Exploiting Parallelism in Pattern Matching: An Information Retrieval Application

VICTOR WING-KIT MAK, KUO CHU LEE, and OPHIR FRIEDER
Bellcore

We propose a document-searching architecture baaed on high-speed hardware pattern matching to increase the throughput of an information retrieval system. We also propose a new parallel VLSI pattern-matching algorithm called the Data Parallel Pattern Matching (DPPM) algorithm, which serially broadcasts and compares the pattern to a block of data in parallel. The DPPM algorithm utilizes the high degree of integration of VLSI technology to attain very high-speed processing through parallelism. Performance of the DPPM has been evaluated both analytically and hy simulation. Based on the simulation statistics and timing analysis on the hardware design, a search rate of multiple gigabytes per second is achievable using 2-μm CMOS technology. The potential performance of the proposed document-searching architecture is also analyzed using the simulation statistics of the DPPM algorithm.

Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structured: Design Styles—*parallel*; B.7.1 [Integrated Circuits]: Types and Design Styles-algorithms *implemented in hardware*, *VLSI*: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—*SIMD*; C.4 [Computer Systems Organization]: Performance of Systems—*design studies*, *modeling techniques*; E.5 [Data]: Files—*sorting/searching*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*pattern matching*, *sorting and searching*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*search process*, *selection process*

General Terms: Algorithms. Design, Performance

Additional Key Words and Phrases: DPPM, pattern matcher

1. INTRODUCTION

Information retrieval is the recovery of documents that match a user's query, which consists of a set of search patterns combined via a set of operators. The relevance of a document can be determined by the occurrences of the set of search patterns defined in the query. With the continued growth of unformatted, textual databases,[1] it is important to reduce the amount of document search time to support adequate response times. A common approach to

[1] The legal database Lexis is estimated at over 125 GBytes of information [23]. It is reported that information retrieval databases have been growing at a rate of 250.000 documents per year [9].

Authors' addresses: V. W. Mak and K. C. Lee. Bellcore, 445 South Street, Morristown, NJ 07960-1910; 0. Frieder, Department of Computer Science, George Mason University, Fairfax, VA 22030.

document search, aimed at reducing the search time, is software indexing. However, if word-level indexing is used, the storage overhead associated with indexing may be as much as 300 percent [7]. Changes in the database also require substantial overhead in maintaining the indices.

Another approach used in reducing the retrieval time of relevant documents is the utilization of high-speed hardware filters to perform the pattern-matching operations. This significantly reduces the need to maintain the indices and simplifies updates to the database. Hardware filters can support operations, such as arbitrary wild cards and searches for embedded strings, that are infeasible with indices. Hardware filters are also suitable to search and retrieve relevant documents on-the-fly from ever-changing information sources transmitted through high-bandwidth optical fibers, such as news articles and stock quotes. Because of the rapidly decreasing cost of VLSI technology, compared to the design and maintenance of software systems and the increased availability of CAD tools, the exploitation of customized hardware for information retrieval merits investigation.

In this paper we propose a document-searching architecture, based on high-speed hardware pattern matchers, to increase the throughput of an information retrieval system. The proposed architecture is comprised of a set of customized document search engines (Data Parallel Pattern Matching Engines, or DPPMEs) and a single master Processing Element (PE). An information retrieval query is decomposed by the PE into basic match primitives to be executed in the DPPMEs, while the query (a sequence of operators) itself is evaluated at the PE using results from the document search engines. The PE is similar to the query resolver as proposed in [10]. By separating the operator and query complexity from the DPPMEs, the complexity of the customized hardware is made independent of the complexity of the query. This separation results in a simpler and hence more efficient hardware implementation of the DPPMEs.

The PE instruction set is based on the text-retrieval machine instruction set presented in [10], but modified to be match-bused; that is, each instruction is defined as a set of match conditions with a simple imposed control structure. As the imposed control structure requires minimal computation time relative to the amount of processing involved in searching the document stream, we believe it can be implemented in software without significantly affecting the system throughput.

Supporting a modular design with a simple interface enables component substitution. Thus, an implementation based on such a design is easily modified to incorporate algorithmic improvements and technological advances. The match-based interface proposed here adheres to the modular-design principle.

Hardware pattern matchers have been previously proposed and implemented [3, 4, 6, 8, 15, 17, 18, 21, 24] to speed up the time-consuming task of document searching. Currently proposed searching rates have hovered at roughly 20 MBytes per second [21]. This rate may be sufficient to match the I/O bandwidth of current disk technology (about 10 MBytes per second), but is certainly inadequate for future optical disk transfer rate (estimated at approximately 200 MBytes per second [1]) and semiconductor main memory bandwidth of supercomputers (as high as 1 GBytes per second). In this paper we propose a new parallel VLSI algorithm called Data Parallel Pattern Matching (DPPM) and a corresponding VLSI document search engine called DPPME. The DPPM algorithm differs from most previous work in that it serially broadcasts each character in the pattern and compares the pattern to a block of data in parallel. The DPPM algorithm utilizes the high degree of integration of VLSI technology to attain very high-speed processing through parallelism. Based on simulation statistics and timing analyses of the hardware design, a search rate of multiple gigabytes per second per DPPME is achievable using Z-pm CMOS technology.

The remainder of the paper is organized as follows. Section 2 provides an overview of the proposed document-searching architecture. Section 3 first reviews previous work in hardware document search engines, followed by the description of the DPPM algorithm and a VLSI design of the DPPME. Technology issues in implementing data broadcasting in VLSI are also addressed in this section. Section 4 presents the performance evaluation of both the DPPM algorithm and the proposed document-searching architecture using DPPMEs. The DPPM algorithm has been evaluated both analytically and by simulation on a text database. Using the simulation statistics of the DPPM algorithm, the potential performance of the proposed document-searching architecture is also analyzed. Finally, a summary is given at the end of this paper.

2. THE PROPOSED DOCUMENT-SEARCHING ARCHITECTURE

The structure of the proposed document-searching architecture is comprised of a single *master* Processing Element (PE) controlling a set of *slave* Data Parallel Pattern Matching Engines (DPPMEs) (see Figure 1). A complex query is decomposed by the PE into basic match primitives to be executed at the DPPMEs. Each DPPME compares the document data stream against its own assigned pattern and forwards the comparison results back to the PE to be processed. Previously proposed text filters [3, 8, 24] evaluate an entire query via integrated custom hardware. However, the ability to evaluate complex queries requires complicated circuitry to support the state transition logic and partial results communication for cascaded predicate evaluation. Since the predicate evaluation and query resolution are decoupled from the primitive pattern-matching operations, the complexity of an individual query is retained at the PE level, and hence only simple comparator circuitry is required for the DPPME to execute the pattern-matching operations.

As seen in Figure 1, multiple DPPMEs can read data from a single source through a common broadcast bus. Each DPPME reads the data and compares the assigned pattern to the input stream. If a match is detected, an interrupt is issued to the PE. As will be explained in Section 3, each DPPME may require different amounts of processing time for a block of data stream. Input buffers are used in each DPPME to buffer up incoming data blocks so that all DPPMEs can execute at approximately the same search rate.

The match-based PE instruction set and the corresponding match sequence that implements each of the individual instructions are shown in Figure 2. The instruction set in the described approach is based on the text-retrieval

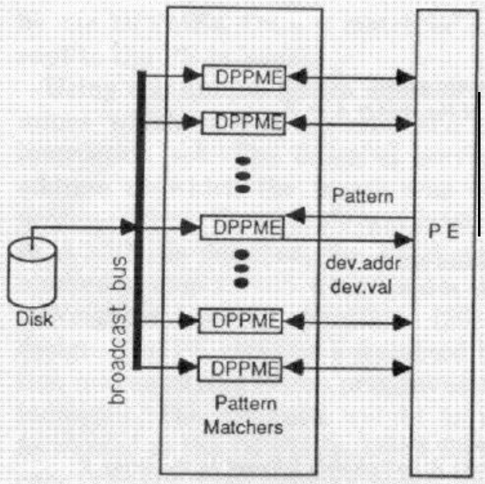

Fig. 1. Proposed document-searching architecture.

machine instruction set presented in [10], with a corresponding proposed implementation utilizing DPPMEs. The occurrence of multiple overlapped copies of a search pattern in a given input string is permitted and is supported by the instruction set. An instruction set prohibiting search-pattern overlap is described in [5]. In the instruction set presented here, the leftmost column presents the actual instruction. A semantic description of the instruction is provided in italics followed by the control structure implementing the instruction.

For example, the dev := match(X) primitive is defined as follows. The match primitive assigns dev the PE memory location where the DPPME scanning for X stores its results. The match primitive blocks until either an END_OF_DOC indication is detected or a match is found. The following information is returned.

END-OF-DOG detected:	dev.val = FALSE
	dev.addr = DEFAULT
	where DEFAULT = −(MAX_DOC_LENGTH + 1).
Match detected:	dev.val = TRUE
	dev.addr = address of the last character of X.

From the above definitions, whenever a new document is scanned, dev.val = FALSE anddev.addr = DEFAULT.

In the instruction set, each *numbered action* is an atomic operation. All subactions within an action, each separated by a semicolon, are enclosed within a *cobegin/coend* pair and are performed concurrently, Actions are separated by a period and executed serially. An action does not terminate until all the subactions comprising an individual step terminate. Once a pattern is assigned to a particular DPPME, the same DPPME is used throughout the entire instruction to search for the given pattern. Thus, throughout the execution of an instruction, invoking the same dev := match(X) primitive repetitively, always results in the same PE memory location address being assigned to dev. Finally, the entire instruction

```
A          Find any document containing the string A
           1.   dev_A := match( A ).
           2.   if (dev_A.val)
                     then return TRUE else return FALSE.
A OR B     Find any document containing either of the sirings A on B
           1.   cobegin
                     dev_A := match( A );
                     dev_B := match( B )
                coend.
           2.   if (dev_A.val or dev_B.val)
                     then return TRUE else return FALSE.
A AND B    Find any document containing both the strings A and B
           1.   cobegin
                     dev_A := match( A );
                     dev_B := match( B )
                coend.
           2.   if ( dev_A.val and dev_B.val )
                     then return TRUE else return FALSE.
AB         Find any document containing the string A immediately followed by the string B
           1.   C := A B. /* concatenate A and B */
           2.   dev_C := match(C).
           3.   if (dev_C.val)
                     then return TRUE else return FALSE.
A?B        Find any document containing string A followed by any character followed by airing B
           1.   C := A#B. /* # is the don't care character */
           2.   dev_C := match(C).
           3.   if (dev_C.val)
                     then return TRUE else return FALSE.
A.. .B     Find any document containing the string A followed either immediately on after an arbitrary
           number of characters by string B
           1.   dev_A := match(A).
           2.   if (dev_A.val) then dev_B := match(B).
           3.   if (dev_B.val)
                     then return TRUE else return FALSE.
A.n.B      Find any document containing the string A followed by string B within n characters
           1.   last_A := DEFAULT.
           2.   cobegin
                     while (not dev_A.val) do
                     {    dev_A := match(A).
                          if (dev_A.val) then last_A := dev_A.addr.
                     };
                     while (not dev_B.val) do
                     {    dev_B := match(B).
                          if ((dev_B.addr - last_A) ≤ (n + length(B))
                               then return TRUE.
                     }
                coend.
           3.   return FALSE.
```

Fig. 2. Master PE instruction set.

processing terminates for a given document once a return statement is issued. Document addresses corresponding to TRUE results are recorded and retrieved.

To evaluate an instruction, the PE allocates a DPPME per match comparison required by the instruction for the entire duration of the instruction. For example, processing A OR B requires 2 DPPMEs. The documents are scanned by the DPPMEs. Once a match or an END_OF_DOC is detected, an interrupt to the PE is issued.

Using the unique device address of the signalling DPPME, the PE determines which match occurred and processes the match as shown in the instruction set. The order of servicing interrupts is based on the block address at which the match was detected; lowest address interrupts are processed first. If a nonrelevant interrupt, such as match(B) prior to match(A) in A.. . B, is detected, the PE disregards the interrupt, as it is currently blocked at step 1.

Weighted boolean operations [20] are easily incorporated in the above design. The PE receives a weighted boolean condition and parses the operation into substring match primitives; the associated weight of the operation is indicated. The assigned weight is recorded, and the match patterns are forwarded to the DPPMEs. Upon completion of the match operations, the PE computes the final boolean result and associates the result with the stored weight. We now focus on the DPPM algorithm and the corresponding DPPME processor.

3. DATA PARALLEL PATTERN-MATCHING ENGINE

3.1 Previous Work in Hardware Pattern Matching

The pattern-matching problem is to find all occurrences of a p-character pattern, P, constructed from a vocabulary of m distinct characters, in an s-character data string, S. The pattern P may also contain *don't care* characters. For typical applications, $p \ll s$ and $m \ll s$. Since the size of the data string is usually very large, sequential search via general-purpose processors is prohibitively slow.

To expedite the search, numerous hardware-based solutions to pattern matching have been investigated, and some are actually implemented [3, 4, 6, 8, 15, 17, 18, 21, 24]. As fast software pattern-matching algorithms [2, 12] are based on finite state automata (FSA), hardware realizations of FSA pattern matching were investigated by [8]. FSA requires precompilation of the patterns and processes the data string one character at a time. Although precompilation of the pattern eliminates the need to compare each character of the data string to every pattern character, the sequential character-at-a-time processing severely limits the search rates of these systems.

Unlike the software and hardware FSA approaches, many hardware approaches [3, 4, 6, 24] use comparator arrays to perform pipelined pattern matching directly without precompilation of the patterns. Multiple patterns are compared concurrently to the data string to achieve higher throughput. However, the search rate is still limited by the sequential processing of the data string. Furthermore, in comparison with the software and hardware FSA implementations, most of the comparator array approaches exhibit. a significant amount of redundant comparisons. Redundancy can be classified into physical redundancy and logical redundancy. Physical redundancy indicates low utilization of hardware, while logical redundancy indicates the number of comparisons that occur after a mismatch is identified.

In the systolic array approach [4], data and pattern characters are routed in opposite directions. At any given clock cycle, only half of the cells in the array can perform meaningful computation; therefore, half of the physical hardware is actually wasted.

The cellular array approaches proposed by [3] and [24] avoid the problem of physical redundancy; however, logical redundancy still exists. In these two approaches, pattern characters are preloaded into the comparator cells. Each character of the data string is broadcast into all cells serially. and comparison results are generated by all cells simultaneously. Through the control of finite state automata, a match signal is propagated through an array of control cells to the outside world. However, most of the comparisons between the data string and the last few pattern characters are redundant because a mismatch may have occurred among the first few pattern characters. Since pattern-matching operations on text databases exhibit low selectivity, comparisons beyond the first few characters of the pattern are usually redundant.

To reduce the number of redundant. comparisons and to increase the degree of effective parallelism in the pattern-matching problem, both ALTEP 1151 and our DPPME utilize a data parallel, pattern serial scheme in which pattern characters are broadcast and compared to a block of the data string in parallel. While ALTEP is a cellular processor optimized for regular expression comparisons with microprogrammed control, DPPME is a VLSI filter optimized for variable-length text processing with hardwired control. The decoupling of query resolution from the primitive match operation simplifies the structure of the DPPME so that it. can he implemented compactly, and hence is more efficient. A major issue in supporting pattern matching for variable-length text using the data parallel, pattern serial scheme is in the handling of multiple partial matches crossing block boundaries. The DPPME has architectural support that provides an integrated and efficient mismatch detection and partial-match propagation mechanism. This is more efficient than using microprogrammed control to store the partial-match positions into multiple registers, to scan for mismatch conditions, and to issue anchor(O) instructions to the cellular array in multiple steps of microinstructions.

3.2 The DPPM Algorithm

As in previous approaches [3, 6, 241, the DPPM algorithm also uses a comparator array to parallelize search operations. However, instead of serially broadcasting the data string characters to a comparator array containing the pattern, the DPPM algorithm serially broadcasts the *pattern* characters to a comparator array containing a block of the data string.

Let $S[1:n]$ be the data string of n characters to he searched and $Pat[1:p]$ be the pattern of p characters. The data string is divided into blocks of b characters each and searched a block at a time. Let $Blk[1:b]$ be the current data block of size b characters. Basically, the DPPM algorithm serially broadcasts each pattern character to a block of the data string. If the pattern character matches any of the characters in the block, the next pattern character is broadcast in the next comparison cycle. If, at any cycle, no match

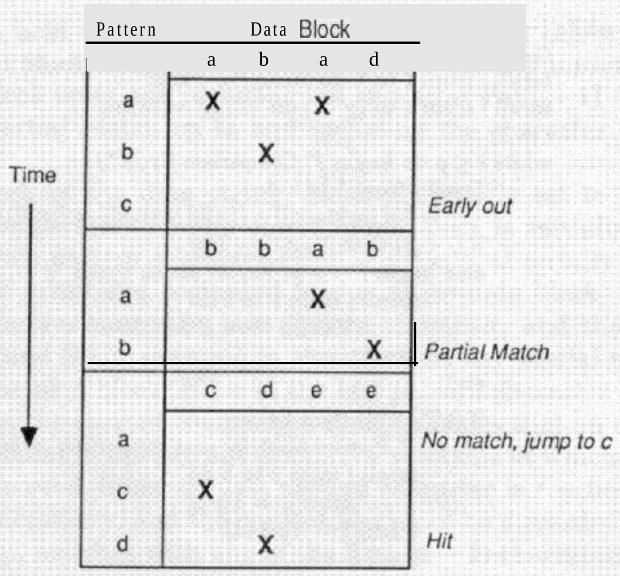

Fig. 3. DPPM example.

is found between the current pattern character and the data block, and if no partial match is carried over from the previous block, the data block is discarded, and the search continues with the next block. A partial match occurs when Pat[i], i < p, matches the last data block character Blk[b]. This partial-match information is stored and used in the next block to continue the search by comparing Pat[i + 1] to the first data block character Blk[1].

The DPPM algorithm can be best illustrated using a simple example. Suppose a search for the pattern *abcd* in the data string *abadbbabcdee* is conducted. Figure 3 shows the operation of the DPPM algorithm using a block size of 4. The first block, containing the characters *abad*, is first loaded into the comparator array. When compared to the first pattern character *a*, two matches are detected. The second pattern character *b* is then broadcast and compared to the characters to the right of the matched characters in the previous cycle, i.e., *b* is compared to the second and the fourth characters in the block. Since a match is detected again at the second character, a comparison of the third pattern character c with the third character in the block is necessary. This time no match in the block is observed, so the current block is discarded; and the search continues with the next block. This early mismatch detection mechanism avoids broadcasting and comparing the fourth pattern character *d* to the current block since this comparison is redundant.

The next block contains the characters *bbab*. The pattern compares successfully up to the second character *b*. At this point, the end of the block is reached. There is a possibility that the pattern may span the block boundary. DPPM remembers this partial-match information and continues the match in the next cycle. Since there is no other match in the block, the current block is also discarded.

```
while (¬ End of Data String) do begin
    GetNextBlock(Blk);
    forall i from 1 to b do Mask[i] := TRUE;
    forall i from 1 to (p - 1) do Vout[i] := FALSE;
    i := 1;
    while i ≤ p do begin /* Comparison Cycle */
        forall j from 1 to b do
            T[j] := Mask[j] ∧ (DC[i] ∨ (Blk[j] = Pat[i]));
        if (i < p) then Vout[i] := T[b]
        else begin /* i = p: Report Match Found */
            forall j from 1 to b do
                if (T[j]) then Report Match at j;
            break;
        end;
        if ($\bigvee_{j=1}^{b-1}$ T[j]) then begin
            /* Prepare Mask for next comparison cycle */
            forall j from 2 to b do
                Mask[j] := T[j-1];
            Mask[1] := Vin[i+1];
            i := i + 1;
            end
        else if ($\bigvee_{j=i+1}^{p}$ Vin[j]) then begin
            /* Skip Unnecessary Pattern Characters */
            do i := i + 1 until (Vin[i]);
            Mask[1] := TRUE;
            forall j from 2 to b do Mask[j] := FALSE;
            end
        else /* Early Out */
            break;
    end;
    forall i from 2 to p do Vin[i] := Vout[i-1];
end;
```

Fig. 4. Pseudocode of the DPPM algorithm.

The third block contains the characters *cdee*. The first pattern character a has no match with the block. At this point, DPPM recalls there was a partial match in the previous block up to the second pattern character; therefore, it jumps to the third pattern character *c* and continues the partial match from the previous block. Finally, a hit or an occurrence of the pattern is detected with the fourth pattern character *d*.

Although not shown in this example, the DPPM algorithm can also detect multiple occurrences of the pattern even if they overlap, and no backtracking is required to detect all occurrences.

Pseudocode of the control flow of the DPPM algorithm is shown in Figure 4. The description below follows closely with the pseudocode. DC[1:p] is a bit-vector indicating the don't *care* positions in the pattern. DC[i] is set if there is a *don't care* at Pat[i]. Mask[1:b] controls the activation of the comparator array based on the results of the previous comparison cycle. If Pat[1] matches Blk[i] in the first comparison cycle, then Mask[i + 1] is set in the next. comparison cycle, enabling the comparison between Pat[2] and Blk[i + 1]. T[1:b] holds the results of the comparator array. Vin[2:p] and

Vout[1:(p − 1)] hold the partial-match information from the previous block and the current block, respectively. Vin[i] is set if there is a partial match in the previous block up to and including the character Pat[i − 1]. Vout[i] is set if there is a partial match up to and including the character Pat[i] in the current block.

For each block of the data string, Mask is initially set to be all TRUE, enabling all comparators for the entire block. Vout is initialized to be all FALSE. Then, each pattern character Pat[i], $1 \le i \le p$, is serially broadcast to the comparator array and compared to the entire data block. If $i < p$, the broadcast character is not the last pattern character, and the comparison result of Pat[i] and Blk[b] is stored in Vout, so that any partial match can be continued in the next block. If $i = p$, the last pattern character is broadcast, so any match in the current comparison cycle indicates that an occurrence of the pattern is found in the current block. The positions at which matches are detected are reported. Since the last pattern character is reached, no further comparison is necessary; and the current block can he discarded.

If Pat[i], $i < p$, matches with any of the first $(b - 1)$ characters in the data block, the search continues with Pat[i + 1] for the current block. The comparison result of Pat[i] and Blk[b] indicates only whether a partial match occurs in the current block and does not require further comparison in the current block. If the search is continued with the current block, the Mask for the next comparison cycle is formed by shifting the comparison results of the current cycle. The first bit of the Mask is loaded from Vin[i + 1] to continue any partial match from the previous block.

If no match is detected with the first $(b - 1)$ characters, the algorithm checks to see if any partial match has to be continued for the rest of the pattern characters, Pat[(i + 1):p]. If so, the first pattern character found will be broadcast in the next comparison cycle. In this case, only the first bit of the Mask is set. If no partial match is carried over from the previous block, the current block can be discarded.

With the use of Vin and Vout, partial match can be continued in the next block without any backtracking of the data string. Since the algorithm is independent of the block size, the search rate can be increased simply by increasing the block size or the number of comparators. As a result, the DPPM algorithm can efficiently utilize the high degree of integration of VLSI technology to attain high-speed processing through parallelism.

3.3 VLSI Design Issues

The DPPM algorithm relies on VLSI technology to implement broadcasting of pattern characters to many comparators simultaneously. Broadcasting requires a large fanout as well as a long routing distance. It is therefore important to understand whether data broadcasting can be implemented effectively using current and emerging VLSI technology.

The first design issue is whether the necessary degree of fanout can be achieved without introducing excessive delay. Using a block size of 1000, each bit of a pattern character must be broadcast to 1000 gates. Using 2-μm CMOS technology, a typical gate capacitance is 20 fF, so the total input capacitance for 1000 gates is 20 pF. Assuming a typical gain factor of 50

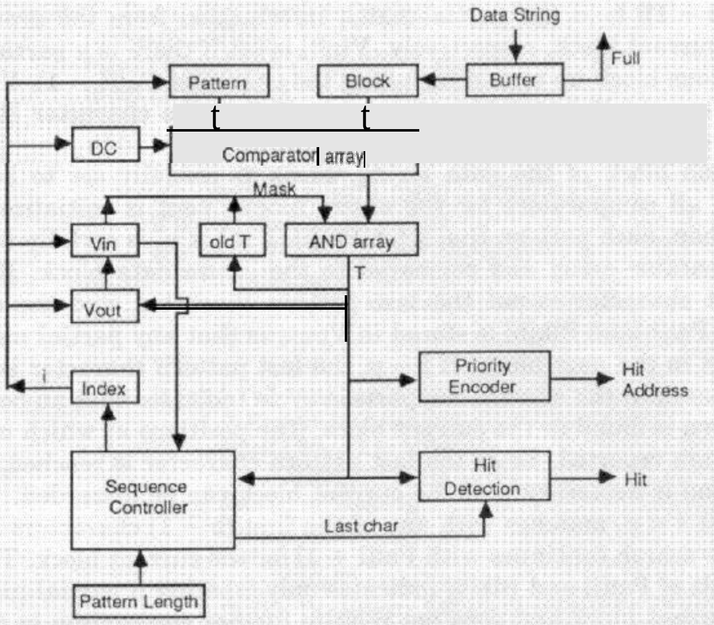

Fig. 5. Circuit block diagram of the DPPM engine.

$\mu A/V^2$ and an output channel resistance of 10 $K\Omega$ using the simplified RC model presented in [25], it is possible to achieve a 1 ns delay for a buffering stage with a fanout of 4. With 5 cascaded buffer stages to achieve a fanout of 1000, a total delay of 5 ns is possible. This estimate matches the circuit simulation results using 2-μm CMOS technology very well. Using 2-μm BiCMOS [13] technology, one BiCMOS gate can drive a load of 700 fF with 0.7 ns delay. This is equivalent to driving 35 gates in parallel. A fanout of 1000 can therefore be achieved in two cascaded buffer stages with a total delay of only 1.4 ns.

Propagation delay due to long routing distances can be minimized by using two layer metal routes. Since a large capacitive load is driven using multiple stages of buffers, the propagation delay is dominated by the large input gate capacitance and high output channel resistance of the previous stage; the delay due to long metal runs can be ignored in the timing analysis [25].

The above analysis shows that with the large fanout capability of BiCMOS technology, together with two-layer metal routes, data broadcasting in the DPPM algorithm can be implemented effectively. Even with 2-μm CMOS technology, data broadcasting can still be implemented with a reasonable propagation delay.

Figure 5 shows the circuit block diagram of the DPPM engine. Before the actual search operation, the pattern, pattern length, and don't care positions are first. loaded into their corresponding registers. The data string is buffered and read one block at a time to the block register. The comparator array performs the actual comparison between the pattern character and the data block. The results of the comparator array are ANDed with the Mask to form

T. The DPPM engine integrates the mismatch detection and the partial-match propagation mechanisms by combining the Vin register (partial-match information from a previous block) with the oldT register (match results of the previous comparison cycle) to form the mask for each comparison cycle. The first bit of the Mask is from Vin[i], and the last $(b - 1)$ bits are from the first $(b - 1)$ bits of T in the previous cycle. The last bit of T is stored into Vout[i]. Each time a new data block is read, the first $(p - 1)$ bits of Vout are loaded into the last $(p - 1)$ bits of Vin. The first bit of Vin is always set.

The sequence controller controls the operation of the DPPM engine by generating the value i, which is used to index the pattern, don't care, Vin, and Vout registers. By monitoring the values of T, the content of the Vin register, and the pattern length, the sequence controller decides for each cycle one of the following three actions:

(1) Compare the next pattern character with the current block.
(2) Jump to a pattern character to continue the partial match from previous block.
(3) Discard the current block and continue the search with the next block.

Step 1 is taken if this is not the last pattern character and the content of T is nonzero (match(es) detected in the current cycle). If T is zero, then the index of the next pattern character to be used for comparison is determined by a priority encoder that encodes the first nonzero bit in the Vin register after masking off the first $(i - 1)$ bits of the Vin register using a linear shift register. Step 3 is taken if the last pattern character is reached or if T is zero and there is no more partial-match propagation from the previous block (early out).

The sequence controller also generates a *lust character* signal when the last pattern character is reached. This signal is used by the hit detection unit, which checks the values of T to report any hit in the search. The priority encoder produces the encoded addresses for all hit positions in the current block.

The critical path of the circuit is from the pattern register, through the comparator and AND arrays, to the priority encoder. Using 2-μm CMOS, the comparison cycle time is about 50 ns for a block size of 1000. This cycle time includes the broadcasting delay of 6 ns. If 1.2-μm CMOS is used, the cycle time will be approximately 33 ns. The chip area of the DPPM engine for a block size of 128 is roughly 200 x 100 mil^2.

Another important issue is to understand whether the I/O bandwidth of a CMOS chip can sustain the gigabyte/sec search rate of the DPPM engine. Marcus [16] has demonstrated, using 32 input pads, that a bandwidth of 5.44 Gb/s can be achieved using 1.2-μm CMOS technology. Even higher bandwidth can be achieved by using advanced packaging technology that supports higher I/O pin counts [11].

4. PERFORMANCE ANALYSIS

Performance of the proposed document-searching architecture depends on the search rate achievable by the DPPMEs, which in turn depends on the performance of the DPPM algorithm. In this section the DPPM algorithm is analyzed both analytically and by simulation on a text database. Using the performance results of the DPPM algorithm, the potential performance of the proposed document-searching architecture is analyzed.

4.1 DPPM Algorithm

4.1.1 Analytical Modeling. Performance of the DPPM algorithm depends on the block size used, the number of comparison cycles required per block, and the cycle time of the hardware. The block size is a parameter chosen by the designer of the system. The cycle time depends on the VLSI technology used in implementing the DPPM algorithm. The number of comparison cycles required per block depends on the probability of finding the pattern in a block, and may be determined analytically or experimentally. In this section an approximate analytical analysis of the performance of the DPPM algorithm is presented. To simplify the analysis, a uniform distribution of characters in both the data string and the pattern is assumed. We also assume that the pattern does not contain any *don't care* characters. Using an approach similar to that in [18], the pattern-matching process is modeled as the detection of random events.

With a block size of b, there are b independent comparisons of the pattern with the data block, with each starting at a different location. Thus, partial matches that span to the next block are considered to be part of the current block. Let p be the pattern length and m the alphabet size of the character set. Using the assumption of uniform distribution of characters, the probability of not finding the pattern in each comparison is

$$\alpha = 1 - \left(\frac{1}{m}\right)^p.$$

Assuming that not finding the pattern in one position of the data string does not affect the probability of not finding the pattern at a different position, then the probability of not finding the pattern for b consecutive comparisons is

$$\beta = \alpha^b = \left(1 - \left(\frac{1}{m}\right)^p\right)^b.$$

Therefore, the probability of finding a pattern with length p in b consecutive comparisons is

$$\delta(p) = 1 - \beta = 1 - \left(1 - \left(\frac{1}{m}\right)^p\right)^b.$$

6(p) converges quickly to zero as p increases. The average number of comparison cycles per block is

$$C = 1 + \sum_{i=1}^{p-1} \delta(i).$$

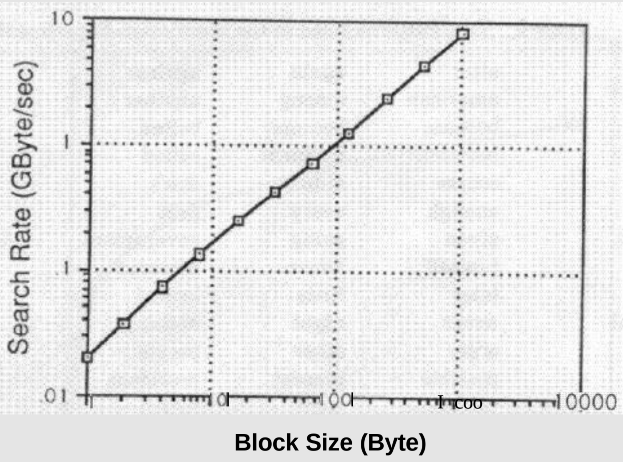

Fig. 6. Search rate of the DPPM algorithm using analytical results.

Let T be the comparison cycle time. The search rate at a block size of b, R_b, is defined as the number of data string characters that can be searched in one second,

$$R_b = \frac{b}{C \times T}.$$

The speedup, S, of the DPPM algorithm is defined as

$$S = \frac{R_b}{R_1}.$$

Using 50 ns as the comparison cycle time and a pattern length of 6, Figure 6 shows the search rate of the DPPM algorithm at different block-sizes. The alphabet size m, used in this case is 45, which corresponds to the case-insensitive English alphabet, digits, and a few common symbols. When the block size is one, the DPPM algorithm degenerates into a sequential algorithm with a search rate of approximately 20 MByte/sec. This search rate is about the same as current state-of-the-art hardware pattern matchers. The DPPM algorithm exhibits a high degree of parallelism; according to the analysis, a multiple gigabytes per second search rate is achievable using a block size of 1024.

4.1.2 Simulation *Experiment.* The analysis described in the previous section is only a simplified model of the actual performance of the DPPM algorithm. Exact analysis of the algorithm is difficult due to the following two reasons:

(1) Different characters have different frequencies of occurrence. For instance, in the English language, a, e, and s occur more frequently than q, x, and z. The assumption of uniform distribution of characters in the data string and the pattern is a simplification that may have a significant effect on the accuracy of the analysis.

Table I. Test Patterns Used in the Simulation Experiment

about	after	again	against	almost
always	american	among	mother	around
asked	because	become	before	being
better	between	business	called	children
could	course	didn't	don't	during
early	enough	every	first	found
general	given	going	government	great
group	himself	house	however	important
large	later	little	looked	might
national	never	night	nothing	number
often	order	other	people	place
point	possible	present	president	program
public	rather	right	school	second
several	should	since	small	social
something	state	states	still	system
their	there	these	things	think
those	though	thought	three	through
toward	under	united	until	water
where	which	while	white	within
without	world	would	years	young

(2) The occurrence of a character is not independent of its neighboring characters. Character groups like ing, th, and un occur frequently in the English language. This cross correlation property implies that we cannot treat the occurrence of a character in the input data string or the pattern as an independent random event; this complicates the analysis.

The DPPM algorithm can be best evaluated by simulating its operation on a real text database. In such an environment we can estimate its performance and gain insights into its behavior. The database chosen for this simulation consists of the Associated Press wire news articles of August 2, 1988. The total size of this database is 4.4 MByte. All uppercase characters in the database were converted to lowercase; case-insensitive pattern matching was used. The test patterns chosen for this simulation experiment were the 100 most frequently used words in American English that are at least five characters long (see Table I) [14]. The pattern lengths vary from 5 to 10 characters with an average of 5.88 characters.

Figure 7 shows the average number of comparison cycles per block, C, measured at different block sizes. Although the pattern characters are serially compared to the data block, early mismatch detection allows the algorithm to search the next block as soon as a mismatch is detected. This feature is especially effective at smaller block sizes where the probabilities of matching the first few characters of the pattern are low. Without early mismatch detection, C is equal to the average pattern length, in this case, 5.88. At larger block sizes, the probability of finding the pattern in the block is higher; thus the value of C also increases. As the block size is increased, the value of C approaches the average pattern length asymptotically.

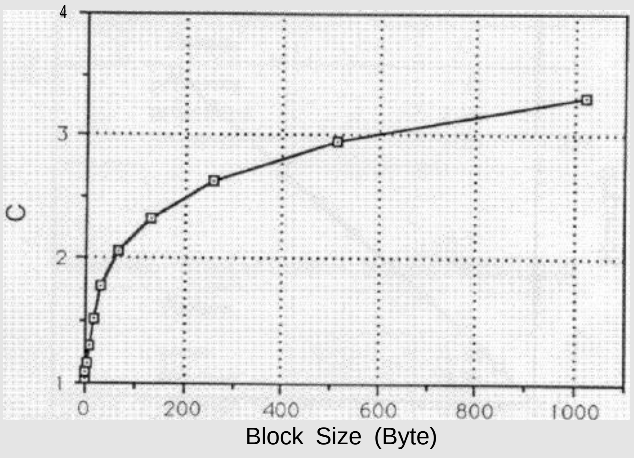

Fig. 7. Average number of comparisons per block.

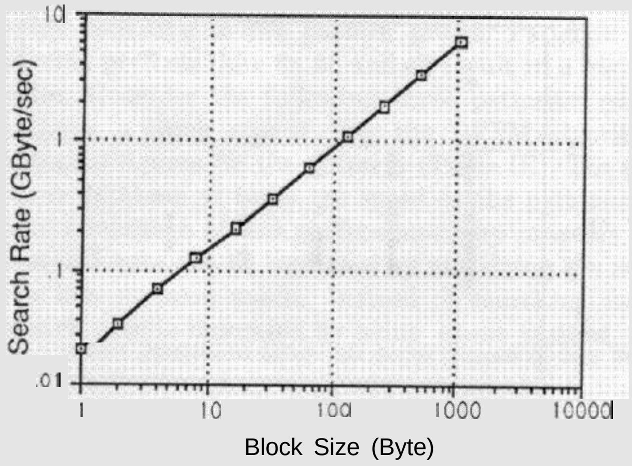

Fig. 8. Search rates at different blocks.

using 50 ns as the comparison cycle time. T, Figure 8 shows the search rates R_b at different block sizes. Recall that increasing the block size requires proportionately more comparators on chip. At a block size of 16, the search rate is 212 MByte/sec. This rate matches the predicted optical disk transfer rate of 200 MByte/sec [1]. At a block size of 128, the search rate reaches 1 GByte/sec. This rate is sufficient to handle the existing memory bandwidth of supercomputers, as well as data input from optical-fiber transmission systems in future communication networks.

Figure 9 shows the speedup, S, of the DPPM algorithm at different block sizes. The DPPM algorithm is indeed scalable and allows exploitation of a high degree of parallelism. Speedup can be obtained easily by increasing the size of the data block.

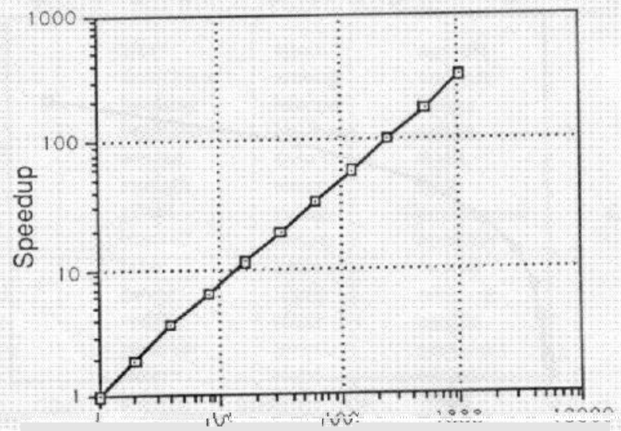

Fig. 9. Speedup of the DPPM algorithm.

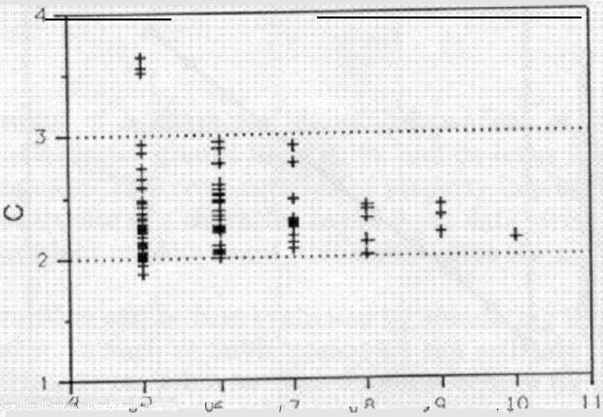

Fig. 10. C for all patterns at a block size of 128.

Intuitively, if the pattern is longer, more comparison cycles per block should be required in testing the pattern against the data string. However, this is not the case for the DPPM algorithm as observed in this simulation study. Figure 10 shows the values of C for all patterns at a block size of 128. This figure indicates no strong correlation between the pattern length and C. This insensitivity to the pattern length is due to the early mismatch detection in the DPPM algorithm. Since most blocks can be discarded after the first few characters, the length of the pattern has very little effect on C.

The algorithm is also insensitive to the last few characters, or suffix, of the pattern. Using a block size of 128, the patterns *processes*, *processor*, and *processing* require essentially the same number of comparison cycles per block (see Table II).

Table II. Effect of Different Pattern Suffixes on C

Pattern	C
processes	2.28
processing	2.28
processor	2.28

Table III. Effect of Different Pattern Prefixes on C

Pattern	C
queue	1.16
dequeue	2.38
enqueue	2.58

Since the result of a comparison cycle determines whether another comparison cycle is necessary for the current data block, the performance of the algorithm is very sensitive to the pattern prefix. Patterns starting with frequently occurring prefixes (like th, st, and al) result in a higher number of comparison cycles. Consider the following three patterns: *queue*, *dequeue*, and *enqueue*. Using a block size of 128, the DPPM algorithm has very different performance figures for the patterns (Table III). The pattern *queue* searches the same database at twice the speed as the patterns *dequeue* and *enqueue*. To take advantage of this prefix-sensitivity property of the DPPM algorithm, the search rate can be increased by modifying the pattern so that it starts with a less common prefix. Instead of searching for the pattern *dequeue*, the search rate is increased by using queue instead. However, the search result has lower precision since all occurrences of the patterns *queue* and *enqueue* will also be found.

Instead of comparing the pattern characters serially to the data block as in the DPPM algorithm, it is possible to compare multiple pattern characters to the data block with Multiple Pattern Multiple Data (MPMD). This approach reduces the number of comparison cycles required per block, and thus results in a higher search rate. However, MPMD requires more hardware comparators for the same data block size since multiple pattern characters have to be compared simultaneously. The additional comparators may also be used to increase the block size of the DPPM algorithm, and thereby increasing the search rate. So the design issue is this: Given the same number of comparators, which of these two approaches (DPPM or MPMD) yields a higher search rate?

Let the number of comparators used in both cases be k. The block size for the DPPM algorithm is also k. The block size for the MPMD algorithm is k/n, if n characters of the pattern are to be compared in parallel. If c comparison cycles are required for a data block of size k/n for the DPPM algorithm, then the number of comparison cycles required for the MPMD algorithm, for the same block, is $\lceil c/n \rceil$.

Table IV. Comparison of the DPPM and MPMD Algorithms

	DPPM		MPMD	
k	C	R_k	C	$R_{k/2}$
1	1.04	0.02	—	—
2	1.08	0.04	1.00	0.02
4	1.16	0.07	1.01	0.04
8	1.30	0.12	1.02	0.08
16	1.51	0.21	1.03	0.16
32	1.77	0.36	1.07	0.30
64	2.06	0.62	1.12	0.57
128	2.34	1.09	1.22	1.05
256	2.63	1.95	1.35	1.89
512	2.95	3.48	1.53	3.34
1024	3.30	6.20	1.74	5.88

Based on simulation results, Table IV compares the average number of comparison cycles, C, and search rates, R_k, of both approaches, given the same number of comparators when n is 2. Although the average number of comparison cycles per block is lower in the MPMD algorithm, the overall search rate is less than that of the DPPM algorithm. Comparing more than one pattern character at a time does not make full use of the capacity of the comparators. If the first pattern character fails, all comparators used for the second pattern character are wasted. Using the extra comparators to increase the data block size is more profitable than increasing the number of pattern characters broadcast simultaneously in terms of the overall search rate.

We can now return to the document-searching architecture and evaluate its potential performance, given the performance of an individual DPPME.

4.2 Document-Searching Architecture

To determine the information retrieval query-processing potential of the proposed document-searching architecture, an analysis of the ratio of the number of DPPMEs to the PE is required. Assume that the arrival process of matches for each DPPME is Poisson distributed with a rate of λ. Also, assume that the arrival processes for all n DPPMEs in parallel are independent with identical distributions; the total input arrival rate is then $n\lambda$. The PE is modeled as an exponential server with a service rate of μ. Hence, we can use an M/M/1 model to analyze the potential performance of the proposed architecture.

The utilization, ρ, of the PE is

$$\rho = \frac{n\lambda}{\mu}.$$

Therefore,

$$n = \frac{\rho\mu}{\lambda}.$$

Table V. Arrival Rates of DPPM Engine

Block size	$\lambda (s^{-1})$
1	2050
2	4000
4	7450
8	13300
16	22900
32	39000
64	67300
128	118000
256	210000
512	375000
1024	670000

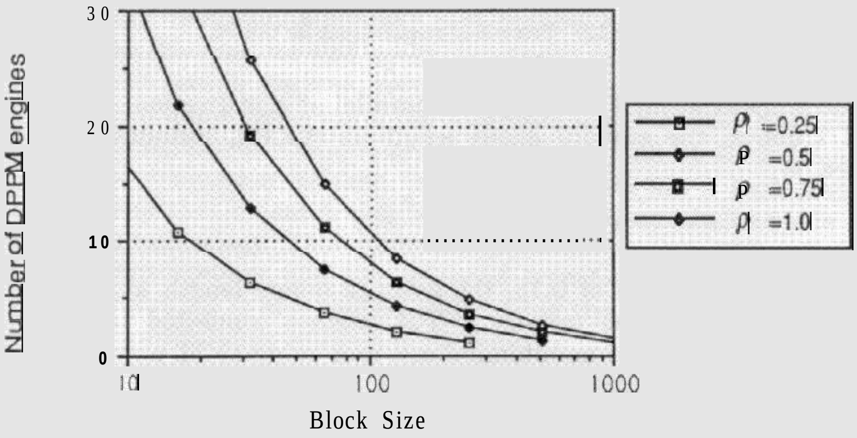

Fig. 11. Number of DPPM engines supported by a 10-MIPS PE for various values of PE utilization, ρ.

The arrival rate of matches for each DPPME, λ, is determined by the ratio of the number of matches to the time required to search the entire database. Using the simulation results of the Associated Press wire news article in the previous section, the average number of matches for the 100 test patterns used in the experiment is 472. The time required to search the entire database is determined by the search rate measured for different block sizes, as shown in Figure 8. Table V shows the values of λ at different block sizes using the simulation results in the previous section.

The service rate of the PE for each match depends on the type of instruction executed at the processor. A conservative analysis of the instruction set in Figure 2 shows that about ten machine instructions are required to process each match from the DPPME. Assuming that we have a 10-MIPS processor, the service rate will then be $10^6 s^{-1}$.

Figure 11 shows the number of DPPMEs that a 10-MIPS PE can support at different block sizes and utilization levels. At a block size of 64, a IO-MIPS

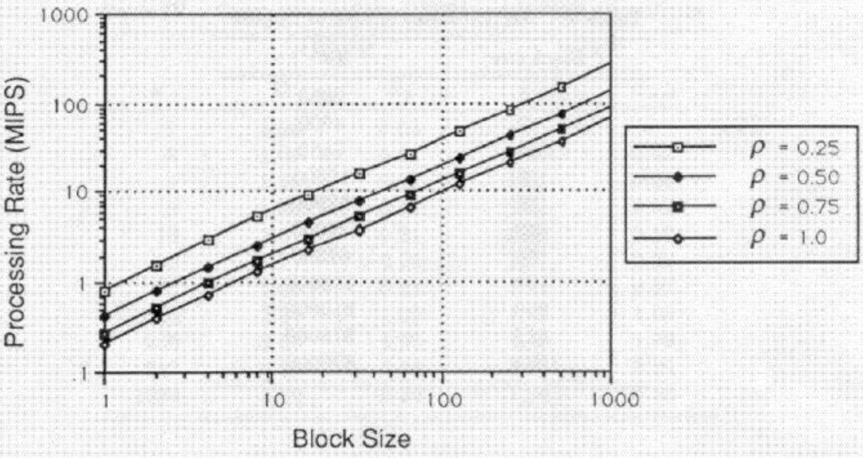

Fig. 12. Processing requirement for the PE.

PE can support about 11 DPPMEs in parallel at a utilization of 0.75. For a M/M/l queue, the average queue length at the PE, including the one in service, is

$$Q = \frac{\rho}{1-\rho}.$$

Hence, at a utilization of 0.75, the average queue length at the PE is 3. Using this combination (eleven DPPM engines connected to a 10-MIPS processor), the proposed architecture can process up to 11 pattern-matching operations simultaneously at a search rate of 600 MByte/sec.

Fixing the number of DPPM engines at ten, Figure 12 shows the processing rate requirement for the processor at different block sizes and utilization levels. With advances in VLSI technology and RISC architectures, the processing rates of microprocessors are increasing at an enormous rate. 50-MIPS RISC processors have recently been announced commercially by a number of vendors. From Figure 12, a 50-MIPS PE can support 10 DPPM engines of block size 128 in parallel at a utilization level of 0.25, with each DPPME having a search rate of 1 GByte/sec. At a search rate of 1 GByte/sec, we can search both the old and new testaments of the Bible in 5 msec, Webster's dictionary in 16 msec, and an entire volume of *The Encyclopedia Britannica* in 400 msec.

The above analyses show that by separating the operator and query complexity from the DPPMEs, the proposed architecture can search documents at 1 GByte/sec using current VLSI technology. Because of the high search rate supported by the DPPMEs, the system bottleneck is now shifted from processing to disk I/O bandwidth: current disk technology cannot deliver data at such a high rate. This problem can be solved partially by using a large number of disks in parallel (disk farm), thus increasing the total I/O bandwidth of the disk system. In the future, we will have to rely on different storage technologies such as optical disk and holographic storage systems. The high-performance optical disk reported in 1221 has a 40

MByte/sec transfer rate; and it was estimated that the transfer rate may reach 200 MByte/sec in future [1]. The MCC Bobcat II holographic storage project 1191 proposed a 100 to 800 MByte/sec transfer rate prototype using 3-dimensional holograms in photorefractive crystals, and projected that a 50 GByte/sec transfer rate is achievable using this technology.

5. SUMMARY

We have proposed a novel searching architecture for information retrieval. This architecture decomposes information retrieval instruction-processing requirements from document-searching needs. Document searching is performed via numerous parallel high-speed hardware pattern matchers. Although any filter design can be utilized as a document processor, our analysis is based on DPPME, a VLSI filter based on a parallel pattern-matching algorithm we developed, called DPPM.

The proposed document-searching architecture has a modular structure that. separates the operator and query complexity from the customized hardware document search engines, resulting in simpler, and hence more efficient, hardware implementation of the document search engine. Performance of the DPPM algorithm was evaluated analytically and by simulation. The results of the evaluation indicate that the algorithm has a high degree of parallelism and that. a search rate of over 1 GByte/sec is achievable using current VLSI technology. This search rate exceeds the memory bandwidth of existing supercomputcrs and the projected transfer rate of-future optical disks, The algorithm is also capable of handling the very high bandwidth of optical fiber transmission systems of the future.

Using the detailed performance analysis of each DPPME, we analyzed the expected system performance. As noted earlier, we estimate that-to fully search *The Encyclopedia Britannica* with a query of at most, ten patterns takes roughly 400 mscc. This processing rate far exceeds current conventional technology.

As the sizes of unformatted text databases rontinue to grow, it becomes inevitable that even forming indices for text documents will require considerable effort. It is possible that information retrieval systems employing parallel hardware filters, such as the one proposed here, will be used to remedy this problem.

ACKNOWLEDGMENT

We would like to thank Gary Herman, Sudhir Aggarwal, and Bill Mansfield for their support in this project. We would also like to-thank Brian Coan, Jonathan Chao, Gita Gopal, Dik Lee, Will Leland, Anthony McAuley and the anonymous reviewers for their comments on this paper, which greatly improved both its presentation and technical content.

REFERENCES

1. BERRA, P. B., AND TROULLINOS, N. B. Optical techniques and data/knowledge base machines. *IEEE Computer* 20 (Oct. 1987), 59-70.
2. BOYER, R. S., AND MOORE, J. A fast string searching algorithm. *Commun. ACM* 20 (Oct. 1977), 762-772.

3. CURRY, T., AND MUKHOPADHYAY, A. Realization of efficient non-numeric operations through VLSI. In *Proceedings of the VLSI '83* (1983). 327-336.
4. FOSTER, M. J., AND KUNG, H. T. The design of special purpose chips. *IEEE Computer 13* (Jan. 1980), 26-40.
5. FRIEDER, O., LEE, K. C., AND MAK, V. W. JAS: A parallel VLSI architecture for text processing. *IEEE Data Eng. 12* (Mar. 1989), 16-22.
6. HALAAS, A. A systolic VLSI matrix for a family of fundamental search problem. Integration *VLSI J. 1* (Dec. 1983), 269-282.
7. HASKIN, R. L. Special-purpose processors for text retrieval. *IEEE Data Eng. 4* (Sept. 1981), 16-29.
8. HASKIN, R. L., AND HOLLAAR, L. A. Operational characteristics of a hardware-based pattern matcher. *ACM Trans. Database Syst. 8* (Mar. 1983), 15-40.
9. HOLLAAR, L. A. Text retrieval computer. *IEEE Computer 12* (Mar. 1979), 40-50.
10. HOLLAAR, L. A., SMITH, K. F., CHOW, W. H., EMRATH, P. A., AND HASKIN, R. L. Architecture and operation of a large, full-text information-retrieval system. In *Advanced Database* Machine *Architecture.* Prentice Hall, 1983, 256-299.
11. JOHNSON, R. Multichip modules: Next-generation packages. *IEEE Spectrum 27,* (Mar. 1990), 34-48.
12. KNUTH, D. E., MORRIS, J. H., JR., AND PRATT, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6* (June 1977), 323-350.
13. KUBO, M., MASUDA, I., MIYATA, K., AND OGIUE, K. Perspective on BiCMOS VLSI's. *IEEE J. Solid-State Circuits 23* (Feb. 1988), 5-11.
14. KUCERA, H., AND NELSON-FRANCIS, W. *Computational Analysis of Present-Day American English.* Brown University Press, Providence, RI, 1967.
15. LEE, D. Altep-A cellular processor for high-speed pattern matching. New *Generation Comput. 4* (Sept. 1986), 225-244.
16 MARCUS, W. A CMOS batcher and banyan chip set for B-ISDN packet switching. *IEEE J. Solid-State Circuits SC-25, 6* **(June 1990), 1426-1432.**
17 MEAD, C. A., PASHLEY, R. D., BRITTON, L. D., YOSHIAKI, T., AND SNADO, S. F., JR. 128-bit multicomparator. *IEEE J. Solid-State Circuits SC-11* (Oct. 1976), 692-695.
18 PRAMANICK, S. Performance analysis of a database filter search hardware. *IEEE Trans. Comput. C-35* **(Dec. 1986), 1077-1082.**
19 REDFIELD, S., AND HESSELINK, S. Data storage in photorefractives revisited. In *Proceedings of the Conference on Optical Computing,* **vol. 963, 1988, 35-45.**
20 SALTON, G., FOX, E. A., AND WU, H. Extended boolean information retrieval. *Commun. ACM 26 (Nov.* 1983). 1022-1036.
21. SCHEFTER, J. Super searcher. *Popular Sci. 231* **(Dec. 1987), 60-61.**
22. SHULL, T. A., HOLLOWAY, R. M., AND CONWAY, B. A. NASA spaceborne optical disk recorder development. *In SPIE 899 Optical Storage Technology and Applications,* 1988, 272-278.
23. STANFILL, C., AND KAHLE, B. Parallel free-text search on the connection machine system. *Commun.* ACM 29 (Dec. 1986), 1229-1239.
24. TAKAHASHI, K., YAMADA, H., AND HIRATA, M. Intelligent string search processor to accelerate text information retrieval. In *Proceedings of the Fifth International Workshop on Database Machines* (Oct. 1987), 440-453.
25. WESTE, N., AND ESHRAGHIAN, K. *Principles of* CMOS *VLSI* Design. Addison-Wesley, Reading, Mass., 1985.