1990

Pipelining Data Compression Algorithms

R. L. Bailey

R. Mukkamala
Old Dominion University

Pipelining Data Compression Algorithms

R. L. BAILEY* AND R. MUKKAMALA**

** UNISYS Defense Systems, Virginia Beach Operations, Virginia Beach, VA 23452, USA*
*** Department of Computer Science, Old Dominion University, Norfolk, VA 23529, USA*

Many different data compression techniques currently exist. Each has its own advantages and disadvantages. Combining (pipelining) multiple data compression techniques could achieve better compression rates than is possible with either technique individually. This paper proposes a pipelining technique and investigates the characteristics of two example pipelining algorithms. Their performance is compared with other well-known compression techniques.

Received February 1990

1. INTRODUCTION

In everyday life, computers manipulate or use data in many forms such as programs, files, messages, and many other types of information. Nearly all data contains some redundant information. The objective of a data compression algorithm is to transform redundant data into a form that is smaller but still retains the same information. When data is compressed, a transformation occurs such that the various character strings of the data are replaced with one or more code words. This relationship can be implemented in various ways. Regardless of the method utilized, an effective compression algorithm must output fewer bits than are input.

Data compression can be useful for various data processing applications. Computer networks require data to be transmitted from one site to another. Data compression can reduce communication costs in computer networks by compacting messages before transmission. Data compression can also reduce the storage requirements of databases and file systems, and thereby increasing the effective capacity of storage systems.

In general, data compression techniques can be classified into two categories: static and dynamic (sometimes referred to as adaptive). Static data compression algorithms are effective when the frequencies of occurrence of characters (or strings) do not significantly change within a given data set.* When the input data sets do not exhibit this uniformity, static data compression algorithms cannot result in optical codes. Dynamic (or adaptive) data compression algorithms are designed to adapt to such nonuniformities in the input data by keeping track of the changes in character probabilities within a data set.

Currently, a number of static and dynamic data compression algorithms have been suggested in literature.[9] Each algorithm is shown to be effective in compressing data with certain characteristics (see Section 2 for details). There is no universal data compression algorithm that can optimally compress all data sets. In this paper, we suggest a new *pipelining* technique that combines two or more coding algorithms to compress data more effectively than the individual algorithms. Our current efforts represent some initial steps in constructing a universal data compressor using pipelined structures.The choice of the candidate algorithms as well as their ordering within the pipeline influence the performance of the proposed technique. In this paper, we present a summary of the results obtained by some pipelined algorithms. The initial efforts only considered pipelining of two algorithms.

The remainder of the paper is organized as follows. Section 2 examines several existing data compression algorithms. The rationale for the proposed pipelining technique and the criteria for the candidate algorithms' selection are discussed in Section 3. Section 4 presents the initial results obtained with some candidate algorithms. Fimally, Section 5 summarizes our observations and discusses proposals for future work.

* In this paper, a data set is defined as a sequence of characters. The likelihood of any given character occurring in a data set is referred to as a character probability.

2. DATA COMPRESSION ALGORITHMS

Static data compression algorithms were among the first attempts at reducing data size. Huffman coding[5] is among the best known static data compression algorithms. Huffman coding is implemented in a binary tree using a prefix coding scheme to assign variable length code words. Minimum redundancy is achieved by assigning the shortest codes to the most probable characters, while the longest codes are assigned to the least probable characters. Unless the character probabilities are known beforehand, two passes are required. The first pass analyzes the character probabilities and the second pass performs the actual compression. Huffman coding is typical of static compression techniques insofar as it is optimal only when the character probabilities do not vary within a given source.

Adaptive compression algorithms can dynamically respond to changes in the input source.[2] Some of these techniques have been modifications to static methods. Several adaptive variations of Hoffman coding have been devised.[3,4,8,13] These methods utilize counters to maintain the current probability of each character. The counters are used to dynamically modify the code mapping. With this technique, the more probable characters are moved closer to the root of the tree and therefore, receive shorter code words. Mäkinen[10] suggests using transpose and interval coding to maintain the code mapping in a list. With these methods, only one pass is needed to encode the data.

Ziv and Lemple[15,16] devised a coding scheme that was radically different from the Huffman style. Lempel-Ziv coding uses a parsing technique to dynamically encode the input source. This scheme parses strings of characters that do not exceed a prescribed length and builds a table to map these strings to fixed length code words. The more frequently occurring strings are grouped into longer strings which result in many characters being represented by a single fixed length code word. The length of the code word is dependent upon the size of the table used to contain the string/word code mapping. For example, a table size of 4096 requires a 12 bit code word. The code word is simply the table address of the corresponding string.

LZW coding is a variable of the Lemple-Ziv technique.[14] In Welch's implementation, the table is initialized with the character set and rather than containing strings of a prescribed length, contains fixed length ⟨code word, character⟩ pairs. The table is built by parsing off the longest recognized (in the table) string and using the subsequent character to form a new table entry. This allows a relatively small table to be utilized and provides high compression ratios for most inputs. The LZW algorithm has gained wide acceptance and is used in many data compression programs, such as Unix Compression[12] and the popular PC archiving utilities ARC[1] and PKPAK.[11] The wide use of LZW can be attributed to its speed, high compression ratios, and ease of implementation.

Another adaptive data compression technique was described by Jakobsson.[6] It is similar to the LZW algorithm insofar as it uses a parsing method. However, its dictionary is built with a *forest* of trees rather than a table. Strings of a predetermined length are parsed from the input and added to the dictionary trees. A separate tree is constructed for each character of the input set: all strings beginning with the character *a* are added to tree *a*, strings beginning with *b* are added to tree *b*, etc. As the dictionary forest is built, subsequently longer strings can be parsed off and encoded with the tree address.

A recent innovation in adaptive data compression involves the use of *splay* trees to encode characters.[7] With splay tree coding, characters are encoded in a manner very similar to adaptive Huffman coding. A variable length prefix code is constructed based upon the characters position in the tree. In order to produce an optimal code, the tree must be balanced. If the tree becomes unbalanced, then some characters will require more bits for encoding than would be required if the tree was properly balanced. The advantage of splay tree coding is its ability to move the more probable characters closer to the top of the tree while quickly and easily balancing the tree. Splaying accomplishes this by twisting the tree branch around the current character. The result of this operation is that the distance from the current character to the root is shortened by a factor of 2. A side benefit is that it tends to group the characters with similar probabilities.

3. RATIONALE FOR PIPELINING ALGORITHMS

Individually, each of the various data compression algorithms have certain strengths and weaknesses. To compare these algorithms, the characteristics of redundancy must be examined.

Some data compression techniques take advantage of character redundancy to compress data. Prefix code methods, such as Huffman coding and splaying utilize this technique. The prefix code methods tend to perform well on data where relatively few characters occur frequently. Consider the following data example with 52 characters:

$$a; b; c; d; e; f; g; h; i; j; k; l; m; n;$$
$$o; p; q; r; s; t; u; v; w; x; y; z; \quad (1)$$

In this example, every other character is a semi-colon and therefore has a probability of 0.5. The remaining characters are equally distributed and each has a probability of 0.0192. If fixed length codes were used to represent the 27 unique characters, each code would be a minimum of 5 bits in length. The resulting data length would be at least 52 * 5 or 260 bits. If Huffman encoding is used, the semi-colon, being the most probable character, could be represented by 1 bit. The remaining characters, all of equal probability, would each require 5 or 6 bits (details of computing these are omitted for brevity). The total data length would then be (26 * 1) + (16 * 6) + (10 * 5) or 172 bits. This results in compressing the data to 66% of its original size.

String redundancy occurs when the same string of characters appears two or more times in the data. Data compression techniques that use string parsing, such as LZW or dictionary trees, exploit this data characteristic. In both of these algorithms, strings of characters are parsed from the data and are used to build a dictionary of strings. Each string is assigned a code based upon the address of the string within the dictionary. As the length of the repeated strings increase, so does the rate of compression. The rate of compression also increases as the frequency of string occurrence increases. Consider this example with 55 characters:

$$aababcabcdabcdeabcdefabcdefgabcdef$$
$$ghabcdefghiabcdefghij \quad (2)$$

If fixed length codes were used to represent the 10 unique characters (*a*–*j*), each code would be a minimum of 4 bits in length. The resulting data length would be 55 * 4 or 220 bits. To compress this data using the LZW algorithm, a table size of 32 could be used, requiring a fixed length code word size of 5 bits. Following the LZW algorithm, the first ten table entries are initialized with the 10 unique characters. After initialization, the string parsing begins. When complete, the algorithm will reduce the example to 19 code words of 5 bits each for a total of 95 bits (details of computation are omitted for brevity). This effectively compresses the data to 43% of its original size.

If LZW encoding is applied to the first example, no compression occurs. With the 27 characters in this example, 5 bits are required for encoding. This allows a table size of 32. The reason that no compression occurs is that this data contains redundant characters, but does not contain any redundant strings. Since the LZW algorithm finds no redundant strings, it must send a separate code word for each character. The result is 52 * 5 or 260 bits. If the table size is increased, the data would be expanded rather than compressed.

If the data string in the second example is compressed using Huffman codes, some compression occurs, but, it is less than in example 1. Example 2 contains some redundant characters, but there is less character redundancy than in example 1. Since the Huffman algorithm compresses redundant characters, less compression is possible. In this instance, the character probabilities are more equally distributed. The number of output bits will be 173, resulting in a compressed size 77% of the original.

These examples show that the various algorithms behave differently with a variety of data. With character redundancy, the algorithms which use variable length code words (e.g. Huffman,[5] Splay[7]) outperform the string parsing algorithms. If the data exhibits string redundancy, the string parsing algorithms perform better. It should be noted that if the data contains string redundancy, it will always have some character redundancy. The converse, however, is not true.

Given these observations, it appears to be desirable to combine string parsing with variable length code words. The advantage of combining the two techniques would be to represent strings as variable length code words. If this were possible, more efficient compression should be obtainable. One important observation was made while examining the output of the string parsing algorithms: for every occurrence of a given redundant string, the algorithms output the same code word to represent it. This is another form of redundancy. If this redundancy could be reduced or eliminated, further compression would be possible. The proposed pipelining scheme includes this feature. Fig. 1 illustrates the concept of a two-stage pipelining of compression algorithms.

```
         -----------           ----------
        I         I  FIXED   I        I VARIABLE
DATA    I STRING  I  LENGTH  I PREFIX I  LENGTH
------->I PARSING I--------->I  CODE  I------------->
        I  ALGO.  I  CODE    I  ALGO. I   CODE
        I         I  WORDS   I        I   WORDS
         -----------           ----------
```

Figure 1. Illustration of a 2-Stage Pipelining Algorithm.

As shown here, the redundant strings were first compressed utilizing one of the string parsing algorithms. The output of this was then pipelined directly into a variable length code word algorithm. The sequence of the algorithms is critical. If the order of the algorithms were reversed, the string parsing algorithm would convert the variable length code words into fixed length code words. This would cause the advantage of the variable length code words to be lost.

4. RESULTS

To determine the efficacy of the proposed pipelining scheme, we considered two pipelining algorithms. The first algorithm combines a dictionary forest as described by Jakobsson[6] with the splay algorithm of Jones.[7] The second algorithm combines LZW[14] with splaying.[7] The splay algorithm was chosen rather than Huffman coding, because it dynamically adapts to changes in probability. The performance of these two algorithms is compared with some of the other existing data compression algorithms.

Two sets of data files were used. The first set of data files consisted of a mixture of actual files along with several files that were contrived to provide significant redundancy. These are described in Table 1. The second set of data files (in Table 3) was chosen to further explore the tendencies discovered during the first set of tests. Each algorithm was tested individually before testing the combined pipelined versions. For comparison, the Unix Compress program and the PKPAK program were also run.

Table 1. Data Files—Set-1

File #	Contents
1	A-Z (64X)
2	A-Z,AA-ZZ,...,AAAAA-ZZZZZ
3	A-H (8X)
4	A-H,AA-HH,...,AAAAA-HHHHH
5	ABRACADABRA (700X)
6	PASCAL SOURCE 1
7	PASCAL SOURCE 2
8	PASCAL SOURCE 3
9	PASCAL SOURCE 4
10	OBJECT 1
11	OBJECT 2
12	OBJECT 3

Table 2 summarizes the results of test 1. It lists the original file sizes along with the compressed file sizes† with each of the data compression algorithms considered. The last two columns of the table (i.e. Forest-Splay and LZW-Splay) are the two pipelining algorithms considered here. The performance of these pipelining algorithms is compared with five other algorithms: Forest of trees,[6] Splay,[7] LZW,[14] Unix Compress,[12] and PLPAK.[11] We make the following observations from these results:

- Except for the dictionary forest algorithm, all the other six algorithms were able to achieve some measure of compression on all of the test files. For files 10 and 11, the forest algorithm failed to compress the data.
- In every test, the pipelined forest-splay algorithms produced better compression rates than either algorithm alone.
- Both the pipelined forest-splay and the pipelined LZW-splay excelled in the compression of small files with significant redundancy.
- All of the LZW algorithms (individual and pipelined) produced similar results. They had excellent compression on the program sources.

† The file sizes are in bytes.

Table 2. Comparison of Data Compression Algorithms with Data Set-1

File #	Orig. Size	Forest	Splay	LZW	Unix Comp.	PKPAK	Forest-Splay	LZW-Splay
1	1793	571	1280	347	351	349	571	290
2	953	366	327	383	388	409	286	346
3	34	30	38	31	36	34	19	26
4	131	66	55	72	77	75	47	61
5	9053	2728	3714	655	660	658	1114	584
6	10341	5340	7010	4136	4141	4088	5291	4327
7	5444	3320	3618	2535	2540	2517	2840	2581
8	15077	8905	9494	6152	6120	6117	7924	6358
9	6688	4454	4338	2882	2887	2858	3218	2943
10	15284	19326	14592	15782	14157	13770	14514	13476
11	343	372	295	274	299	296	285	272
12	2594	2241	2020	1812	1891	1856	1864	1726

- Except for the very small, highly redundant files, the pipelined LZW-splay algorithms performed even better than the pipelined forest-splay algorithms.
- LZW-splay demonstrated the highest compression rate for object files.
- LZW-splay showed the best overall performance. On average, it excelled by 6% on both the highly redundant files and the binary files. It was only 1% less efficient on the source files.

Table 3. Data Files—Set-2

File #	Contents
13	GRAPHICS 1
14	GRAPHICS 2
15	GRAPHICS 3
16	GRAPHICS 4
17	GRAPHICS 5
18	DATABASE 1
19	DATABASE 2
20	C SOURCE 1
21	C SOURCE 2
22	C SOURCE 3
23	C SOURCE 4
24	C SOURCE 5
25	C SOURCE 6
26	TEXT 1
27	TEXT 2
28	TEXT 3
29	TEXT 4
30	TEXT 5
31	OBJECT 1
32	OBJECT 2
33	OBJECT 3
34	OBJECT 4
35	OBJECT 5

The second set of tests was chosen to further explore the characteristics discovered in the first round of tests. The results are shown in Table 4. These tests were applied only to the LZW and the pipelined LZW-splay because they excelled in the compression of the actual files. Although the pipelined forest-splay algorithms produced excellent results on the files of fewer than 500 bytes, this particular characteristic has a minimum benefit for most data compression applications. Small files generally do not get compressed because there is little to be gained when they are compressed. This characteristic could, however, be useful for data transmission where the packet sizes are small. This group of tests concentrated on a variety of actual files: graphics files, database files, C sources, text files and binary object files.

After reviewing the results of Table 4, the following observations were made:

- Graphics files 1 and 2 were line art drawings. These contained an abundance of white space. The large white space provided significant redundance for compression. LZW-splay produced the best results on this type of file.
- Graphics files 3, 4, and 5 were black and white scanned photographs. These images contained many areas of varied contrast. The shading in the image was accomplished by various dithered patterns. Because of the many shade changes in the images, there was less redundance than in 1 and 2. LZW showed a slight advantage on 3 and 4, but LZW-splay did better on 5. This image had a more pronounced pattern which provided for more redundance than 3 or 4.
- In the C source files, LZW-splay produced greater compression when the file size was less than 1000 bytes.
- Text files 1–4 were an assortment of documentation files. On the larger files, LZW was better by a few percentage points, while LZW-splay was again slightly better at compressing the small files.
- Text file 5 was a collection of articles from USENET and was the largest test file. In this instance, the LZW programs performed better than LZW-splay.

Table 4. Comparison of Data Compression Algorithms with Data Set-2

File #	Orig. Size	LZW	Unix Comp.	PKPAK	LZW-Splay
13	5437	1303	1391	1297	1285
14	17268	3003	3037	3026	2616
15	18768	7128	7452	7590	7663
16	14471	5136	5336	5360	5573
17	19974	11906	11369	11491	11408
18	25856	5235	6107	5931	5615
19	35072	6857	8293	8393	7359
20	3868	1974	1979	1980	2003
21	11073	5568	5584	5555	5748
22	123	110	115	112	99
23	1012	602	606	604	561
24	3697	2021	2026	2028	2046
25	19374	8348	8062	8059	8575
26	4607	2179	2184	2146	2233
27	8959	4502	4507	4494	4709
28	720	474	479	476	451
29	140355	50994	43508	44291	48178
30	967086	860627	571776	607370	656669
31	25462	23514	23999	23986	22154
32	34678	29808	30124	30120	27775
33	12303	10986	11303	11258	8955
34	51749	43677	45289	44390	35741
35	2994	2031	2165	2098	2036

It is also interesting that the Unix Compress program achieved significantly better compression on this file. This can be attributed to the fact that Compress can utilize 16 bits, providing a string table of 64 K bytes, while the other LZW based programs (including LZW-splay) use a maximum of 12 bits for a 4 K table size. The larger dictionary seems to provide a big advantage when compressing very large files.

- LZW-splay consistently provided higher compression on the binary files.

5. CONCLUSIONS

The idea of pipelining two different data compression algorithms is a viable one. Depending upon the data characteristics and the selection of appropriate data compression algorithms, pipelining can improve data compression ratios. The pipelining concept retains the general characteristics of the individual algorithms and usually enhances the strengths of each while minimizing their weaknesses. Like other universal dynamic compression techniques, this is a one pass process which adapts to provide excellent compression ratios for a variety of inputs.

Pipelined algorithms seem to perform best at the opposite extremes of redundancy. They exhibit superior compression ratios when the redundancy is either very slight or very pronounced. Also, they tend to excel in the compression of small files.

Both the forest-splay pipeline and the LZW-splay pipeline could be useful for compressing small data files or for compressing small packets of data for transmission. The LZW-splay pipeline appears to be particularly useful for obtaining maximum compression on files with significant redundance, binary object files and graphics image files. The major disadvantage of pipelining algorithms would appear to be the overhead of running two completely different algorithms sequentially. This study did not address the execution time of pipelined compression. This is an area that should be the topic of future work.

Another area that would seem to warrant further consideration is to add some form of intelligence to the process to allow the use of multiple algorithms. The decision making process could dynamically evaluate the data being compressed to determine which algorithm(s) would provide the best compression ratio. It is conceivable that when the data is not consistent throughout the entire file, that the best compression ratios would be obtained by using different data compression algorithm(s) for portions of the data.

REFERENCES

1. ARC, ARC File Archive Utility, Version 5.1, System Enhancement Associates, Wayne, NJ (1986).
2. C. J. Date, An Introduction to Database Systems, Volume 1, Fourth Edition, Addison-Wesley (1986).

3. N. Faller, An Adaptive System for Data Compression, Record of the 7th Asilomar Conference on Circuits, Systems and Computers (Pacific Grove, CA, Nov. 1973), Naval Postgraduate School, Monterey, CA, pp. 593–597.
4. R. G. Gallager, Variations on a Theme by Huffman, *IEEE Transactions on Information Theory* **IT-24**, (6), 668–674 (1978).
5. D. A. Huffman, A Method for the Construction of Minimum Redundancy Codes, *Proceedings of the IRE* **40**, (9), 1098–1101 (1952).
6. M. Jakobsson, Compression of Character Strings by an Adaptive Dictionary *Bit* **25**, (4), 593–603 (1985).
7. D. W. Jones, Application of Splay Trees to Data Compression, *Communications of the ACM* **31**, (8), 996–1007 (1988).
8. D. E. Knuth, Dynamic Huffman Coding, *Journal of Algorithms* **6**, (2), 163–180 (1985).
9. D. A. Lelewer and D. S. Hirschberg, Data Compression, *ACM Computing Surveys* **19**, (3), 261–296 (1987).
10. E. Mäkinen, On Implementing two Adaptive Data-compression Schemes, *The Computer Journal* **32**, (3), 238–240 (1989).
11. PKPAK Fast! File Archival Utility, Version 3.6, *PKware, Inc., Glendale, WI* (1988).
12. UNIX User's Manual, Version 4.2. Berkeley Software Distribution, Univ. of California, Berkeley, California (1984).
13. J. S. Vitter, Design and Analysis of Dynamic Huffman Codes, *Journal of the ACM* **34**, (6), 825–845 (1987).
14. T. A. Welch, A Technique for High-Performance Data Compression, *IEEE Computer* **17**, (6), 8–19 (1984).
15. J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transactions of Information Theory* **IT-23**, (3), 337–343 (1977).
16. J. Ziv and A. Lempel, Compression of Individual Sequences via Variable-Rate Coding, *IEEE Transactions on Information Theory* **IT-24**, (5), 530–536 (1978).

Announcements

3–6 SEPTEMBER 1990

UNIVERSITY OF DURHAM, U.K.

The Second International Conference on Visual Search

Background

The term 'Visual search' has been used to cover a range of activities from human cognitive phenomena to applied problems, for man and machine, in industrial, medical and military environments. Since the opportunities for academic discussion of visual search are generally restricted to a narrow range of disciplines, workers in a particular area can be unaware of recent developments in related fields.

This is the second in a series of international conferences devoted exclusively to all aspects of visual search processing. The proceedings of the first international conference on visual search was published by Taylor and Francis in October, 1989.

Venue

The conference will take place at the University of Durham. Accommodation will be in Grey College, within walking distance of the lecture halls.

Invited Lecture

Prof. J. Beck
University of Oregan, U.S.A.

Keynote Speaker

Prof. L. Stark
University of California, Berkeley, U.S.A.

Sessions

Conference sessions will include (but not be limited to) the following:

- Attention and Segmentation
- Eye Movements
- Computer Vision
- Search Modelling
- Applied Aspects of Search

There will also be a workshop. The conference proceedings will be published.

Mailing Address

All correspondence and enquiries about the conference should be addressed to:

Bell Howe Conference (VS)
1 Willoughby Street, Beeston, Nottingham NG9 2LT. Tel: (0602) 436323; Fax: (0602) 436440

5–7 SEPTEMBER 1990

UNIVERSITY OF OXFORD

VLSI for Artificial Intelligence and Neural Networks

International Workshop

An International Workshop on VLSI for Artificial Intelligence and Neural Networks is to be held at the University of Oxford on the 5–7 September 1990.

Topics will include Prolog Machines, Lisp Architectures, Functional Programming Oriented Architectures, Knowledge Based Systems, Neural Networks, Architectures for Neural Computing, Logic Programming Systems, Garbage Collection Support, Content-Addressable Memories, Hardware Accelerators, Symbolic Machines, Parallel Architectures.

The Programme Committee is drawn from the UK, USA, Canada and France and has been organised by Dr. Will Moore of the Department of Engineering Science at the University of Oxford and Dr Jose Delgado-Frias of the Department of Electrical Engineering, State University of New York at Binghampton.

The aim of the workshop is to provide a forum where AI experts, VLSI and Computer Architecture designers can discuss the present status and future trends on VLSI and ULSI implementations of machines for AI computing.

The workshop will be held in Jesus College with meals and accommodation available on the nights of the 4–6 September.

Further details are available. If you would like further information please write, telephone or fax:
CPD Unit, University of Oxford, Department for External Studies, Rewley House, Wellington Square, Oxford OX1 2JA England.
Telephone: (0865) 270373–direct line
(0865) 270360—switchboard
Fax: (0865) 270708 (CPD Unit—Ext Stud)

23–25 OCTOBER 1990

BRIGHTON, UK

1990 ADA UK International Conference

The annual Ada Conference has become the major event in the UK calendar, at which those interested in the Ada language and its environments meet to discuss and exchange ideas on recent advances in Ada and its use. The 1990 Programme is structured to allow delegates to attend either the full conference or those parts which are of particular interest and relevance to their needs. Both full conference and day registrations will be available.
The Conference will comprise Tutorial sessions and submitted Papers. A non-exhaustive list of topics will be:

- Real-Time
- Object-Oriented Design
- Long-Life Ada
- Ada 9X
- Secure Subsets
- Management Issues.

The Conference will be held at the Royal Albion Hotel, which is located on the seafront between the Marina and the Brighton Centre. Accommodation will be available at the hotel or, if preferred, elsewhere in Brighton.

Conference organised by Ada Language UK Ltd, telephone +44 904 412740.

Mailing Address: c/o Computer Science Department, University of York, Heslington, York YO1 5DD