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 com pression plans from encapsulated primitives for heterogeneous files was illustrated. The us of property analysis and redundancy metrics was experimentally successful, the latter veri fying the applicability of statistical data analysis to automatic programming in this domain The positive test results obtained with the primitive database currently available woul 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 def inite relevance to the operating systems community, and might be useful in an informatio theoretic context. The benefits of data compression are ubiquitous in that savings throug compression are independent of hardware and storage capabilities; selective techniques in crease 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 an heuristics – such need would arise in any case with the continuing development of ne 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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.