A Keyword Match Processor Architecture using Content Addressable Memory

Long Bu
long.bu@uconn.edu

John A. Chandy
john.chandy@uconn.edu

Dept. of Electrical and Computer Engineering
University of Connecticut
Storrs, CT 06269-1157 USA

ABSTRACT

This paper demonstrates a keyword match processor capable of performing fast dictionary search with approximate match capability. Using a content addressable memory with processor element cells, the processor can process arbitrary sized keywords and match input text streams in a single clock cycle. The processor is capable of determining approximate match and providing distance information as well. A 64-word design has been developed using 19000 transistors and it could be expanded to larger sizes easily. Using a modest 0.5 micron process, we are achieving cycle times of 8 ns and the design will scale to smaller feature sizes.

Categories and Subject Descriptors: B.M [Hardware]: Miscellaneous; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Algorithms, Performance, Design

Keywords: keyword matching, text search hardware

1. INTRODUCTION

Searching data for specific keywords or signatures is an important operation in a wide variety of applications including full-text-search in databases, search engines, and dictionaries for natural language processing. Network protocol operations such as IP routing table lookups, authentication directory lookups, and intrusion detection also require the ability to match input keys on some form of data dictionary. For the most part these search/match operations have been done in with efficient software algorithms [1, 2, 3, 4, 5].

In applications that require fast search times, application specific hardware implementations have been introduced [13, 8, 10, 14], and for the most part, these hardware implementations have included the use of content addressable memories (CAM). CAMs allow keyword matches to be done in a single cycle making search a constant time operation rather than a linear (or worse) time operation when done with RAM based software algorithms. With data streams, the CAMs can be used to accelerate search in linear time relative to the length of the input stream rather than the size of the dictionary. The benefits of CAMs have led to their use in a variety of networking applications including routing lookups, packet classification and address translation.

One of the drawbacks of CAMs, however, is its reliance on fixed size keys. When searching for matches, the input keys must all be the same size. For applications like routing table lookups, where the IP addresses are all the same size, this limitation is not critical. However, for other applications such as directory lookups or signature matching for intrusion detection, the key sizes vary and thus the fixed key size restriction for CAMs can prove to be problematic.

In the context of keyword matching, there has been early work in hardware acceleration particularly for database operations and AI natural language processing [6]. The use of content addressable memories for text search has centered around two strategies: cellular automata [12, 10, 11] and finite state machines [14, 7]. The cellular automata methods cascade CAMs temporally and then use pointers to propagate matches from one CAM set to another [9]. Motomura et al have described an cellular automata based architecture to do dictionary search in VLSI [11]. The keywords grouped in 4 characters segments and they allow for concatenating groups to create larger keywords. However, the grouping by four means that keywords that are not multiples of the group size will result in wasted CAM storage space. Hirata, et al. have designed a FSM-based CAM search architecture that also accommodate variable length keywords by grouping keywords into segments.

In an attempt to address this problem, we have designed a keyword match architecture to process input data streams and allow for arbitrarily sized keys. The design is based on temporally cascaded CAMs and is an extension of Motomura's cellular automata structure. Processor element cells, external to the CAM array, will process character match signals from the CAM and output keyword match signals. The architecture is flexible enough to allow for "approximate word" matches as well. We present the architecture in the following section and then discuss the implementation and performance results in subsequent sections.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-853-9/04/0004 ...$5.00.

Keyword	Distance	
WHALES	1	Deletion of 'S'
SHALE	1	Substitution of 'S' with 'W'
CUE	2	Substitution of 'C' with 'B'
		Insertion of 'L'
BLUE	0	Exact Match

Table 1: Keyword Match Distance given two input strings T_1 = "BLUE" and T_2 = "WHALE"

2. KEYWORD MATCH PROCESSOR ARCHITECTURE

We first define the keyword match problem as follows. Assume a sequence T of length p characters and a set, K, of keywords of arbitrary lengths. Each character has m bits. The goal is to determine if T matches any keyword $k \in K$. T is said to *exactly* match a keyword, k, if $|k|$, the length of keyword k, is equal to p and $T_i = k_i$ for all $i \leq p$. T can *approximately* match a keyword, k, if the distance, $d(T, k)$, is less than some criterion. The distance is the sum of insertions, substitutions, and deletions occurring between T and k. A distance of 0 indicates an exact match. Examples of match distance are shown in Table 1.

We have implemented a hardware solution to the keyword match problem with a keyword match processor (KMP). An architectural overview of the KMP is shown in Figure 1.

2.1 Control Circuit

The Control Circuit is a simple state machine that manages the data flow of the input stream and resultant outputs. A `START` input signal resets the CAM and PE arrays and readies the system to start accepting an input data stream. The `RESUME` output signal is asserted when the KMP has finished processing the input data and is ready for new data word to be matched.

2.2 CAM Array

The CAM stores the keywords as m-bit characters in an m by n array of CAM cells where n is equal to the number of characters in the CAM, i.e. the sum of the lengths of all keyword in K. For keywords comprised of ASCII characters, as would be the case for text search, the character size, m, would be set to 7. For more general byte-based keywords, m is equal to 8. Since the CAM is a sequence of variable length keywords, we need some way to separate the keywords. To do so, we insert between keywords a delimiter appropriate to the application - such as "space" or 0xFF for text search. Certain applications may not have any appropriate delimiters such as when all possible characters are valid. We will discuss mechanisms to address this case later.

Each column in the CAM array corresponds to a character in a keyword. Each cell is a normal CAM cell with a 6-transistor RAM cell with an additional 3 transistors for circuitry to indicate a match between the stored bit and the incoming bit. If there is no match, the output match line is pulled low. The column match line is shared as a wired-NOR line between all cells in the column. When an input m-bit character is applied to a column, the column match signal is active high when all m bits match. The input character is applied simultaneously to all n columns in the array.

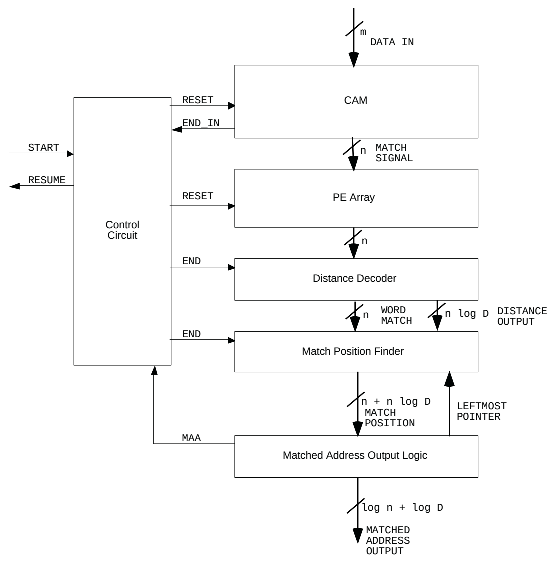

Figure 1: Keyword Match Processor Architecture.

2.3 PE Array

The CAM array described so far is similar to what would be seen in any traditional CAM design. The uniqueness of the KMP is in the processor element (PE) array. The PE array is similar in concept to the cellular automaton processor arrays proposed by [10, 11]. The array is comprised of an n x $D+1$ array of processing elements (PE) where each PE is an finite state machine that carries out the match algorithm described below. The PE holds a binary value called a *flag* which is used in calculating match distance. The maximum distance, D, that can be calculated is equal to one less than the number of rows in the PE array. With a 3 row array, as shown, the maximum distance that can be calculated is 2.

Let us now describe the keyword algorithm carried out by the PE array. We label the PE flags, $PE(i, j)$, and the incoming match signals from the CAM array are labeled $M(i)$, where i is the character number and j is the PE array row number. At the first clock cycle of the keyword match process, we reset both the CAM and PE arrays. The CAM array reset causes a delimiter to be presented to the CAM which activates match signals at every position i where there is a delimiter, i.e. the end of each keyword in the CAM. When the PE array is reset, $PE(i, 0)$ is set to 1 when $M(i-1) = 1$ or $i = 0$ and all other $PE(i, j)$ are set to 0.

On each subsequent clock cycle, a character from the input sequence T is presented to the CAM array and is simultaneously compared against all the characters in all the keywords in K. Any character matches are passed to the PE array for match distance processing. The PE array calculates distance by transferring flags from one PE to the next using a set of rules as described below. Time t indicates the current clock cycle and time $t+1$ is the next clock cycle.

PE flag processing rules

1. If there is a match signal at column i and a flag exists at $PE(i, j)$, the flag is passed to $PE(i+1, j)$ at the next clock cycle to represent the situation that it is an exact character match. The flag remains on row j to indicate no change in the distance when there is

an exact character match. The flag is also passed to $PE(i, j+1)$ to cover the possibility of an insertion character that also matches.

```
if $M_t(j) = 1$ and $PE_t(i,j) = 1$ then
  $PE_{t+1}(i+1, j) = 1$
  $PE_{t+1}(i, j+1) = 1$
```

2. If there is a mismatch signal at column i and a flag exists at $PE(i, j)$, the flag is passed to $PE(i+1, j+1)$ at the next clock cycle to represent the substitution case. The flag is also passed to $PE(i, j+1)$ to cover the chance of an inserted character that does not match.

```
if $M_t(i) = 1$ and $PE_t(i,j) = 1$ then
  $PE_{t+1}(i+1, j+1) = 1$
  $PE_{t+1}(i, j+1) = 1$
```

3. Whenever there is a flag at $PE(i, j)$, flags are set at $PE(i+1, j+1), PE(i+2, j+2)\dots$ to the boundary of the PE array. This rule handles the deletion case.

```
if $PE_t(i,j) = 1$ then
  for $(1 \geq d \leq D - j)$
    $PE_t(i+d, j+d) = 1$
```

Each $PE(i, j)$ is designed to look to the up, upper left, and left for flag signals $(PE(i, j-1), PE(i-1, j-1), PE(i-1, j))$ as well as the match signal to its left $(M(i-1))$. The rules to determine the next state are then implemented using the following logic equation.

$$\begin{aligned} PE_{t+1}(i,j) = {} & PE_t(i, j-1) + PE_t(i-1, j) \cdot M(j-1) \\ & + PE_t(i-1, j-1) \cdot \overline{M(j-1)} \\ & + PE_{t+1}(i-1, j-1) \end{aligned} \quad (1)$$

2.4 Distance Decoder

The last character of T should be the delimiter. When this happens, $M(i)$ will be set at all locations of the CAM where there is a delimiter and an `END_IN` signal is asserted which lets the control circuit know that the input stream has finished. This also causes an `END` signal to be sent to the Distance Decoder and Match Position Finder. If there is a flag present in a delimiter column, as identified with the $M(i)$ signal, we know there is a match. The row position of the flag indicates the distance of the match. The Distance Decoder interprets the row information and match signals to output a word match signal and the distance encoding. It is enabled only when it receives the END signal from the Control Circuit. The distance decoder logic to determine a word match is as follows: $WM(i) = END \cdot M(i) \cdot (PE(i, 0) + PE(i, 1) + \dots + PE(i, D))$

2.5 Matched Position Finder

The distance decoder can only determine a word match at the end of the keyword, but we need to output the start address of the keyword. The Matched Position Finder finds the start address by propagating the word match signal to the left until it reaches a match signal(the previous word end). It can then output a Matched Position (`MP`) location corresponding to the start of the matched keyword. The distance information for this match is passed along with this matched position.

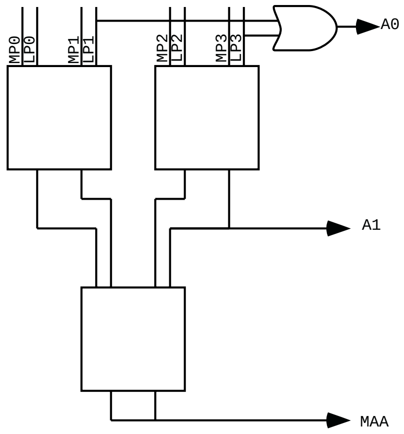

Figure 2: MAO Logic Binary Tree (4 bit example).

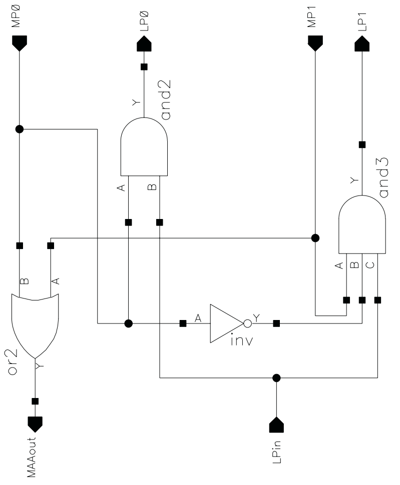

Figure 3: MAO Logic Block.

2.6 Matched Address Output (MAO) Logic

The output of the Match Position finder is an array of n registered `MP` bits. A non-zero entry is an indication that there was a keyword match at that location. This needs to be encoded into a binary address for match processing by an external CPU. Moreover, since there may be multiple matches, these matched addresses must be separated out. The MAO Logic uses a binary tree structure to generate the `MAA` (Matched Address Available) signal which tells the control circuitry that there are matches and to hold off accepting the next input stream. Figure 2 shows the structure for a four-bit tree. The `MAA` signal is generated at the root of the tree and then a `LP` (Leftmost Pointer) signal propagates back up the tree to determine which is the left most `MP` signal. At each level of the tree, a bit of the address is generated from the `LP` signals at that level. The associated distance encoding is output along with the encoded address. When the `LP` signal reaches the top of the tree, it is sent to the MPF logic where it resets the register of the `MP` signal that was just encoded. On the next clock cycle, the MAO can then encode the next left-most `MP` entry. Figure 3 shows the logic for each block in the tree.

2.7 KMP Example

Figure 4 shows an example of the KMP in operation. There are two keywords in the CAM: `UCONN` and `HUSKIES`. On the first clock cycle, $PE(0, 0)$ and $PE(6, 0)$ are set because they represent the start of the two keywords. $PE(1, 1)$, $PE(2, 2)$, $PE(7, 1)$, and $PE(8, 2)$ are also set because of rule 3 described above. With an input string of `CONE`, on the first clock cycle, the character `C` is presented to the CAM. A match on column 1 propagates the flag at $PE(1, 1)$ to $PE(1, 2)$ and $PE(2, 1)$ per rule 1 and then on to $PE(3, 2)$ per rule 3. The gray circles indicate the position of the flag in the subsequent cycle. As characters `O`, `N`, and `E` are applied to the CAM, the flags propagate down and across the PE array. When the input string is complete, the delimiter is applied again to identify the keyword end columns and the flags are processed to determine a match. In the example, there is a flag in column 5, row 2. Since column 5 is a delimiter column, we declare a match. Because the flag is in row 2, the distance is 2 as expected (`CONE` has a deletion, `U`, and a substitution, `E` for `N`).

Since a new input character is presented every clock cycle, the KMP can evaluate an entire input string in linear time relative to the size of the input stream. The processing time is constant relative to the number of keywords. To be exact, the search process takes exactly $p+q+2$ cycles where p is the length of the input stream and q is the number of matches. The two extra cycles are required to initialize the PE array and to process the distance flags from the delimiter column.

2.8 Architecture Extensions

2.8.1 KMP with no delimiters

As discussed above, there may be cases where the input data stream may not have any obvious delimiters. In such a case, we introduce a row of n registers parallel to the CAM array. The register stores a '1' if it is in the column of the last character of a keyword, otherwise it stores a '0'. These registers can be initialized when the CAM is initialized. The PE array initialization and distance match calculation processes can thus use the register values to determine keyword ends rather than the match signal from a delimiter column. During flag propagation, we must be careful not to propagate flags past the keyword end into the next keyword.

2.8.2 Subsequence keyword match

The architecture as presented can only compare the entire input stream against the keyword set. However, there may be applications when the input stream is much longer than any keyword, and thus the goal is to search for matches against subsequences of the input stream. The restatement of the keyword match problem is then to determine if any subsequence in T matches any keyword $k \in K$. The algorithm to handle this new problem is similar to before except now we can potentially have keywords starting at any point of the input stream. This change simply requires that we introduce a flag at the beginning of each keyword on every clock cycle, not just the first clock cycle.

3. KMP VLSI IMPLEMENTATION

The KMP was designed using the AMIS 0.5 micron 2-metal layer process. A 64 character KMP was implemented using roughly 23000 transistors. The CAM array takes 4608

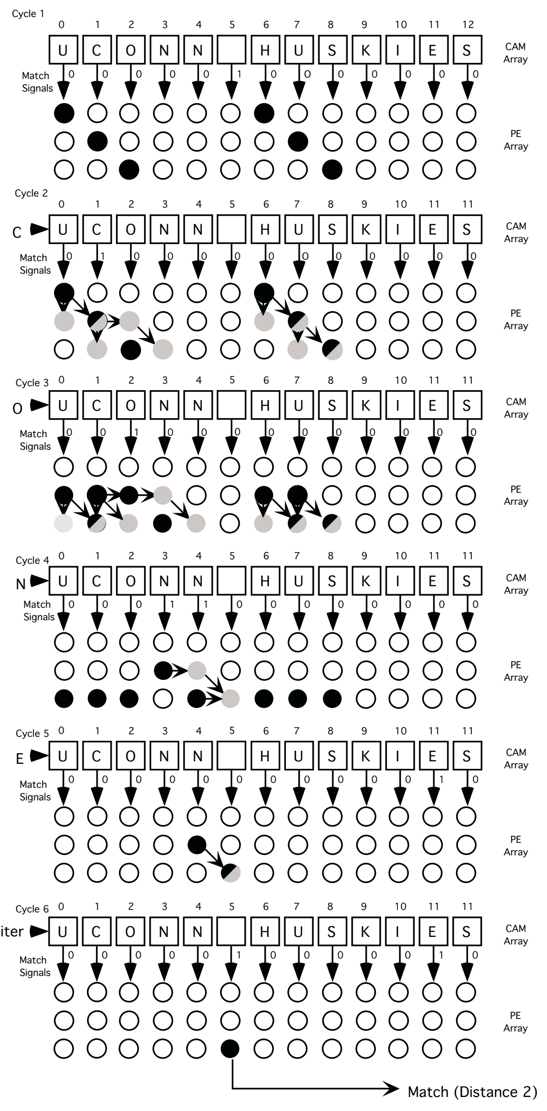

Figure 4: Keyword Match Processor Example.

transistors (9 transistors per CAM bit cell, 8 bits per character, 64 characters) and the PE array uses 7296 transistors (38 transistors per PE, 64 PEs per row, 3 rows). The design consumes about 250 transistors per character. Thus, a 32K character KMP would use roughly 8 million transistors.

Using the Cadence Analog Environment, we measured delays for various components (see Table 2). The CAM and PE delays are relatively short compared to the WM and MP delays. The distance decoder and matched position finder operations must happen within a single clock cycle, so t_{DD} and t_{MPF} are the primary factors influencing maximum clock speed. In other words, $t_{CLK} > t_{DD} + t_{DD2MPF} + t_{MPF} \Rightarrow t_{CLK} > 7.66ns$. A clock period of 7.66 ns leads to a maximum clock frequency of 130.5 MHz. Taking into account the setup times and propagation times of the registers, we have been able to comfortably run the circuit at 100MHz. That allows the KMP to process data streams at 100 million characters per second, or 800 Mb/s. By scaling to a 0.1 micron process, we believe that we could improve the operating frequency to near 500MHz thereby providing a data stream

Signal	WM(58)	58	57	56	55	54	53	52	51	50
Simulation time(ns)	0	0.699	1.531	2.150	2.653	3.113	3.430	3.690	3.874	4.082
Delay for each column (ns)		0.699	0.832	0.619	0.503	0.460	0.317	0.260	0.184	0.208

Table 3: MPF word match propagation delays

	Description	Delay
t_{CAM}	Delay from input to match signal	1.14 ns
t_{PE}	Delay for flag propagation	0.85 ns
t_{DD}	Delay to WM and distance output	3.43 ns
t_{DD2MPF}	Delay from DD to MPF blocks	0.15 ns
t_{MPF}	Delay to propagate matched position (8 position shift)	4.08 ns
t_{MP2MAA}	Delay from MP to MAA signal	1.69 ns
t_{MP2LP}	Delay from MP to LP signal	1.37 ns

Table 2: KMP Delays

operating rate of near 4 Gb/s. This data rate is fast enough to process data streams at multi-gigabit network rates.

The delays presented in Table 2 are for a 64 character KMP. For a practical implementation, the size of the KMP should be much larger. In the following discussion, we develop a delay model for larger KMPs. Because of the parallel nature of the processing, the t_{CAM}, t_{PE}, and t_{DD} delays are not dependent on the size of the KMP. However, because of their propagation nature, the t_{MPF} and t_{MAO} delays are dependent on size parameters.

The MPF delay is comprised mostly of the time to propagate the Word Match signal to the beginning of the keyword. Table 3 shows the time required to move the word match signal from column 58 to column 50. This delay is thus dependent on the longest keyword in the set. For an average keyword length of 5 characters, the delay is 3.43 ns. If we do not allow keywords longer than 8 characters, we can guarantee a maximum delay of 4.082 ns. Increasing the maximum length will add roughly .2 ns per character to the delay. The MPF delay is not dependent on the size of the array but only on the length of the keyword.

The MAO logic delay has two components and they are both dependent on the size of the array. The first is the delay from the `MP` signals to the `MAA` signal. Our measurements show a maximum delay of .226ns per level of the binary tree. With t_{MP2MAA} equal to 1.69 ns for a 6 level tree ($n = 64$), we can derive that $t_{MP2MAA} = .226\ log_2 n + .335$. The second component of the delay is the LP propagation back up the tree. The delay per level of the tree is 0.228 ns. With t_{MAA2LP} equal to 1.31 ns for a 6 level tree, we can derive that $t_{MAA2LP} = .228\ log_2 n$. We can generalize this and calculate the delay for the entire MAO tree: $t_d = t_{MP2MAA} + t_{MAA2LP} = .454\ log_2 n + .335$ ns.

For a 64 character KMP, the DD and MPF delays dominate the MAO delays. However as we increase the size of the array to over 1000 characters, the MAO delay becomes more significant. Instead of lengthening the clock cycle to accommodate the longer delays, it might make more sense to pipeline the MAO stage. Likewise, the DD and MPF stages can be pipelined as well to get a faster clock frequency.

4. CONCLUSIONS

We have designed and fabricated a VLSI implementation of a keyword match processor. The design extends upon cascaded CAM designs to do fast keyword matching on input data streams with arbitrarily sized keywords. The current implementation allows us to process data at 800Mb/s. The applications of such a KMP are numerous, but the most promising area is in network processing. We have used a modified KMP design to create a network intrusion detection system. The system can process incoming networks at over 800Mb/s. Current software based solutions can barely handle 100Mb/s before dropping packets. Other potential applications include directory lookups in network storage file systems and URL matching for load balancing.

5. REFERENCES

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–343, June 1975.

[2] K. Anagnostakis, S. Antonatos, E. P. Markatos, and M. Polychronakis. E2XB: A domain-specific string matching algorithm for intrustion detection. In *Proceedings of IFIP International Information Security Conference*, 2003.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[4] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In *Proceedings of DARPA Information Survivability Conference and Exposition II*, pages I:367–373, 2001.

[5] M. Fisk and G. Varghese. Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, Department of Computer Science, University of California, San Diego, May 2001.

[6] K. E. Grosspietsch. Associative processors and memories: A survey. *IEEE Micro*, 12(3), May/June 1992.

[7] M. Hirata, H. Yamada, H. Nagai, and K. Takahashi. A versatile data string-search VLSI. *IEEE Journal of Solid-State Circuits*, 23(2):329–335, Apr. 1988.

[8] A. J. McAuley and P. Francis. Fast routing table lookup using cams. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1382–1391, Mar. 1993.

[9] T. Moors and A. Cantoni. Cascading content addressable memories. *IEEE Micro*, 12(3):56–66, May/June 1992.

[10] M. Motomura, J. Toyoura, K. Hirata, H. Ooka, H. Yamada, and T. Enomoto. A 1.2-million transistor, 33 MHz, 20-b dictionary search processor (DISP) ULSI with a 160-kb CAM. *IEEE Journal of Solid-State Circuits*, 25(5):1158–1164, Oct. 1990.

[11] M. Motomura, H. Yamada, and T. Enomoto. A 2k-word dictionary search processor (DISP) with an approximate word search capability. *IEEE Journal of Solid-State Circuits*, 27(6):883–891, June 1992.

[12] A. Mukhopadhyay. Hardware algorithms for string processing. *IEEE Computer*, pages 508–511, 1980.

[13] R. Panigrahy and S. Sharma. Sorting and searching using ternary CAMs. *IEEE Micro*, 23(1):44–53, January/February 2003.

[14] J. P. Wade and C. G. Sodini. A ternary content addressable search engine. *IEEE Journal of Solid-State Circuits*, 24(4):1003–1013, Aug. 1989.