A Massively Parallel VLSI Design for Data Compression Using a Compact Dynamic Dictionary

James A. Storer
Computer Science Department
Brandeis University
Waltham, MA 02254

John H. Reif, Tassos Markas
Computer Science Department
Duke University
Durham, NC 27707

EXTENDED ABSTRACT

January 28, 1990

Summary:

Data compression is becoming an essential component of high speed data communications and storage. *Lossless* data compression is when the decompressed data must be identical to the original. *Textual substitution* methods are among the most powerful approaches to lossless data compression. We present a massively parallel architecture for textual substitution with a dynamic dictionary that is based on a systolic pipe of 3,839 identical processing elements that forms what is essentially an associative memory for strings that can "learn" new strings based on the text processed thus far. Key to the design of our architecture is the formulation of an inherently "top-down" serial learning strategy as a "bottom up" parallel strategy. A custom VLSI chip for this architecture that is capable of operating at 320 million bits per second has been recently fabricated. We discuss an improved buffer strategy that will provide the basis for a "compact" version of this design that that can achieve the same throughput rate with current technology (or higher rates by taking advantages of advances in technology) but with substantially less hardware.

1. Introduction

Lossless data compression is the process of encoding ("compressing") a body of data into a smaller body of data which can at a later time be uniquely decoded ("decompressed") back to the original data. Not all data is compressible. However, data that arises in practice typically contains redundancy of which compression algorithms may take advantage. The two most common applications of data compression are the following:

Data Communications: A sender can compress data before transmitting it and a receiver can decompress the data after receiving it, thus effectively increasing the data-rate of the communication channel.

Data Storage: A body of data is compressed before it is stored and decompressed when it is retrieved, thus increasing the effective capacity of the storage device. Our architecture is fast enough to accommodate the transfer speeds used by "super computers" such as the Connection Machine.

Preprint of paper appearing in VLSI Signal Processing, No. 4, 1990 (edited by H.S. Moscovitz and K. Yao and R. Jain), IEEE Press, 1990, New York, NY, pp. 329-338.

In Section 2 we discuss a particular form of *dynamic dictionary textual substitution*, a state of the art serial approach to data compression.

In Section 3, we discuss how the serial approach of Section 2 can be modified to yield equivalent compression and admit a simple massively parallel architecture based on a systolic pipe. The key technical idea is the formulation of an inherently "top-down" serial learning strategy as a "bottom up" parallel strategy. We have demonstrated feasibility in current technology of our architecture by designing a prototype VLSI chip containing 128 processing elements (patent pending). Thirty of these chips comprise a complete encoding / decoding pipe of 3,839 processing elements that is capable of operating at 320 million bits per second (a 40 meg hertz clock is used with a 8-bit data stream). These chips, together with some control logic, can be placed on a single board to form a complete system. The chip layout was done by the CDSR group at the Research Triangle Institute of North Carolina and fabrication was done through MOSES at Hewlett-Packard using 1.2 micron double-metal CMOS technology.

Even assuming that advances in technology can increase the number of processors that can be placed on a single chip, it is of both practical and theoretical significance to consider how the inherent amount of hardware can be reduced. In the Section 4, we discuss an improved buffer strategy that will provide the basis for a "compact" version of this design that that can achieve the same throughput rate with current technology (or higher rates by taking advantages of advances in technology) but with substantially less hardware.

2. Textual Substitution Methods

To simplify our discussion we henceforth assume all *data* to be a sequence of 8-bit *characters* and we refer to the set of all possible characters (the integers 0 through 255) as the *input alphabet* Σ. In addition, we assume that all handling of the data while it is in compressed form is *noiseless* (no bits are added, lost, or changed). We make this assumption for convenience; a host of techniques are available in the literature for error detection and correction.

We employ an *on-line* model for data compression where the *encoder* and *decoder*, each have a fixed (finite) amount of local memory, which we refer to as a *local dictionary*, D. What distinguishes this model from an *off-line* model is that neither the sender or the receiver can see all of the data at once; data must be constantly passing from the sender through the encoder, through the decoder, and on to the receiver.

A powerful (and practical) approach to compression of a string of characters is *textual substitution*. For a general treatment of textual substitution techniques as they apply to both on-line and off-line algorithms, as well as further references on the subject, see the book of Storer [1988]. The basic idea is to use the local dictionary D to store a constantly changing set of

strings. Data is compressed by replacing substrings of the input stream that also occur in D by the corresponding index into D; we refer to such indices as ***pointers***. Thus, the input to an on-line data compression algorithm employing textual substitution is a sequence of characters and the output is a sequence of pointers (where typically, most of the pointers specify strings of length greater than 1 but a few are essentially character codes).

Most on-line textual substitution algorithms can be viewed as instances of one general approach. The following page shows a generic ***encoding algorithm***, which reads a stream of characters over Σ and writes a stream of bits, and a generic ***decoding algorithm***, which receives a stream of bits and outputs a stream of characters over Σ. Both algorithms use $|D|$ to denote the current number of entries in D. In addition, we shall use $< D >$ to denote the maximum number of entries that D may contain (all of the algorithms we discuss employ a data structure that uses constant space per entry, independent of the length of the string to which it corresponds). We shall use $< D >= 3839$ for the architecture presented here (which when combined with the 256 characters of Σ and a special pointer to be discussed later, makes a total of 4,096 distinct pointer values).

The encoder and decoder work in lock-step to maintain identical copies of D (which is constantly changing). The encoder repeatedly finds a match between the incoming characters of the input stream and the dictionary, deletes these characters from the input stream, transmits the index of the corresponding dictionary entry, and updates the dictionary with some heuristic that depends on the current contents of the dictionary and the match that was just found; if there is not enough room left in the dictionary, some deletion heuristic must be performed. Similarly, the decoder repeatedly receives an index, retrieves the corresponding dictionary entry as the "current match", and then performs the same algorithm as the encoder to update its dictionary.

Our architecture is based on the following choices for heuristics:

MH: The *greedy heuristic* is to always take the longest possible match between the input stream and a string in D. Key to our architecture will be a "bottom up" method for approximate determination of the longest match.

UH: The *identity* (ID) heuristic is to add the previous match concatenated with the current match. Key to our construction will be a modified version of the ID heuristic to be discussed later.

DH: The *swap* ($SWAP$) heuristic works as follows: When the *primary* dictionary first becomes full, an *auxiliary* dictionary is started, but compression based on the primary dictionary is continued. From this point on, each time the auxiliary dictionary becomes full, the roles of the primary and auxiliary dictionaries are reversed, and the auxiliary dictionary is reset to be empty. Although this heuristic does not fit directly

(1) Initialize the entries of the local dictionary D to be empty (the entries 0 to $|\Sigma|$ are always assumed to be the characters of Σ and are not explicitly stored).

(2) **repeat forever**

(a) Get the current match:

$t := \mathrm{MH}(\textit{input stream})$

Advance the input stream forward by $|t|$ characters.

Transmit $\lceil log_2|D|\rceil$ bits corresponding to t.

(b) Update the local dictionary D:

for each string s in UH(D) that is not in D **do**
if (D is not full) **or** (DH(D) is not empty) **then** add s to D

Generic Encoding Algorithm

(1) Initialize D by performing Step 1 of the encoding algorithm.

(2) **repeat forever**

(a) Get the current match:

Receive $\lceil log_2|D|\rceil$ bits.

Retrieve the current match t from the dictionary.

Output the characters of t.

(b) Update D by performing Step 2b of the encoding algorithm.

Generic Decoding Algorithm

NOTE:

MH = "Match Heuristic" (reads an entry of D from the input stream)

UD = "Update Heuristic" (adds new entries to D)

DH = "Deletion Heuristic" (deletes an old entry from D)

into the generic encoding and decoding algorithms, they can be modified to accommodate it.

The general approach of dynamic dictionary data compression that is employed in this paper has its beginnings with the work of Ziv and Lempel [1978] and such approaches are often referred to as "LZ2" type methods to distinguish them from "sliding window" methods, which are addressed in Ziv and Lempel [1977] and often referred to as "LZ1" methods. Both "LZ1" and "LZ2" methods are optimal in the information theoretic sense when given unbounded resources. However, in practice, the approaches are very different. The book of Storer [1988] discusses in more detail the heuristics presented above and includes additional references to textual substitution techniques in general (e.g., Storer and Szymanski [1982]), sliding window methods (e.g., Gonzalez and Storer [1985]), and dynamic dictionary methods (e.g., Miller and Wegman [1985], Storer [1985], Welch [1984], Seery and Ziv [1977,1978]). Zito-Wolf [1990] presents several massively parallel architectures for sliding window compression and compares them to that of Gonzalez and Storer [1985].

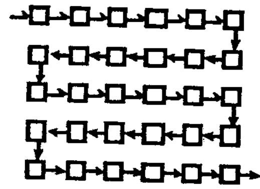
3. A Parallel Implementation

The model of parallel computation employed by our algorithm is a *systolic pipe*; that is, a linear array of processors, each connected only to its left and right neighbors. Systolic pipes are perhaps the most practical model of massively parallel computation to realize in custom VLSI. Here we describe how a systolic pipe can be used to maintain a dynamically changing dictionary based on a variant of the ID update heuristic, the $SWAP$ deletion heuristic, and a "bottom up" parallel version of the greedy match heuristic. The same hardware can perform both encoding and decoding. On the following page, the first figure depicts a standard "snake" layout for a systolic pipe, the second shows an individual processor, and the third shows how power and ground for the processors can easily be added to the snake.

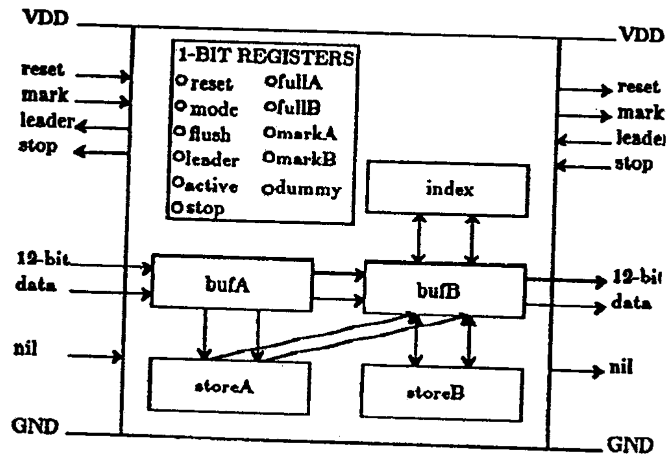
3.1. Encoding

Here we describe a systolic pipe implementation only as it pertains to encoding; unlike the serial case, here, decoding is the more complicated operation and is discussed in the full draft.

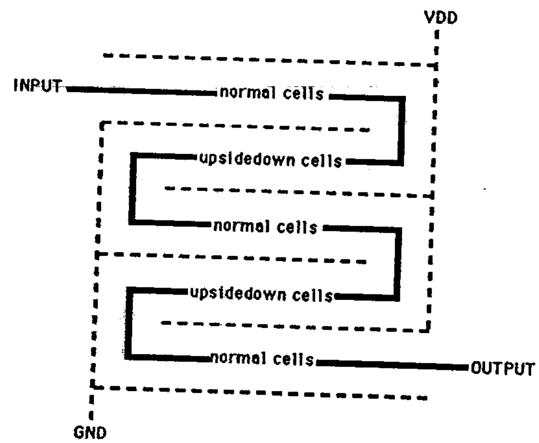
All pointers are represented by the same number of bits; the bits for each input character are padded to the left with zeros to form the corresponding pointer and sent into the left end of the pipe (for our chip, each input character of 8 bits is padded to the left with 4 zeros to form a 12 bit pointer). The pipe consists of $< D > -|\Sigma| - 1$ processors numbered $|\Sigma|$ through $< D > -2$ going from left to right; there are no processors for indices 0 through $|\Sigma| - 1$ since the characters of Σ are always implicitly in the dictionary (we shall discuss



"Snake" Layout



An Individual Processor



Power and Ground Distribution

later the remaining unused pointer with value $< D > -1$). The processors are numbered so that if $i < j$, then processor i is to the left of processor j (and occurs earlier in the pipe than processor j).

The pipe stores a "pair forest" representation of the dictionary (each dictionary entry is represented by a pair of pointers to two other entries or to single characters) and implements a modified version of the ID heuristic. Each processor is capable of holding a pair of pointers (corresponding to the left and right pointers into the pair forest), but is initialized to be empty. The leftmost processor is initially the *leader*. It is always the case that all processors to the left of the leader contain dictionary entries and that all processors to its right are empty. Note that unlike decoding (where a leader bit is used), leadership is not explicitly passed to the right; rather, each empty processor "thinks" it is the leader, but only the leftmost empty processor will be able to learn a new entry.

As data passes through the processors to the left of the leader as it is encoded. That is, whenever a pair of pointers enter a processor and matches the pair of pointers stored at that processor, this pair of pointers is removed from the data stream and replaced by a single pointer (the index of that processor). Data passes unchanged through the processors to the right of the leader. When a pair of pointers enters the leader, they represent adjacent substrings of the original data and can be viewed as the "previous" and "current" match. The leader can simply adopt this pair of pointers as its entry; these pointers are then marked (via the mark bit) so that the processor to the right (which is now the leader) will not adopt the same pair again. Note that unlike the serial algorithm, this is a "bottom up" approach to finding a longest match, since bigger matches are built from smaller ones.

When the rightmost processor becomes the leader, it adopts a pair of pointers as its entry and the dictionary is now full. At this point, the $SWAP$ deletion heuristic can be implemented with a switch between two copies of the pipe that routes input/output lines appropriately as the dictionaries turn over.

4. An Improved Buffer Strategy

From inspection of the layout of the chip described earlier, it can be seen that:

- Approximately 45 percent of the area of each processor is the finite state control.
- With the current hardware design, a parallel match between $BufA, BufB$ and K stored pointer pairs should be possible in the same clock cycle that is currently used to match $BufA, BufB$ to the single pair $StoreA, StoreB$, for small values of K (e.g., $K = 16$).

In this section we describe how K processors can be combined to share a single finite state control and reduced buffer space with no loss of throughput under the assumption that a pair of pointers can be matched in parallel to K other pairs in a single clock cycle. Note that $K = 16$ appears to be a practical choice. Even with the ability to match a pair of pointers to K stored pointers in parallel in a single clock cycle, problems to be overcome include:

- Within any block of K dictionary elements, there may be pointers that point to other entries in that same block.
- It is possible K consecutive processors to match on a particular clock cycle.

Let us refer to the hardware described in the previous section as the "pure" design (one processor per dictionary element) and to an architecture where K dictionary elements are stored per processor (so only $1/K^{th}$ the processors are needed) as a "K-compacted" architecture. A naive brute force approach for a K-compacted architecture is to simply have for $1 \leq i \leq K$ the registers $BufA[i]$, $BufB[i]$ $Store[i]$, $StoreB[i]$ $Index[i]$ and for each clock cycle of the pure architecture, use K clock cycles to simulate the actions of the pure architecture. We begin by describing a simulation that is shown on the following page of a clock cycle of the pure architecture that still has a factor of K slow-down but uses only K buffer registers (corresponding to the $BufB$ registers) together with a single new register called $Carry$. The idea is to "percolate" each input character through the K buffers so that the simulation is functionally identical to the corresponding K processors in the pure architecture (a given input gives the same output) but at any point in the simulation, the carry register will have the value that would have been in the $BufA$ register corresponding to the current $BufB$ register. To simplify notation, we refer to the pair $StoreA[i], StoreB[i]$ as $Store[i]$ and to $BufB[i]$ as $Buf[i]$. We assume that the next input pointer is in a register $Buf[0]$ and that the output is place in a register $Buf[K+1]$. We use $||$ to denote concatenation.

Although the above simulation uses only half the buffer registers, it still has a factor of k slowdown since in the worst case the **while** loop must percolate the $Carry$ register through all k buffer registers. It is useful however, to note how the brute-force simulation works. The Buf registers represent the $BufB$ registers from the pure architecture and learning for $Store[i]$ occurs only when the buffer contains i pointers (so that together with the pointer in $Carry$, $i+1$ pointers are present before $Store[i]$ is learned).

We now consider how the above brute force simulation can be improved from a factor of K slow-down to a factor of 2 slow-down by employing a parallel match to the K storage pairs (later we shall make a further improvement to eliminate the factor of 2). We say that the contents of the Buf registers is *reduced* if no matches are possible between any two adjacent pointers in the buffer and the storage registers. To keep the buffer in reduced form, each

```
{initialize Carry to the input pointer}
i:=0
Carry := Buf[i]

{percolate the Carry register to the right}
while Carry is not empty do begin
      i := i + 1
      if Buf[i] is empty then begin
            Buf[i] := Carry
            make Carry empty
            if i ≤ K then mark Buf[i]
            end
      else if Buf[i] not marked and Carry||Buf[i] matches Store[i] then
begin
            Buf[i] := Index[i]
            make carry empty
            end
      else exchange Buf[i] and Carry
      end

{learn a new entry}
if Carry is not empty and Buf[i] is marked then begin
      Store[i] := Carry||Buf[i]
      Buf[i + 1] := Buf[i]
      Buf[i] := Carry
      mark Buf[i] and Buf[i + 1]
      end
```

Brute Force Simulation using K buffer registers

time a new input character arrives, the first two pointers in the buffer can be compared in parallel to the storage registers and replaced with a single pointer if a match is found; this process must be repeated until a match is not found (because it may be that some storage register pairs contain pointers to others). Since each time a match is found, the size of the buffer is reduced by 1, there is room to accept another input pointer on the next clock cycle. However, when backing up to the left (to process new input pointers that have arrived during a sequence of reductions), the input could back up. It is not hard to show how this can be remedied by slowing down the clock by a factor of 2. It is possible to construct inputs where the use of the parallel match gives a slightly different output than the pure architecture (in fact, the parallel matching can yield slightly better compression); however, in practice, the difference is negligible. The key observation is that learning for $Store[i]$ is still done only when $i + 1$ pointers are currently buffered.

The final improvement, which is described in detail in the full draft, is

to eliminate the factor of 2 slow-down by re-sequencing the learning order. If we take $K=16$ and assume that a parallel match to 16 storage registers is possible in one clock cycle, the net result is an architecture that runs as fast as the pure one but has only one finite state control for every 16 dictionary elements as well as reduced space for buffer registers and index registers (since the 12 high-order bits are the same for all index registers in a group of 16). Thus, the original 45 percent area for finite state control is reduced to 3 percent and the original 55 percent for data path and registers (with some additional improvements discussed in the final draft) is reduced to 30 percent, for a net decrease to 1/3 of the area required by the pure version. Further improvements may be possible with more compact storage logic for a collection of registers than is currently being used for a single register.

By limiting the recursion within a block of K registers, further simplifications of the hardware and increased speed are possible without sacrificing the amount of compression obtained in practice; this approach is discussed in the full draft. The full draft also presents experimental simulations that compare the strategies that we consider as well as design details of the hardware required.

5. References

M. Gonzalez and J. A. Storer [1985]. "Parallel Algorithms for Data Compression", *Journal of the ACM* 32:2, 344-373.

A. Lempel and J. Ziv [1976]. "On the Complexity of Finite Sequences", *IEEE Transactions on Information Theory* 22:1, 75-81.

V. S. Miller and M. N. Wegman [1985]. "Variations on a Theme by Lempel and Ziv", *Combinatorial Algorithms on Words*, Springer-Verlag (A. Apostolico and Z. Galil, editors), 131-140.

J. B. Seery and J. Ziv [1977]. "A Universal Data Compression Algorithm: Description and Preliminary Results", Technical Memorandum 77-1212-6, Bell Laboratories, Murray Hill, N.J.

J. B. Seery and J. Ziv [1978]. "Further Results on Universal Data Compression", Technical Memorandum 78-1212-8, Bell Laboratories, Murray Hill, N.J.

J. A. Storer and T. G. Szymanski [1982]. "Data Compression Via Textual Substitution", *Journal of the ACM* 29:4 928-951.

J. A. Storer [1985]. "Textual Substitution Techniques for Data Compression", *Combinatorial Algorithms on Words*, Springer-Verlag (A. Apostolico and Z. Galil editors), 111-129.

J. A. Storer [1988]. *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD.

T. A. Welch [1984]. "A Technique for High-Performance Data Compression", *IEEE Computer* 17:6, 8-19.

R. Zito-Wolf [1990]. "A Systolic Architecture for Sliding Window Data Compression", 1990 IEEE Workshop on VLSI Signal Processing, San Diego, CA.

J. Ziv [1985]. "On Universal Quantization", *IEEE Transactions on Information Theory* 31:3, 344-347.

J. Ziv and A. Lempel [1977]. "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory* 23:3, 337-343.

J. Ziv and A. Lempel [1978]. "Compression of Individual Sequences Via Variable-Rate Coding", *IEEE Transactions on Information Theory* 24:5, 530-536.