The Mixed Serial/Parallel Approach to VLSI Search Processors

Behrooz Parhami

Dept. of Electrical & Computer Engineering
Univ. of California
Santa Barbara, CA 93106, USA

ABSTRACT

Associative memories (AMs) and algorithms for performing various operations on them have been studied extensively for more than three decades. In recent years, numerous VLSI-based dictionary machines (DMs), that are essentially limited types of associative memories, have been proposed and theoretically evaluated. In both cases, the proposed designs are generally hardware-intensive and postulate the use of one processing element for each record or for a small number of records. Given their special-purpose natures and limited applications, such designs are not technically feasible or economically viable with today's VLSI technology. Even when future VLSI advances do allow the construction of such megaprocessor machines, other architectures and implementation alternatives must be investigated in search of an optimal design for a given set of requirements. In this paper, we show that a mixed serial/parallel design methodology offers a wide range of cost versus performance tradeoffs and results in AM or DM systems that are practically realizable with today's VLSI technology. We discuss a design methodology for VLSI search processors whereby most of the processing elements of previous designs are replaced with high-speed shift registers. Such an approach results in a significant cost reduction while maintaining a reasonable suboptimal processing speed for databases of practical interest. As an added benefit, the proposed designs offer considerable flexiblity in the handling of variable-length and very long records. Such records are impossible to handle or lead to intolerable performance penalties with fixed-format designs. A key to the proposed architecture is the coupling of high-speed shift registers with systolic string comparators that can operate at extremely high clock rates. The speed/cost tradeoffs provided by various architectural features of the proposed system are discussed and its performance is compared to those of certain theoretically optimal but currently unrealizable hardware-intensive architectures.

Keywords: Algorithms, Array processors, Associative devices, Content-addressable memories, Database machines, Dictionary machines, Information retrieval, Parallel processing, Pipelining, Search processors, SIMD concurrency, String matching, Systolic design, VLSI-based design.

1. Introduction

Searching is an important subproblem in many computing applications where its speed and efficiency affect the respective parameters of the overall solution. No direct statistics on the prevalence of searching is available to the author. However, Knuth suggests that over 25 percent of the running time of modern computers is spent on sorting and that in many systems sorting uses more than half of the computing time [KNUT73]. Since a primary motivation for sorting is to simplify searching and since searching can also be done without sorting, it is safe to say that searching consumes directly or indirectly well over a quarter of our available computational power. Therefore, techniques for speeding up different types of searches that are used in practice can have significant performance implications for digital systems. Accordingly, computer designers have investigated the use of special-purpose hardware devices and subsystems to make searching faster and more efficient.

Associative memories (AMs), database computers (DBCs), and more recently, VLSI-based dictionary machines (DMs) have received considerable attention in the literature [PARH90a]. In this paper, we refer to these systems collectively as "search processors". Although DBCs are in general capable of much more than just searching a database, here we are only interested in their searching capabilities/mechanisms that are quite similar to those of AMs and DMs. Proposed designs for search processors are generally hardware-intensive and postulate the use of one processing element for each record or for a small number of records. Given that search processors are special-purpose and have limited applications, such designs are not technically feasible or economically viable with today's VLSI technology. Even when future VLSI advances do allow the construction of megaprocessor machines, other architectures and implementation alternatives must be investigated in search of an optimal design for a given set of requirements.

In its simplest form, an *associative memory* (AM) is a hardware device consisting of N fixed-size cells, each storing a data word or record or marked as empty (Figure 1). When presented with a *search key* or *comparand*, a *mask* showing the relevant field(s) of the stored words, and possibly an *instruction* specifying the type of search, the AM responds by *marking* all the words that *match* the given key. Marking is done by setting or resetting a *response bit* or *tag (bit)* in the AM cell. The N response bits form the AM's *response store*. A *response indicator* mechanism may provide information on the multiplicity of responders (zero, one, several). If there is no responder, the search outcome is negative and we can proceed to the next step of our algorithm. With one responder, the required data item has been located and can be appropriately dealt with by reading it out or modifying it in-place. If there are several responders, the appropriate action might be proceeding to further narrow down the search, simultaneously modifying all responders in-place, or examining the responders in turn through a *multiple response resolver*.

Research on associative memories started in the mid 1950s. Early associative memory proposals were in the "fully parallel" class whereby processing logic is associated with every bit of the memory array; thus all stored data bits are simultaneously compared to the comparand bits and the bitwise match results combined to obtain the word-match indicators. Soon it was realized that "bit-serial, word-parallel" designs (sometimes designated simply as "bit-serial") that read out a complete *bit slice* of data into processing elements external to the memory array are much more practical since they take advantage of the high density and low cost of conventional memories and also

can be designed to perform parallel operations other than simple searches at little extra cost. This led to a large number of actual implementations, including Goodyear's well-known STARAN system. More recently, the Connection Machine [HILL85] has utilized such a bit-serial organization. Although some "bit-parallel, word-serial" designs ("word-serial" for short) based on shift registers were also proposed in the early years, they all turned out to be impractical for real applications. A fourth alternative organization, the so-called "block-oriented" design, that emerged in the early 1970s [PARH72], became the basis for many database machine designs.

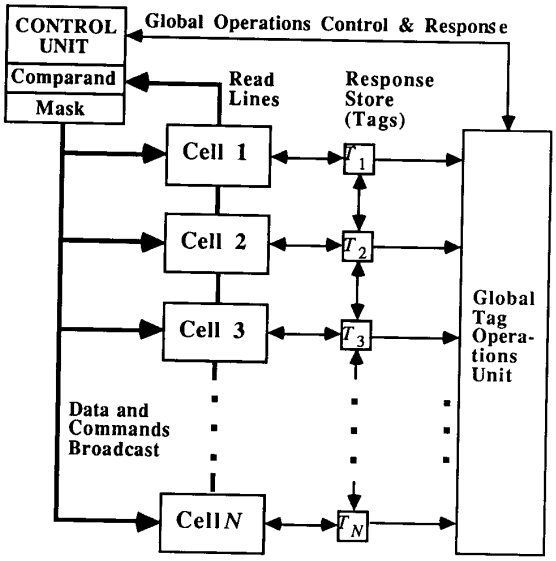

Figure 1. Functional View of an Associative Memory.

Many algorithms have been developed for performing search, retrieval, and arithmetic/logic operations on data stored in AMs [PARH90a]. A common flaw in the analyses offered by designers of AMs and of AM algorithms is the assumption that the basic cycle time of an associative memory is a constant independent of size, thus leading to the optimistic conclusion that an N-word AM offers n-fold speedup compared to linear search in an unordered list of n items ($n \leq N$). While this may be accurate for small values of N, it is grossly unfair to equate the cycle time of a large associative memory (say, one holding hundreds of thousands of data items) with that of simpler hardware used in conventional sequential systems to which AM designs are compared. Realistic analyses of the speedup offered by AMs and AM algorithms would require the development of a model that predicts the cycle time of an N-word AM as a function of N. Such a model would depend on the technology used to implement the AM and on its architecture. The dependence of cycle time on N arises from:

1. Broadcasting instructions to all N cells in the AM.

2. Performing global operations on N response bits.

The required cycle time for broadcasting obviously increases as we go from single-chip to multiple-chip and multiple-board systems, simply because longer interconnections will be involved and a larger number of loads must be driven. As we move towards widespread utilization of advanced submicron technologies for VLSI, the effect of long wires is becoming important even within a single chip. The transmission delay on such wires on a chip is dominated by their RC time delay and can easily overshadow data manipulation times which are functions of logic gate delays. Table I shows the RC-delay and gate-delay values for the MOS technology.

As for global operations, such as those required for multiple response resolution, the effect of long wires is compounded by the requirement for logic manipulation. It is well-known that even if propagation delays are insignificant, a global operation on N bits by limited fan-in circuits requires time that is at least proportional to log N. Yet even recent works on associative memory architectures and algorithms claim "constant time" for N-operand global operations such as logical OR and logical AND for which the use of "wired logic" is generally assumed (see e.g. [DAVI86] and [LEED88]).

It follows that the scalability issue for AMs must be explicitly addressed if large units are to be realized with the VLSI technolgy. Several authors have dealt with the VLSI realization of AMs [WEEM82], [SIVI85], [HURS87]. However, the scalability issue has only been hinted at by Hurson and Shirazi [HURS87] who suggest that "large" AMs can be built by taking advantage of wafer-scale integration and/or a bus-based interconnection of smaller modules. In this paper, we show that a mixed serial/parallel design methodology combined with a pipelined mode of operation not only leads to scalable designs but also offers a wide range of cost versus performance tradeoffs resulting in AMs or DMs that are practically realizable with today's VLSI technology.

We discuss a design methodology for VLSI search processors whereby most of the processing elements of previous designs are replaced with high-speed shift registers. Such an approach results in a significant cost reduction while maintaining a reasonable suboptimal processing speed for databases of practical interest. As an added benefit, the proposed designs offer considerable flexiblity in the handling of variable-length and very long records. Such records are impossible to handle or lead to intolerable performance penalties with fixed-format designs. A key to the proposed architecture is the coupling of high-speed shift registers with systolic string comparators that can operate at extremely high clock rates. The speed/cost tradeoffs provided by various architectural features of the proposed system are discussed and its performance is compared to those of certain theoretically optimal but currently unrealizable hardware-intensive architectures.

The remainder of this paper is organized as follows. Section 2 is devoted to a discussion of dictionary machines (DMs), a class of simple search processors that will be used as a vehicle for making the architectural descriptions and analyses that follow understandable as well as more concrete. In Section 3, we discuss the serial/parallel design paradigm by presenting an example design for a simple DM (with only "search", "insert", and "delete" instructions) based on a combination of high-speed, low-cost shift registers and systolic string comparators. This is followed in Section 4 by an evaluation of the DM design in terms of two key performance attributes: The response latency L and the pipelining period P. Section 5 deals with a generalization of the serial/parallel design paradigm to multi-level systolic search processors with more extensive instruction sets. Counting of responders and shifting of response bits in an AM are used to illustrate additional problems that need to be dealt with. Finally, concluding remarks and ideas for further research are offered in Section 6.

2. A Class of Search Processors

Although the serial/parallel design paradigm presented in this paper is applicable to many searching problems and the associated search processors, we initially limit our presentation to dictionary machines for the sake of clarity. Extensions to the ideas presented, which would be required for realizing more complicated searches and related operations, are discussed in Section 5 of the paper.

A *dictionary machine* (DM) is a device that stores a set of records, each having a unique *key* with some associated information (*info*), and supports the following basic set of operations:

- search(*key*, *info*): Return the *info* associated with the given *key*, if stored; return *null* if not.
- insert(*key*, *info*): Add a (new) record with the given *key* and *info* to the database.
- delete(*key*, *info*): Delete the (stored) record with the given *key* from database; return the *info*.

Actually, the standard form of the *delete* operation only needs a single parameter (the *key*). However, for the sake of uniformity, we assume the the DM returns the *info* part of a record when it is deleted. With this convention, the *search* operation can be viewed as a *delete-insert* pair, where insertion is done only if deletion is successful. Ideally, a DM should be able to deal with duplicate insertions (attempting to insert a record that is already stored) and redundant deletions (attempting to delete a nonexistent record) by treating them as no-operations. However, not all proposed DMs have such a capability.

In addition to the above basic set, many other operations have been suggested for DMs, depending on the requirements of particular applications or the capabilities inherent in suggested hardware implementations. Here is a sample:

- findbest(*key*, *info*): Return the *info* part of a record whose key best matches the given *key*.
- findmin(*key*, *info*): Return the smallest stored *key* along with its associated *info*.
- deletemin(*key*, *info*): Delete the record that has the smallest key; return both *key* and *info*.

In the *findbest* operation, the criteria for judging the "best" match are application-dependent. Some possibilities are "closest" and "next higher" for numeric keys or "smallest edit distance" or "longest substring match" for bit and character strings. The *findmin* and *deletemin* operations above are required operations for a "priority queue" (standard *deletemin* does not return any value) and can be added to the operation set of a sorted-key DM with little or no additional cost. Besides the record having the smallest key, the one with the largest key (*findmax*, *deletemax*) or the median-value key (*findmed*, *deletemed*) may be dealt with in some applications.

Many VLSI-based DMs have been proposed in the literature. These machines typically realize the basic set of DM operations, plus possibly some of the operations in the extended set, by using one processor for each record or for a small number of records. The processors are connected in a tree [ATAL85], [BENT79], [BROW79], [CHAN88], [GOYA88], [OTTM82], [SCHM85], [SOMA85], [SONG80], hypercube [DEHN88], [OMON87], [SCHW87], or mesh [DEHN87], [PROB87], [SCHM85]. In each case, pipelining or systolic operation is used to improve the system performance for multiple searches by allowing a new instruction to be pumped into the system with every clock tick, while the response latency for individual searches may be quite large (perhaps hundreds or thousands of clock cycles). Thus, many searches are processed concurrently. Consistent handling of search, insert, and delete operations in view of the inter-operation concurrency leads to many design problems that have been studied by numerous researchers.

The response latencies of tree- and hypercube-based DMs are logarithmic (ignoring of course the important effects of layout and long wires) while that of mesh-connected machines is realistically proportional to the square root of the number N of processors or the number n of stored records, depending on design. Strictly speaking, tree-structured systems cannot be systolic because they require arbitrarily long wires as the number of nodes grows [PATE81]. However, the wire length growth rate appears to be manageable for DM sizes likely to be of interest in the foreseeable future. The above issue is much more serious for the hypercube interconnection scheme for which the requirement of $\lceil \log_2 N \rceil$ connections to each processor compounds the scalability problem. The excessive cost of the one-processor-per-key approach, along with its implicit assumption of short fixed-length keys, has led to one proposal advocating the use of a relatively small number of processors equal to the maximum key length. The resulting "radix machine" [FISH84] consists of an array of conventional processors (with large random-access memories) executing instructions in a pipelined mode (see also [CARE84]).

Like any other data management problem, an application requiring dictionary operations can be implemented in a variety of ways; from software implementation on a sequential computer using conventional data organization and search algorithms [KNUT73], [MEHL84] at one extreme to the use of hardware-intensive DMs at the other. The serial/parallel approach presented in this paper provides additional points in the design space of special-purpose DMs. We will only consider the three basic DM operations in detail. Implementation of the other operations are briefly discussed at the algorithm level without much attention to hardware design considerations.

3. A Serial/Parallel Architecture

In its simplest form, the hardware of the proposed DM consists of a linear array of p processing elements, $P_1, P_2, \ldots, P_p$. As shown in Figure 2, processing element P_i is connected to a circular shift register (loop) having the register elements $R_{i,1}$, $R_{i,2}, \ldots, R_{i,l}$ and to a systolic comparator of length k consisting of the cells $C_{i,1}, C_{i,2}, \ldots, C_{i,k}$. Each $R_{i,j}$ holds one symbol from a finite record alphabet. Records have the format

$$\langle \textit{key-string} \rangle\ \phi\ \langle \textit{info-string} \rangle\ \rho$$

where ϕ is a special field separator and ρ is an end-of-record marker. It will be assumed that the empty space in each shift register is organized into a record having the *null* key. Thus, every record starts and ends with ρ. Each processing element knows the length of the empty space in its shift register.

Instructions and responses are also symbol strings. Instruction strings enter P_1 and response strings emerge from P_p. To simplify things, we will describe the DM operations as if all processors and shift registers were synchronized by means of external clocks. The shift clock period will be denoted by c_s and the processor-to-processor communication clock period will be denoted by c_p. However, the architecture easily supports an

asynchronous mode of operation in the vertical (processor) dimension if the processors are provided with input and output buffers, with a flag indicating the availability of the input buffer for accepting incoming data. Asynchronous operation in the horizontal (shift-register) dimension, while possible, is much more difficult to implement without losing the advantage of high-speed shifting that is possible in synchronous mode.

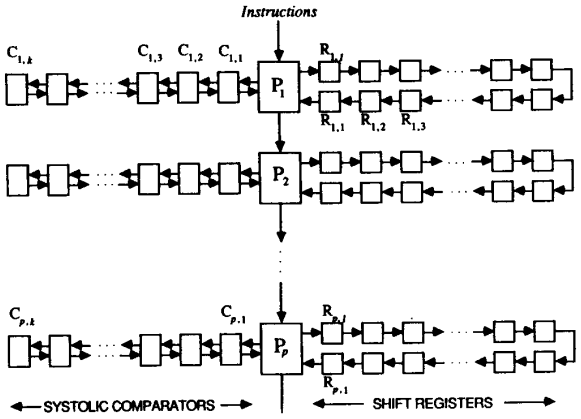

Figure 2. Organization of the Proposed Serial/Parallel Search Processor.

A search instruction is characterized by the following input and output strings:

Input: σ ⟨*reverse-key-string*⟩ ρ

Output: σ' ⟨*info-string*⟩ ρ

The key is input in reverse order to facilitate the matching operation in the systolic comparator (alternatively, one could require that keys be stored in reverse order in the shift registers). A processor receiving the search symbol σ, will pass it and all following symbols up to and including the first ρ to the next processor and shifts the same symbols (σ replaced with ϕ) into its systolic comparator cells (for the key string 'LILIPUT', the comparator cell contents will be 'ϕTUPILILρ'). When a processor in state σ (search instruction input) receives the ρ symbol, it goes into the state σ' (search in progress) and starts pumping the shift register contents into the systolic comparator.

The comparator works as follows. In addition to the permanent copy of the key, a shifted version representing the tail or the unmatched portion of the key is kept. Initially, the tail is the same as the key. The first ρ that emerges from the shift register matches the stored ρ symbol. The key tail is now shifted to the right, with the head symbol ρ disappearing. The shifting can be done without the ripple effect (which would lengthen the required clock cycle) in accordance with the systolic design principles in general and implementation details of systolic stacks in particular (see, e.g., [KUNG82], [GUIB82]).

Intuitively, the following occurs in the systolic stack. The cells go through two phases of operation for each input symbol processed. The first shifted key tail for the above example will be 'ϕTUPILI-L', with a "hole" (denoted by '-') created as a result of shifting the 'L' to the right (refer to Table II). The comparator cell holding 'I' notes the presence of the hole in Phase 2 and shifts its contents to the right, just in time for the next shift into the head cell. This process is repeated, with the next required symbol always being in the correct cell. Because insertions are not required during the use of our systolic stack for matching, its design is simpler than a general systolic stack which requires more "idle" phases between operations. The first non-matching symbol generates a signal which resets the key tail to its initial full-key value. Even though the reset signal ripples from one comparator cell to another, this causes no slowdown as each symbol will have been reset by the time it is needed.

A complete match is detected when both the key tail symbol in $C_{i,1}$ and the next input symbol are ϕ. At this point, the processor passes a σ' symbol to the next processor and then pumps downward the *info-string* symbols as they emerge from the shift register. An incoming σ' cancels the processor's own search and puts it in state σ'' for relaying the incoming information stream downward.

Table I

Analysis of a 12mm Connection in Metal [LEWI87].

Lambda (μm)	RC delay (nsec)	Gate delay (nsec)
5.0	0.1	5.0
2.5	0.2	2.5
0.5	1.0	0.5
0.1	5.0	0.1

Table II

Example for the Operation of a Simple-Match Systolic Comparator.

Stored Key →	ϕ T U P I L I L ρ	↓ Input String ↓
Key Tail (Ph. 1)	ϕ T U P I L I L ρ	↔ ρLILIPUTϕ
Key Tail (Ph. 2)	ϕ T U P I L I L -	
Key Tail (Ph. 1)	ϕ T U P I L I - L	↔ LILIPUTϕ
Key Tail (Ph. 2)	ϕ T U P I L - I -	
Key Tail (Ph. 1)	ϕ T U P I - L - I	↔ ILIPUTϕ
Key Tail (Ph. 2)	ϕ T U P - I - L -	
Key Tail (Ph. 1)	ϕ T U - P - I - L	↔ LIPUTϕ
Key Tail (Ph. 2)	ϕ T - U - P - I L	
Key Tail (Ph. 1)	ϕ - T - U - P - I	↔ IPUTϕ
Key Tail (Ph. 2)	- ϕ - T - U - P -	
Key Tail (Ph. 1)	- - ϕ - T - U - P	↔ PUTϕ
Key Tail (Ph. 2)	- - - ϕ - T - U -	
Key Tail (Ph. 1)	- - - - ϕ - T - U	↔ UTϕ
Key Tail (Ph. 2)	- - - - - ϕ - T -	
Key Tail (Ph. 1)	- - - - - - ϕ - T	↔ Tϕ
Key Tail (Ph. 2)	- - - - - - - ϕ -	
Key Tail (Ph. 1)	- - - - - - - - ϕ	↔ ϕ
Key Tail (Ph. 2)	*Complete Match Detected!*	

The search latency, defined as the period of time between the input of the last key symbol and the output of the first information symbol, is roughly:

$$L = p\,c_p + l\,c_s$$

The pipelining period, defined as the period of time between two successive search requests, is equal to the time needed for one trip around the loop or roughly:

$$P = l\,c_s$$

These equations are approximate in the sense that some constant instruction set-up times have not been included. However, since p and l are relatively large, the inaccuracy is not significant.

An *insert* instruction is characterized by the following input and output strings:

Input: ι ⟨*reverse-key-string*⟩ ϕ ⟨*info-string*⟩ ρ

Output: ι'

The first processor that has sufficient empty space for the record will do the insertion. Upon receipt of the ρ symbol, a processor will know whether it has enough space for the insertion. If so, it simply passes an "insertion-complete" symbol ι' to the next processor instead of the ρ symbol. This symbol is passed downward, invalidates all insertion requests, and eventually emerges as the DM response to the *insert* operation. It is relatively easy to handle duplicate insertions if the design is such that the *insert* instruction also causes any existing copy to be deleted (as discussed for *delete* below).

A *delete* instruction is characterized by the following input and output strings:

Input: δ ⟨*reverse-key-string*⟩ ρ

Output: δ' ⟨*info-string*⟩ ρ

The search part of the *delete* instruction is identical to that of the *search* instruction. Once a match is indicated, the *info-string* is pumped downward following the symbol δ'. This *info-string* is not restored into the shift register. The only remaining problem is how to delete the key which under the search mode of operation is completely restored into the shift register by the time a match is detected. A possible approach would be to provide a *key buffer* in each processor. With this buffer, keys are not automatically restored into the shift register but are pumped into the key buffer which becomes a variable-length extension to the shift register. Upon the detection of a match during a *delete* operation, the key buffer is bypassed and the empty-space length counter is incremented by the key length initially and by 1 with every information symbol pumped out.

The main problem with the above approach is the requirement for a large key register in each processor to accomodate the longest possible key. This not only increases the processor complexity substantially, but also nullifies our architecture's flexibility in dealing with very long and variable-length keys. An alternative solution would be to provide a single-symbol buffer in each processor that essentially delays the storing back of symbols into the shift register by 1 shift cycle. When a match is detected during a "delete" operation, the last key symbol will be replaced by a special "nullifying" symbol. Encountering this special symbol during a search operation invalidates partial match results obtained thus far. Future research must deal with methods for reclaiming the space occupied by such nullified key.

The response latency and pipelining delay for insertion and deletion operations are the same as those for searching which were derived earlier. Thus, the same equations can be used for L and P without regard to the dictionary operation performed. The implications of these equations on the response latency and throughput of the proposed DM are discussed in Section 4.

If all we needed was "exact comparison" of a given key with fixed-length stored keys, the proposed DM would probably not be competitive with conventional (software) approaches in view of the relatively large pipelining period P. However, if more complicated search operations are required, then the architecture presented here becomes quite attractive. For example, if the search must be based on the longest substring match, the systolic comparator might work as follows. Initially, all comparator cells contain zero as the length of the longest match thus far. The longest match information, when updated, ripples rightward, so that ignoring pending propagates, the head comparator cell $C_{i,1}$ always contains information about the longest substring match thus far. Each symbol shifted through the comparator carries with it a count representing the length of the present substring match ending in that symbol. Table III shows an example where the search key is 'LILIPUT' and 'LIP' enters as an input key. The dots in Table III represent either the absense of further information (due to the processor skipping the information part of the record) or the next key that must be compared. Upon completion of one circulation of the shift register and after a short delay for the rightward propagation of the "longest match" information, each processor will know the longest match that it has found. In the next circulation, each processor sends the string

σ' ⟨*match-length*⟩ ⟨*shift-register-contents*⟩

downward. Each processor receiving the above string, compares the received ⟨*match-length*⟩ to its own value and decides on relaying or ignoring the input information. Some post-processing by the host is required to retrieve the desired record from the shift register contents received. In this way, the DM filters out a great deal of irrelevant information.

Systolic comparators can be designed for other types of searches as required. Details for several useful search classes (e.g. the ones needed for executing instructions such as *findmin* and *findmax*) are being worked out. Such systolic comparators are useful components for designing high-speed special-purpose systems and may find applications other than those suggested here in connection with search processors. Eventually, if the proposed architecture proves to be practical, shift-register and systolic comparator libraries may be developed, with individual members of such libraries automatically selected and combined by appropriate VLSI design tools into a final hardware layout.

4. Performance Parameters and Tradeoffs

The analyses in this section are based on the latency and delay equations obtained in Section 3. Given a fixed overall DM capacity N (in terms of the number of R-symbol records), we can write $N = l\,p\,/\,R$ or:

$$l = N\,R\,/\,p$$

Substituting this value for l in the latency and pipelining period equations, we get:

$$L = p\,c_p + N\,R\,c_s\,/\,p$$

$$P = N\,R\,c_s\,/\,p$$

Clearly, the DM throughput (inverse of P) is proportional to the number p of processors. Thus if throughput is of primary importance, the number of processors must be made as large as possible within the given cost constraints. On the other hand, if latency is to be minimized, the number of processors should be:

$$p = \sqrt{N R c_s / c_p}$$

This choice will yield the minimal latency of

$$L_{\min} = 2\sqrt{N R c_s c_p}$$

and a corresponding delay of half this amount. This delay is too large to be of practical interest. Thus the DM is likely to be designed with a higher latency and a correspondingly higher throughput.

Let us compare these performance parameters to those of mesh-connected DMs that have constant pipelining delays of c_m and response latencies proportional to $\sqrt{N}$; say $2c_m\sqrt{N}$. Because each processor in such DMs has to compare a complete key in each clock cycle of length c_m while a processor in our design compares only a single symbol in its clock cycle of length c_p, it is true that:

$$c_s << c_p << c_m$$

Although precise estimates cannot be given at this stage, let us assume somewhat arbitrarily

$$c_m / c_p = c_p / c_s = 20$$

corresponding to clock periods of a few nanoseconds for shifting, a few tens to a hundred nanoseconds for simple interprocessor transfers, and a couple of microseconds for matching a complete key in a more complex processor. With these values, the above latency and pipelining period equations yield the following ratios for the latency and pipelining period of our proposed dictionary machine relative to mesh-connected DMs assuming an average record length of $R = 20$:

$$\text{Latency Ratio} = 0.025\,(p/\sqrt{N} + \sqrt{N}/p)$$

$$\text{Pipelining Period Ratio} = 0.05\,N/p$$

Again arbitrarily choosing $p = \sqrt{N}$, latency and pipelining period ratios become 0.05 and $0.05\sqrt{N}$, respectively. This means that the proposed DM with $\sqrt{N}$ processors is about 20 times faster in terms of response latency. With a database of size $N = 10^6$ records, the delay ratio and thus throughput reduction factor will be 50. This performance is achieved with orders of magnitude less hardware.

Table IV shows the tradeoffs available to the designer by listing the above two ratios for different values of p assuming a database of $N = 10^6$ records. The important point is that even if the values used for c_m/c_p and c_p/c_s are not valid for a particular technology or implementation alternative, the existence of such tradeoffs will not be affected. Only the values and specific crossover points will change. Note that in the extreme case of one record per processor , we obtain a DM based on a linear array of processing elements. In the other extreme of a single processor for all records, we obtain a record-sequential DM which is similar to the word-serial associative processors of many years ago. Clearly, neither of these two extremes is of practical interest for large dictionaries.

The above estimates are quite rough and are therefore subject to refinement. However, consider what would happen even if

Table III

Example for the Operation of the Longest-Substring-Match Systolic Comparator.

Stored Key →	ϕ	T	U	P	I	L	I	L	↓ Input String ↓
									ρ L I P . . .
Current Match *	-	-	-	-	-	-	-	-	
Longest Match *	-	-	-	-	-	-	-	-	
Comparands								ρ	L I P ϕ . . .
Current Match	-	-	-	-	-	-	-	0	
Longest Match	-	-	-	-	-	-	-	0	
Comparands							ρ	L	I P ϕ . . .
Current Match	-	-	-	-	-	-	0	1	
Longest Match	-	-	-	-	-	-	0	1	
Comparands						ρ	L	I	P ϕ . . .
Current Match	-	-	-	-	-	0	1	0	
Longest Match	-	-	-	-	-	0	1	1	
Comparands					ρ	L	I	P	ϕ . . .
Current Match	-	-	-	-	0	1	2	0	
Longest Match	-	-	-	-	0	1	2	1	
Comparands				ρ	L	I	P	ϕ	. . .
Current Match	-	-	-	0	1	0	0	0	
Longest Match	-	-	-	0	1	1	2	2	
Comparands			ρ	L	I	P	ϕ	.	. . .
Current Match	-	-	0	1	2	0	0	0	
Longest Match	-	-	0	1	2	2	2	2	
Comparands		ρ	L	I	P	ϕ	.	.	. . .
Current Match	-	0	1	2	0	0	0	0	
Longest Match	-	0	1	2	2	2	2	2	
Comparands	ρ	L	I	P	ϕ	.	.	.	. . .
Current Match	0	1	2	3	0	0	0	0	
Longest Match	0	1	2	3	2	2	2	2	

* initial zeros are denoted by '-' for clarity.

these estimates were an order of magnitude too optimistic. We would still have a DM that is considerably less expensive than one with millions of processors while having competitive response latency and a throughput that is inferior by two to three orders of magnitude. Thus, a search instruction can be inititated every millisecond or so rather than every few microseconds. Such a system can satisfy the requirements of many applications and provides a valid point in the design space of special purpose search processors.

If the processors of Figure 2 are connected in a two-dimensional mesh or in a binary tree structure instead of a linear array, performance parameters will improve theoretically. However,

such a tree or mesh of shift registers is not directly realizable in today's two-dimensional VLSI technology. Thus to be completely fair in our comparison, we chose to compare our linear array of shift registers, which is essentially a 2-dimensional structure, to 2-dimensional standard DMs. Despite the above observation, such variations on the basic architecture discussed in this paper deserve further study.

Table IV

Latency and Pipelining Period Ratios for a Database of $N = 10^6$ Records.

p	Latency Ratio	Pipelining Ratio
100	0.253	500
200	0.130	250
500	0.063	100
1000	0.050	50
2000	0.063	25
5000	0.130	10
10,000	0.253	5
20,000	0.501	2.5
50,000	1.250	1.0
100,000	2.500	0.5

5. Multi-Level Systolic Search Processors

The serial/parallel design discussed in Sections 3 and 4 can be viewed as a spcial case of a two-level search processor where at the coarse or high level, p processors are used that are arranged as a linear array and operate in parallel while at the fine or low level simple sequential search is utilized. It is natural to generalize this scheme to a multi-level search processor where possibly different architectures and processor interconnection schemes are used at each level. To simplify our discussion, we will only deal with two-level systolic search processors here.

We start by presenting the functional specification of a simple AM (henceforth, SAM). With reference to Figure 1, SAM can execute the following instructions:

SET	Initialize all tag bits to 1.
MATCH	Reset the tag bits of all mismatching cells to 0.
READ	OR the contents of all cells with their tag bits set.
WRITE	Write into all cells that have their tag bits set.
COUNT	Tell Central Control how many responders there are.
SHIFT	Move each tag bit value up or down by one position .

We next discuss how a basic AM might be configured from small SAM-like building blocks (BB-SAMs) according to systolic design principles. Clearly a BB-SAM with multiple cells can be realized as a single VLSI chip. The actual number of cells that can fit on such a chip will depend on implementation details, including internal organization and word length. The internal organization can be fully parallel, bit-serial, or word-serial [PARH73]. With the fully parallel organization, which is the most complex and offers the highest speed, a MOS VLSI chip can contain a few tens of cells. With the bit-serial organization, hundreds of cells per chip may become feasible, but speed is also reduced by a factor related to the average field length in associative operations. The word-serial organization is the least complex and can be implemented by high-speed shift registers, making it possible to fit thousands of cells on a single chip to achieve significant cost benefits.

Let each BB-SAM contain m cells and its instruction cycle time be c. To build an N-word AM from such building blocks, we need $b = N/m$ building blocks or chips. To connect BB-SAMs into a large system, special interface requirements must be met. Figure 3 shows a possible interface specification for a BB-SAM. Any one of the three instruction inputs I, J, or K can carry a command for the BB-SAM of Figure 3 (not all three are needed in some organizations). Similarly, a BB-SAM can pass on an instruction to at most two other BB-SAMs by using the instruction outputs I' and J'. The values on count inputs and output of a BB-SAM are related by:

C' = C + D + Internal Count

For the purpose of executing the SHIFT instruction, each cell within a BB-SAM is assigned a sequence number. Let h and k be the smallest and largest sequence numbers within the BB-SAM under consideration. It has been established [PARH90], [PARH90b] that for tree-structured designs, the cells cannot be consecutively numbered from h to k and that two extra pairs of shift input and output lines (say A1, A1', B1, and B1') will be needed to properly handle the SHIFT instruction. For now, let us assume that cells are consecutively numbered $h, h + 1, \ldots, k$. Then the shift inputs A and B will carry the tag bit values from cells $h - 1$ and $k + 1$, respectively. Similarly, the shift outputs A' and B' will carry the tag bit values from cells h and k and are connected to the shift inputs of BB-SAMs containing the cells $h - 1$ and $k + 1$, respectively.

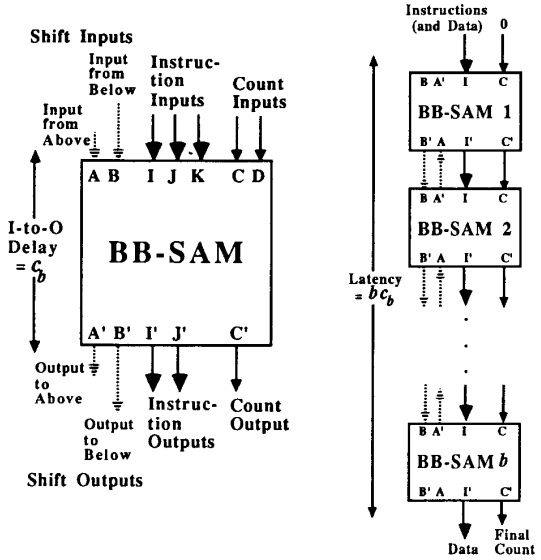

Figure 3. The I/O Interfaces of a Building-Block SAM (BB-SAM) and a Linear Array of Such Building Blocks.

The simplest possible interconnection scheme for BB-SAMs is a one-dimensional or linear array (Figure 3). Instructions are fed to the topmost BB-SAM. Upon receiving an instruction, a BB-SAM executes it and forwards the result (if any) along with the

instruction itself to the next BB-SAM in the array. Thus, the result of an instruction input at time t will emerge from the last BB-SAM at time $t + bc$, where c is the cycle time of a BB-SAM. However, the next instruction can be input to the topmost BB-SAM one cycle later (at time $t + c$). Instructions follow one another, causing full utilization of the BB-SAMs in the ideal case of an infinite sequence of independent instructions.

Higher-dimensional arrays are clearly more desirable because of their smaller graph-theoretic diameters. In the case of a two-dimensional mesh, the result of an instruction input at time t to the leftmost BB-SAM in the top row will emerge from the rightmost BB-SAM in the bottom row at time $t + (2\sqrt{b} - 1)c_M$, where c_M is the cycle time of a BB-SAM for mesh interconnection. However, the next instruction can be input to the topmost BB-SAM one cycle later (at time $t + c_M$). The cycle time c_M for a mesh BB-SAM is likely to be larger than the cycle time c for a linear-array BB-SAM due to the slightly greater complexity. For a binary-tree organization of BB-SAMs, the result of an instruction input at time t to the root cell will emerge from the root BB-SAM at time $t + (2a - 1)c_T$, where c_T is the cycle time of a BB-SAM for tree interconnection. Again, the next instruction can be input to the topmost BB-SAM one cycle later (at time $t + c_T$). The cycle time c_T for a tree BB-SAM is larger than the cycle times c and c_M for linear-array and mesh BB-SAMs, both because the former is more complex internally and because it is connected to its neighbors by longer wires. Details of these organizations are omitted here for brevity.

As mentioned earlier, the number m of cells that can fit on a chip will depend on the internal organization of the BB-SAMs. The slower word-serial organization accomodates more words per chip, thus reducing the number b of chips, and thus the cost, at the expense of a larger pipelining period c_b (lower throughput). A fast fully parallel organization will maximize the throughput but will also increase the cost and possibly the response latency (which may become quite unacceptable in the case of a linear array). A wide range of tradeoffs are thus available.

To illustrate these tradeoffs in more concrete terms, let us examine in some detail the design of a BB-SAM for the linear array interconnection scheme. We consider 3 implementation alternatives of fully parallel, bit-serial, and word-serial, with the number of cells that would fit on a chip being m, m', and m'', respectively. In general, the parameters for the fully parallel, bit-serial, and word-serial organizations will be denoted by simple, primed, and double-primed variables. We are interested in two performance indicators: the response latency $L = bc$ and the pipelining period $P = c$. The hardware cost will be roughly proportional to b, the number of chips. We have $b = N/m$, $b' = N/m'$, and $b'' = N/m''$. Thus, to compare the performance of the three organizations being considered, estimates for the relative magnitudes of c, c', c'' and m, m', m'' are needed.

To obtain the required estimates for c, c', and c'', let us look at the operation of each scheme. In the fully parallel organization, m comparators are built into each chip. Because of the relatively long words of typical AMs (say 256 bits or more), rippling time for the match signals in a comparison cycle, and thus c, may well exceed 1 microsecond (256 * 2 logic levels * 2 nsec, say). In the bit-serial organization, data is stored in a conventional memory module with bit-slice access. The time needed for MATCH, READ, and WRITE instructions is proportional to the average length k of the search or data field. Thus, we have $c' = kc_m$, where c_m is the on-chip memory cycle time (typically less than 100 nsec). Processing time can be overlapped with memory access and thus need not be considered separately. Clearly, $c_m < c$, with the ratio $r'_c = c/c_m$ potentially exceeding 10. In the word-serial organization, records are shifted sequentially into the processing section. If the processing function is pipelined, the data rate is dictated by the much smaller shift cycle c_s rather than the full comparison cycle c. Thus, $c'' = m''c_s$. With proper design, c_s should be much smaller than c_m (as low as 10 nsec or less) and like c/c_m, the ratio $r''_c = c_m/c_s$ can easily approach or exceed 10.

The relative magnitudes of m, m', and m'' cannot be as easily deduced without the benefit of actual hardware realizations. Assume $m'/m = r'_m$ and $m''/m = r''_m$. Combining the above relationships and estimates, we can write $P = c$, $L = Nc/m$, and:

$$P' = c' = kc_m = ck(c_m/c) = Pk/r'_c$$

$$P'' = c'' = m''c_s = cm''(c_m/c)(c_s/c_m)c_b = Pm''/(r'_c r''_c)$$

$$L' = Nc'/m' = Nkc_m/m' = Lk(c_m/c)(m/m') = Lk/(r'_c r'_m)$$

$$L'' = Nc''/m'' = Nc_s = Lm(c_m/c)(c_s/c_m) = Lm/(r'_c r''_c)$$

Because k and m are almost certainly less than $r'_c r'_m \approx 100$ and $r'_c r''_c \approx 100$, respectively, the bit-serial and word-serial organizations perform better with respect to the latency parameter L. On the other hand, since k is likely to be greater than $r'_c \approx 10$ and m'' greater than $r'_c r''_c \approx 100$, the fully parallel organization has the edge in terms of pipelining period P or throughput. To see how sensitive the above analyses are to the rough estimates $r'_c \approx r''_c \approx r'_m \approx 10$, we proceed as follows. Taking our previous discussions and estimates into account, it is not unreasonable to assume for the sake of simplicity (reducing the number of parameters to be dealt with) that:

$$r'_c = r''_c = r'_m = r$$

Figure 4 shows the variation of the ratios P'/P and P''/P for a wide range of values for the parameter r and for a few representative values of the parameters k and m''. Similarly, Figure 5 depicts the variations of the ratios L'/L and L''/L for several reasonable values for the parameters k and m. Figures 4 and 5 confirm our previous conclusions in a broader context.

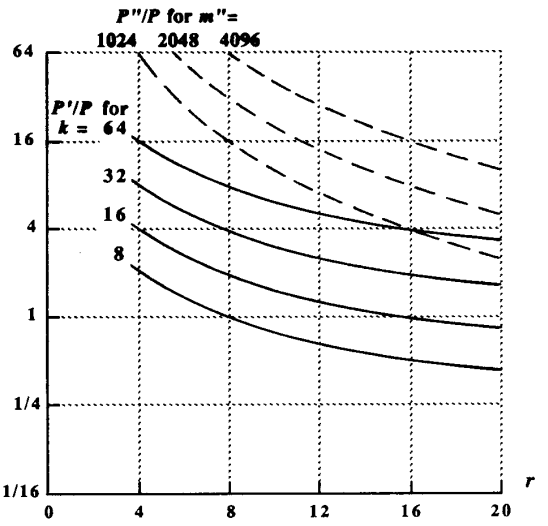

Figure 4. Ratios of Pipelining Periods for the Three AM Organizations.

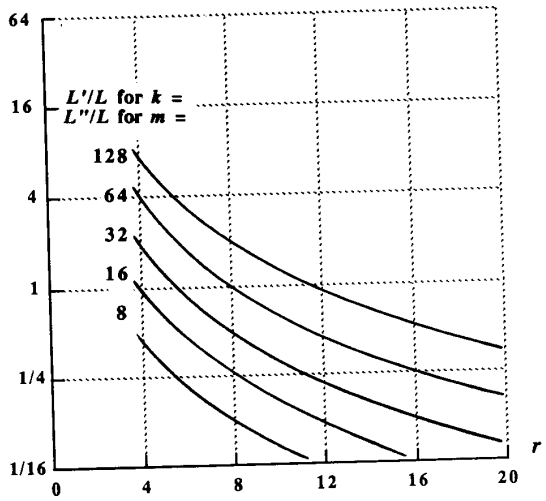

Figure 5. Ratios of Response Latencies for the Three AM Organizations.

6. Conclusion

We have suggested design paradigms for special-purpose search processors that provide a wide range of speed/cost tradeoffs to designers. A key feature of our serial/parallel approach is the replacement of many processors in previously proposed systems by a much denser high-speed shift register connected to a single processor. This results in considerable reduction in size and cost and at the same time provides the flexibility of dealing with variable-length and very long keys at acceptable speeds. Such machines can be constructed with today's VLSI technology for databases containing a million or so records (a few thousands of processors). Our analysis demonstrates that hardware-intensive designs do not necessarily offer significant performance gain over architectures that match the requirements of the application but utilize fewer processors.

We have also proposed systolic architectures for associative memories, resulting in systems whose performance parameters are realistically independent of size for long sequences of operations with proper optimization of instruction sequencing. The proposed architectures should lead to practical VLSI realizations of large associative memories which would be impossible to implement under the "operand-broadcasting" and "reduction-by-wired-logic" paradigms.

The proposed serial/parallel approach is applicable to many other searching problems, provided that the required matching can be performed in a character-serial fashion by a systolic string comparator. Systolic comparators can be designed for other types of searches as required. Such systolic comparators are useful components for designing high-speed special-purpose systems and may find applications other than the one suggested here in connection with DMs. Eventually, if the proposed architecture is judged to be practical, shift-register and systolic comparator libraries may be developed, with individual members of such libraries automatically selected and combined by appropriate VLSI design tools into a final hardware layout. Further research is clearly needed to refine our approximate analyses and to translate the concepts presented into actual hardware and software design guidelines.

7. References

[ATAL85] Atallah, M.J. and S.R. Kosaraju, "A Generalized Dictionary Machine for VLSI," *IEEE Transactions on Computers*, Vol. C-34, No. 2, pp. 151-155, Feb. 1985.

[BENT79] Bentley, J. and H. Kung, "A Tree Machine for Searching Problems," *Proc. of the Int'l Conf. on Parallel Processing*, 1979, pp. 265-266.

[BERT90] Bertossi, A.A., "A VLSI System for String Matching," *Integration*, Vol. 9, pp. 129-139, 1990.

[BROW79] Browning, S.A., "Computations on a Tree of Processors," *Proc. of the Caltech Conf. on VLSI*, Jan. 1979, pp. 453-478.

[CARE84] Carey, M.J. and C.D. Thompson, "An Efficient Implementation of Search Trees on $\lceil lg\ N + 1 \rceil$ Processors," *IEEE Transactions on Computers*, Vol. C-33, No. 11, pp. 1038-1041, Nov. 1984.

[CHAN88] Chang, J.H., O.H. Ibarra, M.J. Chung, and K.K. Rao, "Systolic Tree Implementation of Data Structures," *IEEE Transactions on Computers*, Vol. 37, No. 6, pp. 727-735, June 1988.

[DAVI86] Davis, W.A. and D.-L. Lee, "Fast Search Algortihms for Associative Memories," *IEEE Transactions on Computers*, Vol. C-35, No. 5, pp. 456-461, May 1986.

[DEHN87] Dehne, F. and N. Santoro, "Optimal VLSI Dictionary Machines on Meshes," *Proc. of the Int'l Conf. on Parallel Processing*, University Park, PA, Aug. 1987, pp. 832-840.

[DEHN88] Dehne, F. and N. Santoro, "An Optimal VLSI Dictionary Machine for Hypercube Architectures," School of Computer Science, Carleton Univ., Ottawa, Canada, Report SCS-TR-135, Apr. 1988.

[FISH84] Fisher, A.L., "Dictionary Machines with a Small Number of Processors," *Proc. of the Int'l Symp. on Computer Architecture*, pp. 151-156, 1984.

[GOYA88] Goyal, P. and T.S. Narayanan, "Dictionary Machine with Improved Performance," *The Computer Journal*, Vol. 31, No. 6, pp. 490-495, Dec. 1988.

[GUIB82] Guibas, L.J. and F.M. Liang, "Systolic Stacks, Queues, and Counters," *Proc. of the MIT Conf. on Advanced Research in VLSI*, 1982, pp. 155-164.

[HILL85] Hillis, W.D., *The Connection Machine*, The MIT Press, 1985.

[HURS87] Hurson, A.R. and B. Shirazi, "Associative Memories: Has Their Time Come? -- Applications and VLSI Complexity," *Proc. of the Hawaii Int'l Conf. on System Sciences*, 1987, pp. 284-292.

[KNUT73] Knuth, D.E., *The Art of Computer Programming: Vol. 3 — Sorting and Searching*, Addison-Wesley, 1973.

[KUNG82] Kung, H.T., "Why Systolic Architectures?" *Computer*, Vol. 15, pp. 37-46, Jan. 1982.

[LEED88] Lee, D.-L. and W.A. Davis, "An $O(n + k)$ Algorithm for Ordered Retrieval from an Associative Memory," *IEEE Transactions on Computers*, Vol. 37, No. 3, pp. 368-371, Mar. 1988.

[LEWI87] Lewis, E., "The Design and Performance of 1.25μ CMOS," *VLSI System Design*, Mar. 1987.

[MEHL84] Mehlhorn, K., *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, 1984.

[OMON87] Omondi, A.R. and J.D. Brock, "Implementing a Dictionary on Hypercube Machines," *Proc. of the Int'l Conf. on Parallel Processing*, University Park, PA, Aug. 1987, pp. 707-709.

[OTTM82] Ottman, T.A., A.L. Rosenberg, and L.J. Stockmeyer, "A Dictionary Machine for VLSI," *IEEE Transactions on Computers*, Vol. C-31, pp. 892-897, Sep. 1982.

[PARH72] Parhami, B., "A Highly Parallel Computing System for Information Retrieval," *AFIPS Conf. Proc.*, Vol. 41 (Fall Joint Computer Conf.), AFIPS Press, Montvale, NJ, 1972, pp. 681-690.

[PARH73] Parhami, B., "Associative Memories and Processors: An Overview and Selected Bibliography," *Proceedings of the IEEE*, Vol. 61, No. 6, pp. 722-730, June 1973.

[PARH90] Parhami, B., "Associative Memory Designs for VLSI Implementation," *Proc. of the Int'l Conf. on Databases, Parallel Architectures, and Their Applications*, Miami, Mar. 1990, pp. 359-366.

[PARH90a] Parhami, B., "Massively Parallel Search Processors: History and Modern Trends," *Proc. of the Fourth Annual Symp. on Parallel Processing*, Fullerton, CA, Apr. 1990, pp. 733-747.

[PARH90b] Parhami, B., "Systolic Associative Memories," *Proc. of the Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 1990, pp. I-545 to I-548.

[PATE81] Paterson, M.S., W.L. Ruzzo, and L. Snyder, "Bounds on Minimax Edge Length for Complete Binary Trees," *Proc. of the ACM Symp. on the Theory of Computing*, May 1981, pp. 293-299.

[PROB87] Probst, D.K. and H.F. Li, "Optimal Systolic Dictionary Machines on Mesh-Type Architectures," *Proc. of the Canadian Conf. on VLSI*, Winnipeg, Oct. 1987, pp. 247-252.

[SCHM85] Schmeck, H. and H. Schroder, "Dictionary Machines for Different Models of VLSI," *IEEE Transactions on Computers*, Vol. C-34, pp. 472-475, 1985.

[SCHW87] Schwartz, A.M. and M.C. Loui, "Dictionary Machines on Cube-Class Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 1, pp. 100-105, Jan. 1987.

[SIVI85] Sivilotti, M., M. Emerling, and C. Mead, "A Novel Associative Memory Implemented Using Collective Computation," *Proc of the Chapel Hill Conf. on Very Large Scale Integration*, Computer Science Press, 1985, pp. 329-342.

[SOMA85] Somani, A.K. and V.K. Agarwal, "An Efficient Unsorted VLSI Dictionary Machine," *IEEE Transactions on Computers*, Vol. C-34, No. 9, pp. 841-852, Sep. 1985.

[SONG80] Song, S.W., "A Highly Concurrent Tree Machine for Database Applications," *Proc. of the Int'l Conf. on Parallel Processing*, Aug. 1980, pp. 259-268.

[WEEM82] Weems, C. et al., "TITANIC: A VLSI Based Content Addressable Parallel Array Processor," *Proc. Int'l Conf. on Computer Communications*, 1982, pp. 236-239.