RC23804 (W0511-130) November 16, 2005
Mathematics

IBM Research Report

Data Compression with Restricted Parsings

Peter A. Franaszek, Luis A. Lastras-Montaño,
Song Peng*, John T. Robinson**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Computer Systems Laboratory
Cornell University
Ithaca, NY 14850

**Worked performed while at IBM

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Data Compression with Restricted Parsings

Peter A. Franaszek[1], Luis A. Lastras-Montaño[1], Song Peng[2], John T. Robinson[1,*]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598, (paf, lastrasl@us.ibm.com).
[2] Computer Systems Laboratory, Cornell University, Ithaca, NY, 14850, (speng@csl.cornell.edu).

Abstract- We consider a class of algorithms related to Lempel-Ziv that incorporate restrictions on the manner in which the data can be parsed with the goal of introducing new tradeoffs between implementation complexity and data compression ratios. Our main motivation lies within the field of compressed memory computer systems. Here requirements include extremely fast decompression and compression speeds, adequate compression performance on small data block lengths, and minimal hardware area and energy requirements. We describe the approach and provide experimental data concerning its compression performance with respect to known alternatives. We show that for a variety of data sets stored in a typical main memory, this direction yields results close to those of earlier techniques, but with significantly lower energy consumption at comparable or better area requirements. The technique thus may be of eventual interest for a number of applications requiring high compression bandwidths and efficient hardware implementation.

I. INTRODUCTION

Most work on compression has concentrated on algorithms well suited to obtaining compression ratios ultimately targeted to the entropy of the data. Examples include arithmetic coding with adaptive context modeling [8], Lempel-Ziv coding algorithms [15], the Burrows-Wheeler transform [1], grammar based codes [7], etc. There are however applications where the constraints require compressors to operate on data blocks which are quite small by traditional measure, and where such compression needs to be done at extremely high bandwidths.

Prime examples of such an applications are to computer systems incorporating compressed main memory, such as for example in IBM's Memory Extension Technology or MXT (see [13][3][4] and references within). Here cache lines or small collections of cache lines may be compressed/decompressed on storeback/retrieval events. The larger the unit of data compression, the longer the access time and the larger the overhead. In practice, this means that such units may not exceed 1KB, the compression unit employed in MXT. The bandwidth requirements in current systems are in the GB/sec range, and require hardware implementations.

Compressors for such applications cannot do much learning on the data and must also be relatively simple. The problem of the design and implementation of fast hardware

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004

* Worked performed while at IBM (jtrobinson@optonline.net).

compression algorithms was considered as early as the work of Gonzalez-Smith and Storer [5] (see also the subsequent publication of Storer and Reif [11]). The schemes considered in this work are more closely related to LZ-like algorithms and their extensions [15][10] which also admit practical hardware implementations. Examples include ALDC [6] (IBM's implementation of LZ'77), and that used in MXT [4]. The latter is a parallel generalization of LZ'77 where the data is partitioned into N separate streams for N compressors, which however share a common dictionary. Both ALDC and MXT are implemented using content addressable memories (CAMs), which are memory units that are accessed by content instead of address [14].

CAMs offer the capability of immediate lookup of symbols, and their composition into phrases. For example, all locations holding a symbol i can be located in one cycle. In the following cycle, subset of locations immediately following those holding i can be located, and so on. Thus CAMs are convenient for implementing LZ-like techniques. However, the CAM approach has the following shortcomings: (1) limited pattern capacity due to the functional and design complexity, resulting in poor scalability; (2) slow access time and high power dissipation because of concurrent loopkups; (3) low storage density and high implementation cost per storage bit. All these factors make CAM-based designs unsuited for very fast data compression with the requirements of a small silicon budget and low energy consumption.

A key motivation for this investigation was to obtain a class of compressors which relied less heavily on CAMs. For example, SRAM, the other type of memory generally found on chips, has better density and power properties. Use of SRAM in particular suggests a compressor based on tables and entry hashing. Note that SRAM may be designed with multiporting, so that multiple retrievals may be done simultaneously. Complexity considerations lead to the idea of limiting the number of different phrase lengths used in the parsing, a scheme that falls in the general category of what we term restricted parsings. In principle, it could be possible to hash phrases of more than one length into the same table. However, this was not the approach adopted, which relies on a distinct table or set of tables for each of the chosen phrase lengths. The compressor then operates essentially by entering a subset of the phrases encountered into hash tables, which are then used as dictionaries in the restricted parsing. The contents of the hash tables are aged out in a manner analogous to the treatment of lines in a standard cache.

Restricting the set of parsings substantially simplifies the operation of the compressor. However, we are also interested in obtaining potentially several phrases in parallel. This we do by considering one fixed-length subblock at a time. This subblock is then represented by a set of phrases and literals, which are obtained at a rate of one subblock per cycle. In an implementation appropriate for present computer systems, the length chosen for the subblock was 8 bytes, with phrases restricted to lengths of 2, 4 and 8 bytes, aligned on respective boundaries.

The following is a synopsis of the paper. Section II discusses the format for parsing the data to be compressed. Section III describes the structure and operation of the encoder. Section IV considers a generalized version of the dictionary contents and section V describes an adaptive compression strategy. Section VI presents experimental results, which includes both compression ratio results on real data as well as a discussion on the hardware implementation cost. A comparison with LZ'77 indicates the approach

described yields compression ratios which are within reasonable distance with significant gains both in hardware area and energy consumption.

II. RESTRICTED PARSINGS

We start with some definitions.

Defn. A literal is an as-is representation of a symbol.

Defn. The unit of data to be compressed and/decompressed is termed a block. In the following, a block will generally consist of 512 bytes (512B).

Defn. A subblock is a fixed fraction of a block, which is then parsed into disjoint phrases.

Our goal is to parse the sequence using substantial parallelism. In the MXT algorithm [4], this was done by first partitioning the block to be compressed into several subblocks, which were then fed to separate parsers, which however shared their dictionaries. Here we proceed rather differently: as in MXT, we partition the sequence into subblocks, then each subblock is parsed into a restricted number of phrase lengths. The different components are then matched with a set of previously encountered phrase lengths stored in separate dictionaries. Unlike MXT, however, one subblock is parsed at a time at a throughput of one subblock per machine cycle. Once the subblock (say 8B as done here) is parsed, it is described by a combination of phrase length descriptors, pointers, and literals. This too can be done in one cycle. Note that these requirements imply that our subblocks are significantly smaller than those in MXT.

One property of this approach is that the encoding of the source symbols by the compressor proceeds at a fixed rate, which is advantageous from a system aspect. This is a property shared with the MXT algorithm. However, in the decoding stage of our new method the first (say) 8*N bytes of the original data are produced in N cycles, unlike the case in MXT, where each cycle produced results for every subblock, and the decoding of any given subblock is not finished until the entire block was decoded. This has some advantages in the retrieval of data in the compressed computer memory context, as with the new technique whole lines are available before the entire block decoding finishes.

The practicality of this method is dependent on finding a subblock size and restricted set of phrase lengths which combine good compression performance with simple hardware implementation. Here we took advantage of a particular property of the typical contents of computer memory, namely that data tends to reside on aligned boundaries; extensions of the basic algorithm to more general data sources are also discussed in this paper. Further, the blocks of data to be compressed are small, so that (except for long runs of identical subblocks), the candidate phrases were likely to be short. Thus our subblock length was set at 8B. This also ensured that our parsing and generation of output for each such phrase could be done in one cycle. The next question was that of choosing candidate phrase lengths. Possible candidates were of course all integers no larger than 8. An exception is runs of identical subblocks, which are treated separately.

Defn. In the following, a literal is equivalent to a phrase of length 1.

Defn. A parsing i is said to be better than another parsing j if has fewer phrases. We denote this as $i<j$. We say that $i=j$ if the two have the same number of phrases.

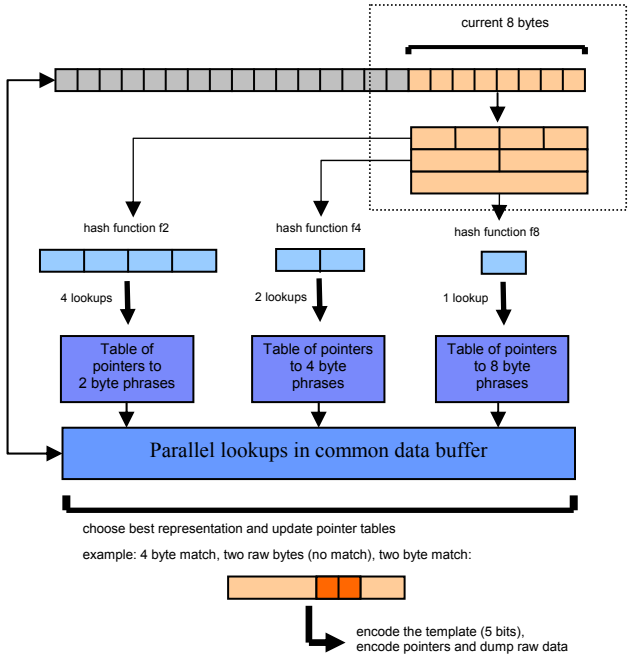

Figure 1. A family of parsings (enclosed in the box) and operation of the encoder.

Defn. A parsing i is optimal if for every parsing j in the allowed class, i<=j.

Defn. A set of parsings has a unique optimum if there are no two optimal parsings of a given subblock,

Defn. A feasible phrase is one which conforms to the restrictions of parsings within a subblock.

Note that our definitions assume that a parsing with fewer phrases has a smaller description cost than a parsing with more phrases. This assumption holds for a large class of settings including our target application. In the following, we shall denote a phrase of length n as Pn, and a literal as L.

Restricting the parsings to have a unique optimum has several advantages. These include a) the description of the parsings is more compact, so for example one does not need descriptors for both say (P4,P2,P2) and (P2,P4,P2), and b) no logic is required to make the choice between the competing solutions. For example, if both (P4,P2,P2) and (P2,P4,P2) are feasible optimal parsings for a particular subblock, this means that in this case one needs to distinguish between the two, at some cost in representation as well as logic to implement the choice.

Defn. A set of parsings is nested if any two distinct parsings of a subblock require that either each phrase in one is the same as the other, or that two or more comprise a phrase of the other.

Observation. A sufficient condition for a set of parsings to have a unique optimal is that they be nested.

A set of parsings with the above properties is illustrated in Figure 1, in the drawing enclosed by a box. For this example, the allowed phrase lengths are 8B, 4B, 2B and 1B. The latter are identified as literals, this is, they will be encoded as raw data instead of using pointers to past symbols. Examples of allowed parsings are (P8), (P4,P2,P2), (P2,LL,P4), (LL,LL,LL,LL), (P4,P4), etc. Not every concatenation of allowed phrase lengths is permitted. In general, assuming that the subblock length is M symbols, for this nested class of parsing constraints, the number of possible phrase lengths is $\log_2 M$, and M-1 phrases of length 2 or greater need to be examined in parallel if all feasible phrases are to be considered simultaneously.

III. ENCODER STRUCTURE

The general operation of the encoder can be found in Figure 1. Each successive subblock of M symbols is first partitioned into all M-1 feasible phrases of length two or greater that might be included in the parsing. Let #(M) denote the number of different parsings of the block of length M; it is easy to see that

$$\#(M) = 1 + \#(M/2)^2 \qquad (1)$$

with starting count #(2)=2. For the parsing class of Figure 1, we obtain #(8)=26. Note that phrases of lengths 4 can only be used at edges of the subblock due to the nesting property. In addition to the 26 possible states enumerated so far, it proves advantageous to incorporate additional states to encode other events such as runs of identical subblocks.

Each feasible phrase is examined for a possible match in past encoded data. Instead of using an associative memory to find this match, the encoder employs tables with pointers to past data that are looked up using hash functions tailored to each of the $\log_2 M$ phrase lengths. These hash functions accept a phrase of the appropriate length and return an index onto a dictionary row. After the retrieval of these pointers, a common buffer containing past encoded data is used for obtaining the actual phrases; this technique results in less storage redundancies when compared to an alternative which stores the phrases in the tables along with the associated pointers. The results are then combined to form a representation, which consists of the identity of the optimal parsing, pointers to previous occurrences, plus any literals. For our experimental section, the hash functions employed are simply a contiguous subset of the bits comprising the phrase of appropriate length.

Note one may implement similarly a close approximation to the standard Lempel-Ziv parsing if we include dictionaries for every possible phrase length, up to a certain threshold. Nevertheless this is associated with significant dictionary area and encoding complexity.

In some instances the data being encoded has sequences of two or more subblocks which are identical; we can capitalize on these patterns easily using run length coding. Our runs have a minimum granularity of a subblock; to encode them in addition to performing the matching described above we also compare the current subblock with the previously encoded subblock and increment a run length counter in case of a match. We continue the above until we find a non-matching subblock, and then simply encode a information about the existence of a run using a special state (in addition to the #(M) other necessary states), and also encode the run length.

Additionally, if the most recent encoding of a subblock is a full M symbol match we also compare both the current subblock and the subblock following this M symbol match. If

this succeeds we continue comparing subsequent subblocks until a mismatch occurs. The encoding is treated similarly as in the case of run lengths.

Decoding is done as in LZ'77, simply by replacing pointers to previous phrase occurrences by the actual phrases, as found in the already decoded data. The encoder outputs a representation of M symbols every cycle. Similarly the decoder outputs M symbols of decoded data on every cycle.

After the lookup, the encoding mechanism enters the feasible encountered phrases into the hash tables or dictionaries together with the associated pointer on every cycle. In a simple example we may define the encountered phrases as those that were looked up previously (this we term *restricted dictionaries*), but as we shall see, other useful possibilities exist. If there is a collision, the phrase currently occupying that location is replaced. It is thus important that the hash function yield good performance, making good use of the available table space by avoiding unnecessary collisions. We discuss this in greater detail below.

IV. Restricted parsings with unrestricted dictionaries

The initial insight for the idea of restricted parsings was the observation, as mentioned above, that data in computer memory is often stored on aligned boundaries as a consequence of the lengths of the fundamental data types. This led to the approach described in the previous section, where all phrases examined are aligned. However, for short block lengths, this constraint may be overly constrictive.

It is important to note that the fact that the encoding in this scheme is restricted does not imply that the phrases stored in the dictionaries need to come from restricted locations. For example, one may incorporate in the dictionaries all phrases of a given length regardless of their starting point; in this case we say that each of the dictionaries is *unrestricted.* Although physically such dictionaries must be capable handling insertions at a correspondingly increased rate, the encoder operation is similar to that outlined above.
This observation becomes particularly valuable when the data being compressed does not conform to the assumptions concerning alignment in data structures. Note that when our encoder operates with unrestricted dictionaries it becomes necessary for the pointers to consist of enough bits so as to address any location within the decoded data. This is in contrast with restricted dictionaries, where the pointers need only point to locations in the data that are aligned with the associated phrase length.

Further, unrestricted dictionaries could tax the capacity of the hash tables. That is, if the alignment assumptions are correct, the tables (if of limited size) could fill with irrelevant phrases. Thus one may expect that in some cases, the use of unrestricted dictionaries could degrade compression performance. In the following we discuss a class of algorithms that address these concerns.

V. Restricted parsings with adaptive mode selection

We assume that any given block either contains aligned data (such as data structures), suitable for encoding with restricted dictionaries or general data (such as text), which would in principle be benefited by the use of unrestricted dictionaries. As discussed, our practical block lengths are very small (512B), so that the data may be expected to be homogeneous within a block. Blocks are assumed to be randomly accessed and thus decisions of mode selection are made separately for each individual block. An interesting aspect of this problem is the tradeoff between the quality of the statistics measured by the adaptive encoder, which improves as more data is processed, and the overall compression gain which is favored when the *correct* mode is chosen *early*.

We present one heuristic solution to this problem. Initially we use unrestricted dictionaries and switch to restricted ones if enough evidence supports this decision. Thus initially all possible phrases (of the restricted lengths) are candidates to be incorporated in the dictionaries. Nevertheless, we bias the phrases to be associated with aligned (restricted) positions via the policy of disallowing the replacement of a phrase if the potential victim has an aligned pointer. This way we ensure that the phrases found through unrestricted dictionaries are a superset of those phrases found through restricted dictionaries.

For each dictionary D, the policy of restricted parsings with unrestricted dictionaries (as described above) lasts for a specific number N_T of phrases of length L (called the *training phase*), which may be different for varied dictionaries. During the training phase, we count (1) the phrases encoded using aligned pointers M_{ali} and (2) the phrases encoded using misaligned pointers M_{mis}.

At the end of the training phase, the mode selection decision is made for the dictionary D. A basic assumption is that after parsing N_T phrases the encoder can accurately predict the total number of new phrases that would be encoded with unrestricted dictionaries as

$$(M_{ali} + M_{mis}) \times N / LN_T \tag{2}$$

where N is the length of the remainder of the block in symbols. Similarly, we assume that the policy of restricted dictionaries would result in the efficient encoding of

$$M_{ali} \times N / LN_T \tag{3}$$

new phrases. The bit savings due to the encodings on either strategy can be estimated by making a reasonable assumption of what the average pointer length would be for the unrestricted dictionaries for the remainder of the block (a restricted dictionary pointer will always use $\log_2 L$ lesser bits than its unrestricted counterpart). For such assumption, one may average all pointer lengths that could potentially be encountered during the encoding; let P_L denote the estimate.

The task of the encoder is to choose the strategy that is expected to yield the greatest bit savings, which in the case a symbol is equal to a byte are

$$K \times (M_{ali} + M_{mis}) \times (8L - P_L) \quad \text{and} \quad K \times M_{ali} \times (8L - P_L - \log_2 L) \tag{4}$$

for the unrestricted and restricted strategies respectively. The symbol K denotes the common multiplicative factor in (2) and (3) which is irrelevant for purposes of the decision. Since one of the product terms is always independent of the data being encoded then the hardware implementation of this scheme is simple.

When there is a switch of strategies in a dictionary in general there may be changes in the statistics tracked for dictionaries of lesser block lengths. For example, if one stops the use of unrestricted dictionaries for the 8 byte dictionary, then the frequency of phrases encoded using 4 byte and 2 byte dictionaries may increase as a result. This suggests that assigning identical training phrases to all dictionaries may not be sensible, and instead one should try to allow for new training phases when a change in strategies occurs. On the other hand, a late decision to switch a strategy will have lesser effect on the overall performance, and thus a sensible balance must be made. For space reasons we do not present our full policy; it suffices to state that it follows the model set by the description above.

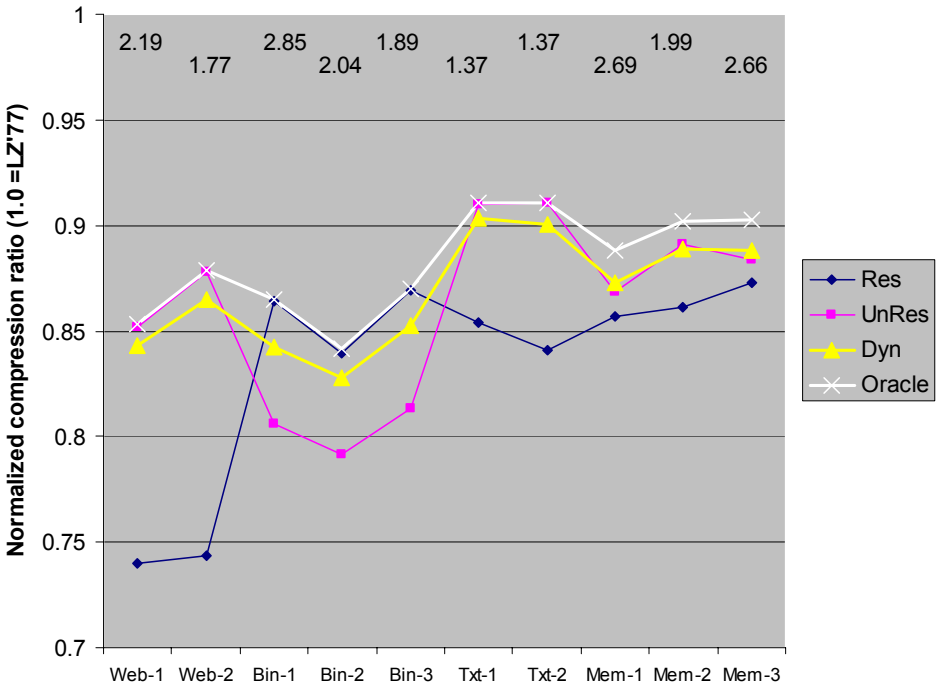

Figure 2. Normalized compression ratios of several benchmarks. The absolute compression ratio of LZ'77 is also reported at the top of the plot.

V. EXPERIMENTAL EVALUATION

We present experiments contrasting the new compression algorithm to a standard LZ'77 implementation with window size 512B and a symbol size of one byte (so that the length of all phrases is an integer multiple of a byte). Note that we do not compare to the MXT algorithm, which in general has a negligible performance degradation with respect to standard LZ'77 due to the parallel parsing. For the restricted parsings algorithm the number of entries in each of the three dictionaries is 758 (chosen so that the total storage budget is equal to 3KB).

Figure 2 shows the compression ratios of the restricted parsings algorithm with respect to different benchmarks and policies. Here, *Res*, *UnRes* and *Dyn* denote restricted dictionaries, unrestricted dictionaries and dynamic mode selection, respectively. In order to investigate the prediction accuracy of dynamic mode selection, we also show the results of restricted parsings with ideal mode selection, which tries both schemes and always chooses the best one for the entire block (shown as *Oracle*). All these results are normalized to the compression ratios of an implementation of the LZ'77 algorithm. The types of data considered for this experiment are: data resident in webservers (web), images of database application data structures (bin), english text from the Calgary corpus (txt) and general memory images (mem). From this data it can be seen that the restricted dictionaries method does not achieve the compression ratios of LZ'77, with losses (for the dynamic switching mode) in the order of 10%-17%. The data shows that restricted dictionaries are sometimes the simplest and best strategy among the ones considered (see the data for bin), but there are instances in which their performance suffers noticeably (web, txt). Similarly, unrestricted dictionaries are desirable for some benchmarks but not for bin. Finally, the dynamic switching data shows that it is possible to have a single

strategy that operates within a reasonable distance of an Oracle that can only be implemented by encoding the data twice.

An interesting question is to assess the performance degradation of our technique when a more stringent hardware budget is imposed. The following table illustrates the performance of our compressor when the budget is 1KB, which includes a 512B common data buffer and pointer dictionaries with approximately 150 entries each. The results are the relative compression ratios to those with 3KB hardware budget. It can be concluded that the method is robust to further limitations on hardware resources.

Web-1	Web-2	Bin-1	Bin-2	Bin-3	Txt-1	Txt-2	Mem-1	Mem-2	Mem-3
0.93	0.93	0.95	0.95	0.95	0.92	0.94	0.96	0.96	0.96

We now give a coarse estimate of the energy usage of an ALDC-type implementation of LZ'77 [6] and compare it to our method. The comparison accounts for the energy consumed after processing the entire 512 bytes. In ALDC, each byte is matched with the previously processed data, which in average across the block is approximately 256 bytes resulting in a total average of 1024K 1-bit CAM cell accesses/block (1K = 1024). For this SRAM-based approach, the processing of 8 bytes includes looking up pointers in dictionaries (seven 9-bit accesses), updating the dictionaries (same count for restricted dictionaries or 8 times this for unrestricted ones), and retrieving phrases from the common buffer (3*8*8=192 SRAM bit accesses). We shall account for the energy consumption of the various comparisons in our algorithm by doubling the latter. The total is approximately 32K 1-bit SRAM cell accesses/block for restricted dictionaries and 60K for unrestricted ones. An efficient implementation of a CAM cell includes an SRAM cell as a building block [14] and thus consumes at least as much power as this cell. Therefore it is safe to conclude that the energy consumption of a full LZ implementations is an order of magnitude higher of that of this method.

The hardware cost can be estimated in terms of layout area. Regarding this SRAM-based approach, the hardware cost primarily comes from the storage arrays of three dictionaries and the common buffer. Concurrent 8 reads/writes (writes always after reads) are expected for the case of unrestricted dictionary as well as dynamic mode selection. In order to reduce the hardware cost required for concurrent operations, banked SRAM is used for the storage implementation (instead of multi-ported SRAM cells). According to [16] a banked SRAM (with 1-read/write port) to simulate a register file with 8 read and 4 write ports only incurs ~25% speed loss (due to conflicts) while the layout area is reduced to 25-30%. For a SRAM cell with 8 read/write ports, the layout area is around 4 times of a baseline SRAM cell with 1 read/write port [14]. Thus, the layout area of the banked SRAM arrays (to simulate 8 concurrent reads/writes) is around 120% of that of baseline SRAM arrays. As to ALDC, a typical implementation (MXT [4]) uses 512-entry 1-byte CAM arrays with 4 read/write ports so that the processing speed is 4B(Byte)/cycle [6]. A baseline CAM cell (with 1 read/write port) is generally twice the area of a baseline SRAM cell, and a 4-port CAM cell can be six times the area of a baseline SRAM cell [14]. If the restricted parsing (with unrestricted dictionary) uses 3KB banked SRAM, the layout area is around 1.1~1.2 times of that of MXT. According to the previous table, the silicon cost (layout area) of our approach can be further reduced without significant compression loss. For instance, only 40% of MXT layout area is required if 1KB SRAM is used while the compression loss is only 6%. Note that these estimations are actually pessimistic to our approach, as the processing speeds of two approaches are different: the banked SRAM in our approach can achieve maximum processing speed of 8B/cycle (as long as there is no conflict) and the average speed of 6-7B/cycle (compared with 4B/cycle for MXT).

CONCLUSIONS

Motivated by the problem implfementation efficient, high-bandwidth data compression, we have introduced the idea of restricted parsings. This paper addresses the basic properties of these parsings and the question of whether they can have compression performance close to that of standard techniques such as LZ'77; this question is answered in the affirmative through the use of a combination of restricted parsings and various dictionary content management policies. Our method can be implemented with SRAM arrays. Compared with traditional CAM-based implementations, this approach results in comparable compression ratios, comparable silicon cost, lower design complexity, high processing speed as well as significantly less energy consumption.

ACKNOWLEDGMENT

The authors would like to thank Brett Tremaine of IBM Research for his insights regarding the implementation complexity of different compression techniques, and his encouragement to pursue alternative designs.

REFERENCES

[1] Burrows, M., Wheeler, D.J. **"***A Block-sorting Lossless Data Compression Algorithm"*, Research Report SRC-124, DEC, California, May 1994.
[2] Craft, D.J., "Method and Apparatus for Compressing Data", US Patent 5,652,878
[3] Franaszek, P.A., Heidelberger, P., Poff, D.E., Robinson, J.T., "Algorithms and data structures for compressed-memory machines", IBM Journal of Research and Development, vol. 45, No. 2, pp. 245-258, March 2001.
[4] Franaszek, P.A., Robinson, J., Thomas, J., "Parallel compression with cooperative dictionary construction", *Proceedings of*, pp. 200-209.
[5] Gonzalez-Smith, M.E. and J.A. Storer, "Parallel algorithms for data compression", *Journal of the ACM*, vol. 32, Issue 2, pp. 344-373, April 1985
[6] IBM Journal of Research and Development Topical Issue on Data compression in ASIC cores, Vol. 42, No. 6, 1998.
[7] Kieffer, J.C., Yang, E., "Grammar-Based Codes: A New Class of Universal Lossless Source Codes", *IEEE Transactions on Information Theory*, Vol. 46, No. 3, May 2000
[8] Langdon, G.G., "An introduction to arithmetic coding", *IBM Journal of Research and Development*, 28: 135-149, 1984.
[9] Ranganathan, N, Henriques, S., "High-Speed VLSI Designs for Lempel-Ziv-Based Data Compression", *IEEE Transactions on Circuits and Systems II*, Vol. 40, No. 2, February 1993.
[10] Storer, J. A., Szymanski T.G. "Data Compression Via Textual Substitution*", Journal of the ACM,* 29:4, (1982) 928-951.
[11] Storer, J.A. and Reif, J.H.. "A Parallel Architecture for High Speed Data Compression", *Journal of Parallel and Distributed Computing 13*, 222-227, 1992.
[12] Takeda, K., Aimoto, and Y., Nakamura, N. et al. "A 16-Mb 400-MHz Loadless CMOS Four-Transistor SRAM Macro", *IEEE Journal of Solid-State Circuits,* 35(11), 2000.
[13] Tremaine, R.B., Franaszek, P.A., Robinson, J.T., Schulz, C.O., Smith, T.B., Wazlowski, M. and Bland, P.M., "IBM Memory Expansion Technology (MXT)", *IBM Journal of Research and Development*, vol. 45, No. 2, pp. 271-285, March 2001.
[14] Weste, Neil and Harris, David, "CMOS VLSI Design: A Circuits and Systems Perspective", Addison Wesley Publisher, 2005.
[15] Ziv, J. and Lempel, A. "A universal algorithm for sequential data compression", *IEEE Trans. Inf. Theory* 23, 3 (1977), 337-343.
[16] Tseng J. H. and Asanovic, K., "Banked Multiported Register Files for High Frequency Superscalar Microprocessors", *Proceedings of International Symposium on Computer Architecture*, 2003