Evolving Nonlinear Predictive Models for Lossless Image Compression with Genetic Programming

Alex Fukunaga and Andre Stechert

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr., M/S 126-347
Pasadena, CA 91109-8099
alex.fukunaga@jpl.nasa.gov, andre.stechert@jpl.nasa.gov

ABSTRACT

We describe a genetic programming system which learns nonlinear predictive models for lossless image compression. S-expressions which represent nonlinear predictive models are learned, and the error image is compressed using a Huffman encoder. We show that the proposed system is capable of achieving compression ratios superior to that of the best known lossless compression algorithms, although it is significantly slower than standard algorithms.

1 Introduction

Lossless image compression is a problem with many real-world applications which has been studied by many researchers. The current dominant paradigm for lossless compression is predictive coding. State of the art lossless image compression algorithms (based on predictive coding) include the CALIC algorithm of Wu and Memon [WM97] and the LOCO-I algorithm of Weinberger et al. [WSS96]. Reviews of lossless image compression can be found in [MW96, MW97].

This paper proposes the use of genetic programming (GP) [Koz92] in lossless image compression as the mechanism for representing and learning nonlinear models for predictive coding. Because of the enormous computational cost of evolving nonlinear predictive models would be prohibitively expensive using standard GP systems, we have implemented a highly efficient, *genome-compiler* GP system which compiles s-expressions into native machine code to enable the application of GP to this problem.

We evaluate our GP-based compression system by comparison with the state of the art lossless image compression algorithms and show that it is possible to obtain compression ratios superior to the best known algorithms.

The rest of the paper is organized as follows. In Section 2, we review predictive coding based image compression. Section 3 describes the GP-based compression system. Section 4 presents an empirical evaluation of our system using some test images, and compares the results with that of the best known lossless compression algorithms. We discuss related work in Section 5, and we conclude in Section 6 with a discussion and directions for future work.

2 Predictive Coding Based Image Compression

Predictive coding (Figure 2) is an image compression technique which uses a compact model of an image to predict pixel values of an image based on the values of neighboring pixels. A *model* of an image is a function $model(x, y)$, which computes (predicts) the pixel value at coordinate (x, y) of an image, given the values of some *neighbors* of pixel (x, y), where neighbors are pixels whose values are known. Typically, when processing an image in raster scan order

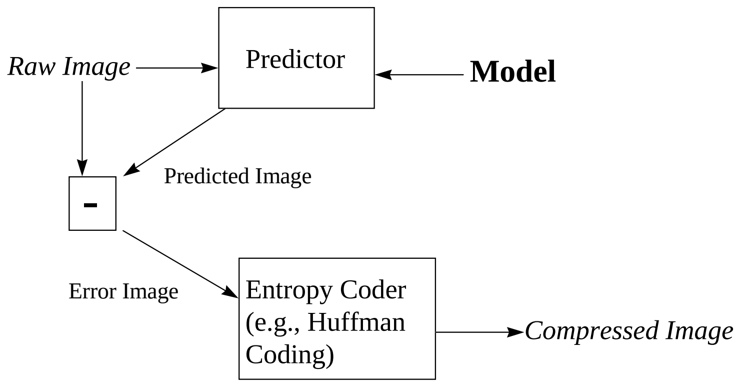


Figure 1 Predictive Coding Based Compression

(left to right, top to bottom), neighbors are selected from the pixels above and to the left of the current pixel. For example, a common set of neighbors used for predictive coding is the set {*(x-1,y-1), (x,y-1), (x+1,y-1),(x-1,y)*}. *Linear predictive coding* is a simple, special case of predictive coding in which the model simply takes an average of the neighboring values. *Nonlinear* models assign arbitrarily complex functions to the models.

Suppose that we have a perfect model of an image, i.e., one which can perfectly reconstruct an image given the pixel value of the border pixels (assuming we process the pixels in raster order). Then, the value of the border pixels and this compact model is all that needs to be transmitted in order to transmit the whole information content of the image. In general, it is not possible to generate a compact, perfect model of an image, and the model generates an *error signal* (the differences at each pixel between the value predicted by the model and the actual value of the pixel in the original image.

There are two expected sources of compression in predictive coding based image compression (assuming that the predictive model is accurate enough). First, the error signal for each pixel should have a smaller magnitude than the corresponding pixel in the original image (therefore requiring fewer bits to transmit the error signal). Second, the error signal should have less entropy than the original message, since the model should remove many of the "principal components" of the image signal.[1] To complete the compression, the error signal is compressed using an entropy coding algorithm such as Huffman coding or arithmetic coding [NG96]. State of the art algorithms such as CALIC also perform context modeling prior to applying entropy coding – see [MW96, MW97]. Our system does not apply context modeling techniques.

If we transmit this compressed error signal as well as the model and all other peripheral information, then a receiver can reconstruct the original image by applying an analogous decoding procedure (see Figure 2).

3 Evolving Nonlinear Predictive Models with a GP

We developed a GP system which, given an image, compresses the image by evolving a nonlinear predictive model for the image. The nonlinear model is represented as a s-expression.

[1]If the model were perfect, then the error signals would consist of all 0's, and could be compressed to a few bytes.

```
**Encoder**(Model,Image)
  for $x = 0$ to $xmax$
    for $y = 0$ to $ymax$
      Error[x,y] = Image[x,y] - Model(x,y)

**Decoder**(Model)
  for $x = 0$ to $xmax$
    for $y = 0$ to $ymax$
      Image[x,y] = Model(x,y) + Error[x,y]
```

Figure 2 Algorithm schema for encoding/decoding in predictive coding. $Model(x, y)$ is a function that takes the coordinates of a pixel and returns a predicted value of that pixel. *Image* and *Error* are two-dimensional arrays.

The terminals used for genetic programming were:

- values of the four neighboring pixels *Image[x-1,y-1], Image[x,y-1], Image[x+1,y-1], Image[x-1,y].*
- selected constant values: 1, 5, 10, 100.

The functions used were:

- arithmetic functions: +,-,*,% (protected division [Koz92])
- *MIN(a,b)* and *MAX(a,b)* functions which return the minimum and maximum values of their two arguments, respectively.

As we noted in Section 2, a standard entropy coding algorithm needs to be applied to the error image. For this experiment, we used the Huffman coder described in Section 3.2. Note that the codebook (which is the mapping from pixel values to the prefix codes used by the Huffman coder) needs to be output in the compressed image file.

In addition, given the four pixel neighborhood we use, the pixel values of the borders of the image, i.e., the top row, the leftmost column, and the rightmost column need to be stored explicitly (these are the border cases for which we can not apply the predictive model). Also, the model (which is unique for each image) must also be stored in the compressed image data. We applied Unix *compress* (which uses Lempel-Ziv coding) to the border pixels and the model, and concatenated these to the Huffman-coded error signal. Finally, two integer values indicating the size of the image (height,width) were added to the file. Given this data, we can reconstruct an image without loss of information.

Thus, the exact file size of the compressed image (the values reported in the experiments below) is:
sizeof(HuffmanCodedError) +
sizeof(HuffmanCodeBook)+
sizeof(CompressedBorder) +
sizeof(CompressedModel) + sizeof(2 integers)

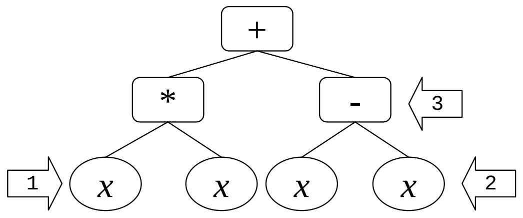
The size of the codebook depends on the number of unique values in the error image. Because the s-expression models are unconstrained in the range of values that they can return, the worst-case size of the codebook is actually the number of pixels in the image, even though there are only 256 unique pixel values in the original image. To enable fast entropy coding (Section 3.2), we have actually limited the codebook size to $2^{16} = 65536$ entries (the 32-bit integers returned by the s-expression are stored modulo 2^{16}). This means that the size of the codebook can possibly become a very significant component of the compressed data that must be transmitted. We therefore implemented a execution speedup heuristic which abandons evaluation of an individual once the number of unique error image pixel values generated exceeds either some constant value (we used 40000 in the experiments below) or 25% of the total number of pixels in the input image. By doing this, we avoid the cost of running the entropy coder for individuals that both a) look very unpromising and b) will be extremely computationally expensive to evaluate.

In the experiments described in Section 4, our GP system was configured as follows: population=500, generations=30, tournament selection (size=5), 90% crossover, 10% reproduction, no mutation).

3.1 Genome Compiler

In order to evaluate a single individual, the s-expression needs to be evaluated for each pixel in the image (excluding the borders). This requires hundreds of thousands to millions of repeated execution of the same s-expression per evaluation. We originally implemented the image compression application using lil-gp 1.1 [PZ96], a well-known, efficient C implementation of GP. For a single GP run with a 2000 member population executed for 50 generations on a 296MHz Ultrasparc 2, compression of of a 256 by 256 image took two weeks. Not only was this much too slow for practical use, it made experimentation infeasible. Thus, we sought to make individual evaluations as efficient as possible.

We therefore extended lil-gp by implementing a *genome compiler* which translates s-expressions into efficient SPARC machine code prior to execution. The major benefit of compilation is the removal of function call overhead during the s-expression tree evaluation, which we found was responsible for the vast majority of the computation time in the standard lil-gp based system. Figure 3 shows an example of this compilation process.



```
[1] mov  fp31, fp0
    mov  fp31, fp1
    mult fp0, fp1, fp0
    mov  fp31, fp1
[2] mov  fp31, fp2
[3] sub  fp1, fp2, fp1
    add  fp0, fp1, fp0
```

Figure 3 Example of s-expression compilation: An s-expression with corresponding assembly-level code. Numbered breaks in the code correspond to the code generated so far when the post order traversal has progressed to the node indicated in the tree diagram.

The genome compiler's performance,[2] when applied to problems where individuals are repeated many times, compares favorably with the fastest reported GP systems, including the CGPS system of Nordin and Banzhaf [NB95], which directly manipulates SPARC machine code, yielding roughly two orders of magnitude speedup over lil-gp on symbolic regression problems.

Further details about the genome compiler can be found in another paper in this conference proceedings [FSM98].

3.2 Entropy Coding Estimation

The evaluation of a single candidate model requires both the generation of an error image and an estimation of the size of the compressed image, based on the error image.

After the implementation of the genome compiler described above, we found that the vast majority (98% for 256x256 images) of the execution time was being spent in the Huffman coding module.[3] Note that during evolution, we only need to estimate the size of the Huffman coded error image, as well as the size of the codebook – there is no need to actually generate the Huffman coded error image. Generating the coded error image requires that we iterate through the image, applying the codebook to each value, and then outputting the value of the codebook which corresponds to each value. Because this involves numerous bit-level operations per image value, this adds a significant overhead.

Thus, we implemented a *Huffman code size estimator*, which, during evolution, only generates the codebook while counting the frequencies of the values in the error image. This allows us to compute the size of the compressed error image and the codebook without the overhead of actually compressing the error image. This approach resulted in a very significant reduction of execution time. Currently, the Huffman coding related operations consumes only 50% of the execution time during evolution.of a 256x256 image, and has resulted in an overall speedup of 25 over the previous entropy coding approach.

For a 256 by 256 image, our current implementation completes a GP run involving 25000 evalua-

[2]By which we mean the comparative speed in time to execute a GP run over standard C implementations of GP such as lil-gp[PZ96] and SGPC [TC93] running on the same problem on the same machine as the genome compiler.

[3]Originally, we were using a Huffman encoder found in the standard Numerical Recipes library [PTVF92].

tions in approximately 2 hours on a 296MHz Ultrasparc 2 (this previously took 2-3 days before implementation of Huffman code size estimation).

Previous work which optimized predictive coding models have used the entropy[4] of the error image, instead of directly trying to estimate the compressed filesize as we did. Although entropy provides a fast, reasonable approximation of an encoded filesize, the size of the codebook is not accounted for by the entropy computation. Because we use 16-bit error image values, the size of the codebook is potentially a significant component of the compressed filesize. Furthermore, we found that our Huffman code size estimator was as fast as the entropy computation for the images we tested (up to 1024 by 1024).[5]

4 Results

The genetic programming system for evolving models for predictive coding lossless image compression was evaluated by comparing the size of the compressed files with a number of standard lossless compression algorithms on a set of grey scale images. The images used were science images of planetary surfaces taken from the NASA Galileo Mission image archives (we used these images because these are of greatest interest to our sponsor). Note that in this report, we focus on grey scale images, but there is a straightforward extension a color implementation by operating on the three image planes independently – this is what many state of the art algorithms such as CALIC do.

The compression results of the following algorithms are shown in Table 1.

- *evolved*: The evolved predictive coding compression algorithm
- *CALIC*: A state of the art lossless image compression algorithm, described in [WM97]. In general, this algorithm provides the best compression ratio among previous algorithms.
- *LOCO-I*: This is an algorithm developed by Weinberger et al. [WSS96], which was recently selected as the new ISO JPEG-LS (lossless JPEG) baseline standard.
- *gzip, compress, pack*: These are standard Unix string compression utilities; *gzip* implements the Lempel-Ziv (LZ77) algorithm, *compress* implements the adaptive Lempel-Ziv-Welch (LZW) algorithm, and *pack* uses Huffman coding.
- *szip*: A software simulation of the Universal Source Encoding for Science Data (USES) algorithm [VYL92, YRM93, RYM93, Cen93], a standard lossless compression hardware used by NASA (This is sometimes referred to as the Rice Chip).[6]

It important to note that a different model is evolved for each image that the genetic programming system is applied to. In contrast, the other approaches (CALIC, GIF, etc.) apply a single model to every image. Thus, the time to compress an image using the genetic programming approach is several orders of magnitude greater than the time it takes to compress an image using other methods. However, the time to decompress an image is competitive with other methods. Therefore, the genetic programming system is an example of an *asymmetric compression algorithm* (slow compression, fast decompression).

As Table 1 shows, the compressed file sizes obtained using the GP-evolved models is superior to all of the other algorithms for these test images.

5 Related Work

Salami developed part of an evolvable hardware system for lossless image compression [SIH97]. They used a genetic algorithm to evolve weights for a linear predictor for predictive coding, and showed that the entropy of the error image for some test images was lower than that of CALIC and LOCO (the error image entropy was measured instead of the compression ratio, since they did not implement an entropy coder).

Our work differs from that of [SIH97] in two respects. First, their approach is more sophisticated: while we evolve a single model for each image, they

[4]The information-theoretic entropy of the error image can be computed as $E = \sum_{i=1}^{N} log(freq(i))$, where N is the number of unique symbols in the error image, and $freq(i)$ is the frequency of the ith symbol.

[5]Even though the asymptotic complexity of the Huffman code size estimator for an input of length N is $O(NlogN)$, the entropy coder, whose complexity is $O(N)$, has large constant factor cost because of the computations of logarithms. Thus, we found that the entropy coder is not significantly faster than the Huffman code estimator for images sizes that we tested.

[6]For each image, the file sizes generated by the best command line configuration of *szip* is reported.

Image name	original size	**evolved**	CALIC	LOCO-I (JPEG-LS)	compress	gzip	pack	szip
earth	72643	30380	31798	32932	42502	40908	55068	40585
earth4-128	11246	5513	5631	5857	7441	6865	8072	7727
earth6	20400	9288	10144	10488	11339	10925	13264	12793
earth7	21039	10218	11183	11476	13117	12520	15551	13269
earth8	19055	9594	10460	10716	11699	11350	13298	12465

Table 1 Compression results (in bytes) of various compression techniques applied to set of test images.

adapt the model (via a genetic algorithm) as they traverse the image. By doing this, they achieve both high compression ratio and near real-time execution speed. Second, we evolved nonlinear predictive models represented as s-expressions, as opposed to the weighted linear models used in [SIH97].

A promising direction for future work would be to combine Salami et al's adaptive model approach with our GP-based nonlinear model evolution system. While this is likely to be slower than the evolvable hardware implementation of [SIH97], such an architecture implemented using our genome compiler (Section 3.1) may be able to provide acceptable, near real-time performance.

Neural networks have previously been used to learn nonlinear predictors (c.f. [JKHM93]). However, it is difficult to compare these approaches with our work because they do not compare their system to current, standard lossless image compression algorithms (this is in part because many of the advances in lossless compression are quite recent [MW96, MW97]).

Salami and colleagues have also developed an evolvable hardware system for *lossy* compression which evolves nonlinear predictive models on function level evolvable hardware [SMH97, SMH96]. This is the work which is most similar to our own, since their system could be used for lossless compression by not using quantization. Our approach differs in the representation that is used (GP s-expressions vs. evolvable hardware configurations) and in the focus on efficient software implementation techniques (e.g., genome compiler) as opposed to hardware implementation.

Image compression using GP has been previously studied in the context of *programmatic compression* [Koz92, NB96], which is similar to predictive coding, in that an image is used as the target function for symbolic regression. A major difference between programmatic compression and predictive coding is that instead of learning a model which predicts the value of the current pixel based on neighboring pixel values, programmatic compression uses only the current pixel position in its terminal set.[7] That is, programmatic compression does not exploit knowledge of neighboring pixel values). Another major difference is that in programmatic compression, no error image is computed – only the evolved model is stored (and possibly further compressed using a standard dictionary-based or entropy-coding compressor). Thus, programmatic compression is fundamentlaly a lossy technique, although in theory it is possible to achieve lossless compression if a perfect model is discovered by the GP. Koza [Koz92] initially demonstrated this technique on a 30 by 30 bitmap. Nordin and Banzhaf [NB96] used CGPS, a very efficient GP system, and heuristics such as *chunking* (tiling an image into smaller subimages which were separately compressed) to scale up the technique to 256 by 256 images.

Previous work in the implementation of high-performance GP systems closely related to our genome compiler includes that of Nordin and Banzhaf, [NB95], whose CGPS system directly manipulates SPARC machine code, and Juille and Pollack [JP96], whose system compiles s-expressions into virtual stack-based machine code. A detailed comparison between the genome compiler and previous work can be found in [FSM98].

6 Discussion and Future Work

The research reported here is preliminary, and is only the first step in understanding the capabilities and limitations of using genetic programming for lossless image compression.

We believe that our initial results are quite promising, since they show that the evolved models can yield an improvement over CALIC, which is currently the best known lossless image compres-

[7]Koza used the X,Y position in his terminal set; Nordin and Banzhaf found that using a linear index performed better than using both X and Y values.

sion algorithm. Furthermore, these results were obtained without any special tuning of algorithm control parameters or the function/terminal sets for the GP system. Due to the extremely long runtimes for evolving compressed images (at the time that the experiments were being run, the performance improvements described in Sections 3.1 and 3.2 were still under development), we have only collected data for a small number of runs. We are currently collecting more data to better understand the relative efficacy of evolutionary compression on a wide range of image classes. Further experimentation is also necessary to understand the stability of this approach (that is, how sensitive the GP is to random seeds).

However, it should be noted that the genetic programming system takes several orders of magnitude more time to evolve a model that achieves its superior results (several hours per image) than the other approaches (which run in a few seconds). Although slow compression times are acceptable for some applications, as long as decompression is fast (e.g., for archiving images), this is not acceptable for applications requiring near real-time compression times.

We have performed some preliminary experiments using a standard GA to evolve a weighted linear model for our test images (as opposed to the nonlinear models evolved by our GP). So far, we have not obtained any linear predictor which does not result in a larger compressed file size than the evolved GP nonlinear models. This indicates that the ability of the GP to represent nonlinear models plays a significant role in achieving high compression ratios. We are currently performing a more systematic study to verify the utility of the nonlinear models.

There are many more directions which can be pursued in evolutionary image compression. First, it seems worthwhile to experiment with different entropy coders. Second, there are more sophisticated predictive coding architectures which can be explored. One such example would be a two-pass model in which a standard (evolved or hand-coded) linear model is first applied to minimize first order entropy, then followed by the application of an evolved nonlinear model to model the error after the linear model is applied. Third, we can divide the image into subimages and apply the compression separately to the subimages.[8]

Finally, a very intriguing prospect would be to try to evolve models which are more general than the image-specific nonlinear models that were explored in this paper. By using sets of images as fitness cases instead of a single image, it is possible to try to evolve a single predictive model (such as those handcoded in traditional lossless compression algorithms) which works well for a large class of images. This will no doubt require an enormous amount of computation. However, it may be possible to generate a nonlinear model, which when possibly combined with context modeling techniques, yields a better, general purpose model than those used by state of the art algorithms.

Acknowledgments

The research described in this paper was performed at the Center for Integrated Space Microsystems, Jet Propulsion Laboratory, California Institute of Technology, and was supported by the National Aeronautics and Space Administration. Thanks to Darren Mutz for programming assistance. Adrian Stoica, Benny Toomarian, and Ken Hayworth provided helpful comments on the work. Thanks to Bill Punch and Douglas Zonker for making lil-gp publicly available.

References

[Cen93] Microelectronics Research Center. Universal source encoder for space - USES, preliminary product specifcation, version 2.2. Technical report, Microelectronics Research Center, University of New Mexico, 1993.

[FSM98] A. Fukunaga, A. Stechert, and D. Mutz. A genome compiler for high-performance genetic programming. In *Proceedings of the Genetic Programming Conference*, 1998.

[JKHM93] W.W. Jiang, S-Z. Kiang, N.Z. Hakim, and H.E. Meadows. Lossless compression for medical imaging systems using linear/nonlinear prediction and arithmetic coding. In *Proc. IEEE International Symposium on Circuits and Systems*, volume 1, pages 283–6, 1993.

[8]Nordin and Banzhaf called this “chunking” [NB96]; Salami et al [SMH96] have applied this technique in evolvable hardware based lossy compression

[JP96] H. Juille and J.B. Pollack. Massively parallel genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[Koz92] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[MW96] N. Memon and X. Wu. Lossless compression. In *CRC Handbook of Communication*. CRC Press, 1996.

[MW97] N. Memon and X. Wu. Recent progress in lossless image coding. *The Computer Journal*, 40:127–36, 1997.

[NB95] P. Nordin and W. Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In *Proceedings of the International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[NB96] P. Nordin and W. Banzhaf. Programmatic compression of images and sound. In *Proceedings of the Genetic Programming Conference*, pages 345–350, 1996.

[NG96] M. Nelson and J-L. Gailly. *The Data Compression Book (second edition)*. M&T Books, 1996.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 1992.

[PZ96] B. Punch and D. Zonker. lil-gp genetic programming system version 1.1 beta version. Michigan State University, http://GARAGe.cps.msu.edu/software/lil-gp/index.html, 1996.

[RYM93] R.F. Rice, P-S Yeh, and W.H. Miller. Algorithms for high-speed universal noiseless coding. In *Proc. of the AIAA Computing in Aerospace 9 Conference*, 1993.

[SIH97] M. Salami, M. Iwata, and T. Higuchi. Lossless image compression by evolvable hardware. In *Proc. European Conf. on Artificial Life*. MIT Press, 1997.

[SMH96] M. Salami, M. Murakawa, and T. Higuchi. Data compression based on evolvable hardware. In *International Conference on Evolvable Systems*. Springer Verlag LNCS, 1996.

[SMH97] M. Salami, M. Murakawa, and T. Higuchi. Lossy image compression by evolvable hardware. In *Proc. Evolvable Systems Workshop, International Joint Conference on Artificial Intelligence*, 1997.

[TC93] W. Tackett and A. Carmi. SGPC: simple genetic programming in C. ftp://ftp.io.com/pub/genetic-programming, 1993.

[VYL92] J. Venbrux, P-S Yeh, and M.N. Liu. A VLSI chip set for high-speed lossless data compression. *IEEE Trans. on Circuits and Systems for Video Technology*, 2(4), 1992.

[WM97] X. Wu and N. Memon. Context-based, adaptive, lossless image codes. *IEEE Transactions on Communications*, 45(4), 1997.

[WSS96] M.J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: A low complexity, context-based, lossless image compression algorithm. In *Proceedings of the Data Compression Conference (DCC'96)*, pages 140–149, 1996.

[YRM93] P-S Yeh, R.F. Rice, and W.H. Miller. On the optimality of a universal noiseless coder. In *Proc. of the AIAA Computing in Aerospace 9 Conference*, 1993.