENTS

This paper was produced as part of a research project at Johns Hopkins University. We are grateful to the faculty and staff of the JHU Computer Science Department, and to the Brown University CS Department, for their assistance throughout this work.

NetApp; Rackspace Exhibit 1014 Page 152

We would like to thank Leonid Broukhis, Graham Toal, and Kenneth Zeger for discussions on some of the research reported here. We also thank Jonathan Eifrig, Bill Goodman, and Tom Lane for guidance on several technical issues. Finally, we thank the anonymous reviewers for their comments and suggestions, especially for introduction to relevant literature in arithmetic coding.

REFERENCES

1. James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.
2. Phillip W. Katz, PKZIP. Commercial compression system, version 1.1, 1990.
3. Sun Microsystems, compress. Commercial compression system, operating system version 5.3, September 1992.
4. Raymond Lau, StuffIt Classic and StuffIt Deluxe. Commercial compression system, 1990.
5. Bill Goodman, Compact Pro. Commercial compression system, v1.32, 1991.
6. Terry A. Welch, 'A technique for high performance data compression', *IEEE Computer*, **17**(6), 8–19 (1984).
7. Gilbert Held and Thomas R. Marshall, *Data Compression: Techniques and Applications: Hardware and Software Considerations*, 3rd edn, John Wiley and Sons, 1991.
8. Leonid Broukhis, Freeze implementation of LZHuf algorithm. comp.sources.misc archives, Internet, 1991.
9. Jean-Loup Gailly, comp.compression benchmark (Calgary test corpus). In comp.compression FAQ list, J. Gailly, (ed.), 1992.
10. Jeffrey S. Vitter, 'Dynamic Huffman Coding', *ACM Transactions on Mathematical Software*, (June 1989).
11. J. Ziv and A. Lempel, 'A universal algorithm for sequential data compression', *IEEE Transactions on Information Theory*, **23**,(3), 337–343 (1977).
12. J. Ziv and A. Lempel, 'Compression of individual sequences via variable-rate coding', *IEEE Transactions on Information Theory*, **24**(5), 530–546 (1978).
13. Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan and Victor K. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, 320–330 (April 1986).
14. Yooichi Tagawa, Haruhiko Okumura and Haruyasu Yoshizaki, LZHuf: encoding/decoding module for LHarc. Compression system, version 0.03 (Beta), 1989.
15. Haruyasu Yoshizaki, LHA: A high-performance file-compression program. Compression system, version 2.11, 1991.
16. Edward R. Fiala and Daniel H. Greene, 'Data compression with finite windows', *Communications of the ACM*, 490–505 (1989).
17. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, Maryland, second edition, 1987.
18. Graham Toal. Personal communication. Unpublished, 1992.
19. Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
20. Ian F. Darwin, file (program). Berkeley Unix operating system, 1987.
21. David A. Huffman, 'A method for the construction of minimum-redundancy codes', *Proceedings of the IRE*, number 40, 1952, pp. 1098–1101.
22. Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communications*, University of Illinois Press, Urbana and Chicago, 1963.
23. Robert Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
24. Timothy C. Bell, John G. Cleary and Ian H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
25. Sheldon Ross, *A First Course in Probability*, Macmillan Publishing Company, New York, third edition, 1988.
26. Ian H. Witten, Radford Neal and John G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**(6), 520–540 (1987).
27. Independent JPEG Group. 'JPEG image compression system', think.com FTP archives, Internet, 1994.
28. Jean-Loup Gailly. comp.compression/comp.compression.research FAQ list. J. Gailly (ed.), URL `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html`, 1994.
29. James A. Storer, *Image and Text Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.
30. Graham Toal. C implementation of dynamic Huffman compressor by J. S. Vitter. comp.source.misc archives, Internet, 1990.

NetApp; Rackspace Exhibit 1014 Page 153