The AM³ Associative Processor

As an associative 32-bit processor based on a modified Harvard architecture, the Associative Microprogrammable Multipurpose Monoprocessor (AM³) uses a standard address-bus system to handle von Neumann data and instructions. A separate addressless associative bus system links it to associative memory components. Its microprogrammability enables the AM³ to adapt to a variety of associative memory circuits. After discussing the AM³ architecture and programming environment, we present two application paradigms.

Bernd Klauer

Andreas Bleck

Klaus Waldschmidt

J. W. Goethe University

As the current demand for real-time processing demonstrates, modern signal and data processing applications face increasingly stringent performance requirements. Only computer architectures that combine high arithmetic performance with flexible, fast decision-making operations can meet such challenges. Signal and data processing algorithms require matrix and vector operations as well as scalar computations to avoid slow, control-flow repeat structures. Many application programs, however, also need the versatility of standard von Neumann processors.

To balance such demands, the Associative Microprogrammable Multipurpose Monoprocessor (AM³) architecture we have developed can implement applications and algorithms based on neural networks, content-addressable memories—or both, while also providing sequential access to conventional memories. With its object-oriented C++ programming interface, the AM³ processor serves as a paradigm for associative processors. This architecture enables applications programmed in an object-oriented style to execute on the associative processor. As we will show, these features led us to the concept of virtual associative storage that supports manipulations of multiple objects in a single CAM.

AM³ processor

Our processor is a heterogeneous, monoprocessor computer architecture[1,2] that joins sequential, program counter-driven access to random access memory with parallel, content-addressed access to associative storage devices (CAMs). The AM³ can split data according to the most efficient processing type, putting it into RAM/ROM for sequential access or CAM for parallel access, thus providing a convenient method for parallelizing embedded loops that perform search operations. In particular, algorithms for pattern recognition, classification, and symbolic computations benefit from the performance of an associative memory. Decision-making operations in the AM³ may execute efficiently and combine easily with numeric computations.

Figure 1 introduces the basic architectural structure of the AM³ processor and its host. Based on a modified Harvard architecture having two bus systems, the design separates the bus for the random access program and data storage from the addressless associative (content-addressable) bus.[1]

Because its CPU is assembled from AMD 29xx bit-slice elements, the instruction set on the AM³ is adaptable and microprogrammable.[3] The need for flexible instruction sets arises both when using the AM³ design as a testbed for parallel associative algorithm research and for the programming tool kit. A dual-port RAM establishes an efficient hardware link between the AM³ processor and a host workstation.

Consisting of two coupled processors, the system offers two modes of operation: 1) During program design, the host workstation serves as

0272-1732/95/$04.00 © 1995 IEEE

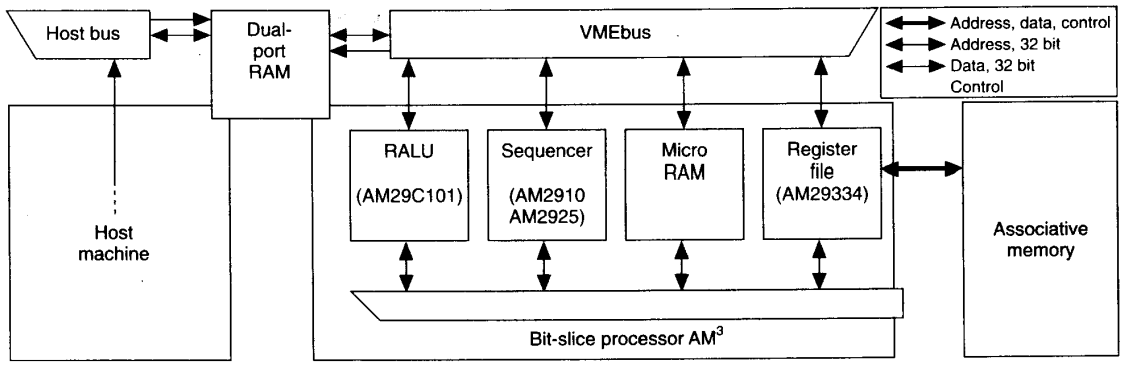


Figure 1. Architecture of the AM³ associative universal processor, its host workstation, and the associative memory. Multiple associative memories and concepts for associative memories are available for the AM³.

a platform for all the development tools and as a monitoring and mass-storage device; 2) after completing the application development phase, users can disconnect the associative processor for use as a stand-alone processor.

As Figure 2 shows, the primary use for the AM³ architectural concept and algorithms discussed here is in applications that combine explicit parallelism with strictly sequential parts. Migrating parallelizable parts of search algorithms in such applications into the associative memory for parallel execution will significantly reduce program execution time. Also, by combining a parallel associative part and a standard RAM with conventional address-driven access in a modified Harvard monoprocessor architecture, the AM³ minimizes communication and administration overhead.

Another important feature of associative processing is the selection mechanism used. After the selected sequential computations (numeric computation, decision operation) execute on a reduced (relevant) data set, this mechanism provides operations with operands by isolating specific data sets from the memory contents. This special feature can also operate in a matching unit of dataflow computers to retrieve executable instructions from an instruction store. In this case, the associative memory must reduce the set of all instructions to the subset of executable instructions: It must select all instructions having all operands available.

Associative memory components. Many recent publications have proposed associative memories that provide a variety of associative functions. Associative memories fall into two subclasses: neural networks and memories with local storage. Even if it has a neural interface link,[4] the AM³ processor design best suits memories with local storage. Special features of a memory implementation require adaptations to the processor's instruction set. Because they have been investigated within the development of the AM³ processor, the memories and concepts for associative memories we discuss next can all replace the associative memory box in Figure 1.

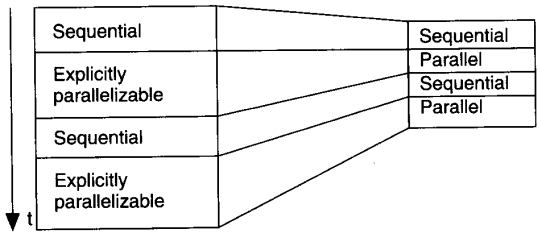


Figure 2. Sequential structure of a program and its parallel execution scheme on the associative processor.

Associative random access memory. ARAM is based on flag algebra.[5] As Figure 3 (next page) shows, flags indicate the existence of data words in a memory vector. The flag algebra reduces powerful relational operations to 1-bit manipulations that proceed amazingly quickly with constant-time complexity. The access time of an ARAM module does not grow with the sum of stored data words.

Its exponential space complexity is a drawback for the ARAM: storing n-bit data words requires 2^n flags (bits). Designs can cascade multiple ARAM modules. ARAM supports the masked search function, search above limit (find all x with $x >$ limit), search below limit (find all x with $x <$ limit), search between limit (find all x with lolimit $< x <$ hilimit), and search on a previously selected subset. For testing and to support a mixed parallel/serial approach for large-capacity associative memories, designers our group at Technische Informatik has implemented ARAMs in VLSI with a storage

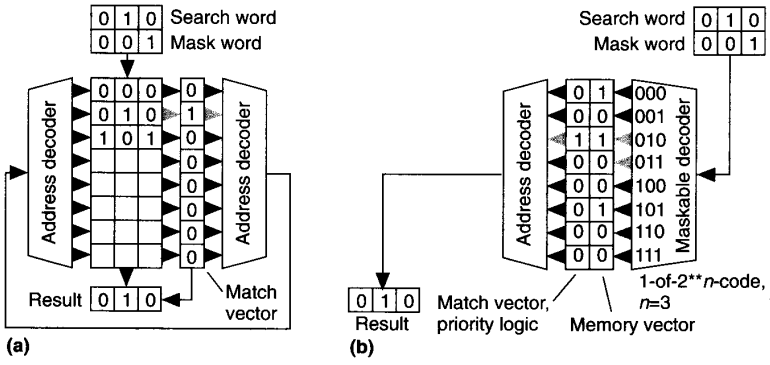


Figure 3. A word-oriented CAM (a) and the ARAM architecture (b).

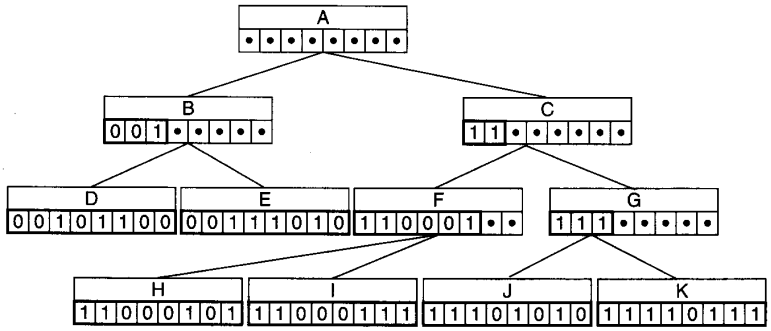


Figure 4. Data structure of the Patricia tree memory.

capacity of up to 64×6 bits per chip (cascadable).

Modified binary trees. As Figure 4 shows, designers have implemented a modified binary tree structure (practical algorithm to retrieve information coded in alphanumeric—Patricia—tree[6]) on the microprogram level of the AMD 29xx processor. The microprogrammed implementation of the Patricia tree structure was the first large-capacity implementation of an associative memory for the AM³.

Based on standard, off-the-shelf static memory and expandable to 64 Mbytes, this implementation supports all the associative functions of the ARAM with linear-space complexity. Later revised to decrease its access time, it also features linear worst-case and logarithmic best-case access time complexity. Supporting the retrieval mechanism of the tree architecture with fully parallel ARAM circuits has provided significant speedup.

Content hybrid addressable memory. The CHAM concept combines ARAM's fast access time with large storage capacity, while also joining the concepts of the tree memory and ARAM. Basically, CHAM is a wafer-scale integration of the ARAM concept. Therefore, fault-tolerance concerns required a revision to the ARAM concept.[7] Splitting the memory into a statically organized CAM (SOCAM) and a dynamically organized CAM (DOCAM) provides the fault tolerance. (Static and dynamic here describe memory-management features, not storage cells.)

DOCAM cells are much more spatially complex than SOCAM cells. The DOCAM module allocates DOCAM cells only if the embedded test logic has labeled them fault free. The CHAM memory logic then can allocate DOCAM cells from the DOCAM module to replace dirty SOCAM cells.

Masked Hamming distance memory. As Figure 5 shows, we derived the MHD memory device from a special pattern-recognition method (discussed later) used originally on the AM³. After proving the method suitable for support by associative hardware, we implemented it as a special memory device.[8] This device uses the MHD measure to compare patterns stored in the memory cells to an unknown pattern. The best of all (BOA) unit computes the minimum of all distances. For cases where there are multiple patterns with the same minimum distance to a prototype, the design includes a multimatch resolver. The encoder computes the address of the best match in the database.

AM³ instruction sets. The AM³ processor does not come with just one static instruction set. Rather, specific requirements of the associative memory's interface and problem-specific requirements can dictate the instruction set's configuration. Underlying the instruction set architecture is the modified Harvard architecture shown in Figure 1 that offers two types of operations: the basic instruction set, which is restricted to von Neumann-type instructions and addressing modes and thus completely independent of the type of CAM applied; and the associative instruction set architecture, with its operands defined for the mechanisms of associative addressing including problem-specific instructions.

Basic instruction set. AMD's 29xx bipolar bit-slice elements form the basis for the AM³ processor. The processor has an orthogonal basic instruction set containing the standard instruc-

tions for logical and numerical computations plus instructions for flow control, I/O, and memory access.[9] A 64-element, dual-port register file interfaces the parallel, addressless associative memories with the sequential, address-related control processor.

Associative instruction set. The most important functions of associative devices are search and relational operations, which work directly in parallel programming. The AM³ lets users program instruction sets to support the associative operations directly. They can also implement additional memory-management functions for initialization, allocation, disposal, and the dataflow between the associative memories and the processor.

A (microprogram) procedure to perform an associative operation consists of two phases:

- *Selection*—We have designed a set of machine instructions to create subsets of matches in the associative memory units. Users must encode the search function and the search arguments to be used as opcode and operands.
- *Match separation and processing*—The AM³ provides instructions to separate single matches from a selected subset. These instructions convert matches from set type into a scalar representation. Users can then apply scalar operations from the basic instruction set to the separated matches. The associative instruction set is tightly related to the functions performed by the memories used within the AM³. Users can microprogram the instruction set for other memories and to support specific applications.

Instruction set implementation. For demonstration purposes, we have implemented an instruction set on the AM³ similar to the MC68000's instruction set. This demonstration required amendments to support specific features of different associative memories and to provide test algorithms with specific support. We used this instruction set to test and develop an assembly language and high-level language programming interface.

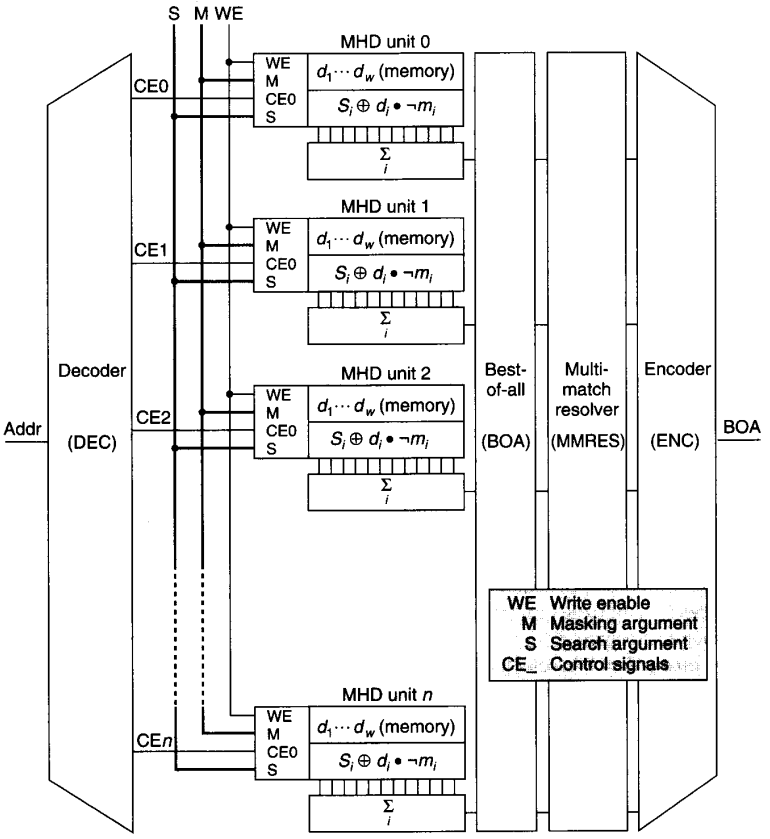


Figure 5. MHD memory architecture.

Assembly language, high-level programming

The assembly language and high-level language programming interface is also important in the design of new hardware concepts. Even with the opportunity for parallel data processing it provides, a system like the AM³ will only find acceptance if it includes appropriate programming tools. The application programmer must have a comfortable programming interface. As described earlier, we have developed programming interfaces based on the modified MC68000 instruction set. These include a micro assembler; assembler; and standard high-level language (C++) extended by an AM³ specific class library.

The micro assembler lets users create custom instruction sets for the AM³. They will use the assembly language interface only during system design and as a user-transparent interface for the compiler. The object-oriented library interface implements easily and flexibly. Application program-

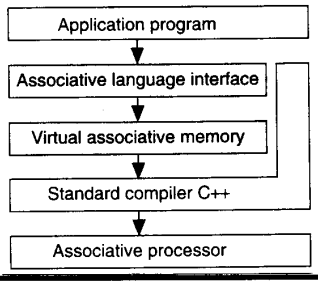


Figure 6. High-level programming interface.

mers need not be familiar with the specific hardware. Nor need they rewrite existing C++ sources; they can optimize them by applying the associative base classes.

We have also considered using special languages or dialects for the programming interface. Such special dialects would make the associative hardware features transparent for the programmer, but would also require considerable compiler design experience and would degrade portability. Because standard applications would need rewriting, using such special languages might undermine application programmer acceptance.[10]

C++ compiler. We selected the C++ programming language[11] to build the programming language interface to the AM^3 associative processor. This programming language is not only available for the host workstation to generate cross code, but also for numerous other hardware platforms and their operating systems.

The GNU-C and GNU-C++ compilers are portable, adaptable translators that are available free of charge. Designed for adaptability to arbitrary processors, the GNU compiler set uses a machine description of the destination processor for code generation. (GNU is a project of the Free Software Foundation, designed to provide free Unix software. GNU stands for "not Unix.") Figure 3 presented the software levels used to program the associative hardware. Besides the definition of the language interface, the object library handles memory management.

The administration functions to be performed closely resemble the functions done by virtual memory in RAMs. Introducing a virtual interface layer between the hardware and its application extends the functionality of the associative storage. This holds true for the limitations of the associative address space vertically and horizontally and for a logical separation of distinct objects. Both parts of the user-interface layer are shaded gray in Figure 6.

Associative objects in C++. Typical associative operations are set operations. Standard programming models paradigm sets by lists or arrays, referencing set members directly by their addresses or indirectly by their neighborhood relationships. To implement associative objects, we have created the base class "association," giving it an array-like structure.

The overloading of the operator is the key to the content-related addressing of formerly address-related structures. The operator usually consumes an index and returns the corresponding object. The overloaded version of the operator consumes a pointer to a data structure containing the search arguments instead of an index. The matching element in the associative memory then evaluates the search arguments to retrieve the required data. The return value is a pointer to an array of all matching objects.

Virtual associative memory. Associative memories still suffer from strict size limitations. Due to the complex hardware involved, these limits are much more restrictive than in conventional RAMs. Besides the number of cells, CAMs show another restriction. An object stored in the main memory of a computer usually distributes into consecutively addressed memory cells. Fixed addresses or simple address computations can serve to calculate their locations. In an addressless storage device, distinct language objects, such as variables, can no longer be distinguished by their location. We have postulated the following requirements for a virtual associative memory management mechanism: 1) Multiple objects of the application reside in a single associative memory simultaneously. 2) Stored wordsextendible without losing parallel-operation efficiency. 3) Objects swapable to main or peripheral storage.

As with the contents of a conventional memory, which are described by an address, data-tuple, we can create a logical address for associative memory by using indices. We divide the stored word into an index, descriptor-tuple. As Figure 7 shows, the index facilitates the access to atomic descriptors with a division into three subindices:

Object	Distinguishes distinct objects of the programming language,
Word	Indicates a word within one object, and
Partition	Splits a word to be stored into smaller pieces that fit into the remaining descriptor.

The following example of an identity search demonstrates the expense involved in a search operation. Every partial word fed into the associative memory will reveal a separate set of matches. The memory must search these sets of matches for identical word indices; an identity hit will be available only if the number of identical indices equals the number of descriptor partitions. We can easily do this kind of algorithm in hardware.

The associative memory must perform a shift on the match vector and a parallel increment operation, then join the resulting match vector with the next associative search. The expense of the search operation thus increases by the number of descriptor partitions.

Applications

So far, we have implemented two algorithms from two totally different problems on the AM³ processor: associatively computable methods to solve a classification problem and a graph-theoretical problem. The first application was a recognition algorithm for handprinted characters. The second was the associatively accelerated implementation of Dijkstra's algorithm to find the shortest path in a graph.

Aquire recognition engine for handprinted characters. Programmed in C++, this engine served originally as a pure software recognizer for pen-based application on mobile personal computers. We mapped it onto the AM³ architecture by replacing the main recognition module (Pattma) with a similar module that directly uses the associative features of the AM³. As Figure 8 shows, the recognizer itself consists of three modules: a preprocessor (Prep), pattern matcher (Pattma), and ambiguity resolver (Magic—manual gesture inconsistency clearance).

The major task of Prep is to provide the Pattma and Magic modules with significant information on the symbol to be recognized. Working on line during the drawing phase, Prep collects data from the digitizer (in this case, a touch panel for fingerprinted input). It converts the string of coordinates received from the digitizer into a Boolean pattern for the Pattma pattern matcher. It also extracts further context information for the Magic module. Though it is a very fast preclassifier, Pattma has insufficient recognition accuracy for optically similar characters such as B and 8 or b and 6. The Magic module detects and resolves these ambiguities.

See the box (next page) for a description of the MHD method used within the Aquire recognition engine.

AM³ implementation of Aquire. We have not directly implemented the Prep module on the AM³, but have programmed it on a special microcontroller. The microcontroller also provides the hardware interface to the touch-panel digitizer. The Magic module (ambiguity resolver) cannot take advantage of the associative features of the AM³ so it runs completely in the nonassociative parts of the AM³. The pattern matcher is perfectly suitable to run with associative hardware support.

We mapped Pattma onto the AM³ architecture. One function supported by all memory components within the AM³ is the masked search operation (MSO), which Kohonen[12] described as one of the basic associative functions. To map the character recognition method this article proposes onto the AM³ processor, we replaced the CAM module implemented as a virtual base class in C++ with a compatible module that directly uses the associative functions

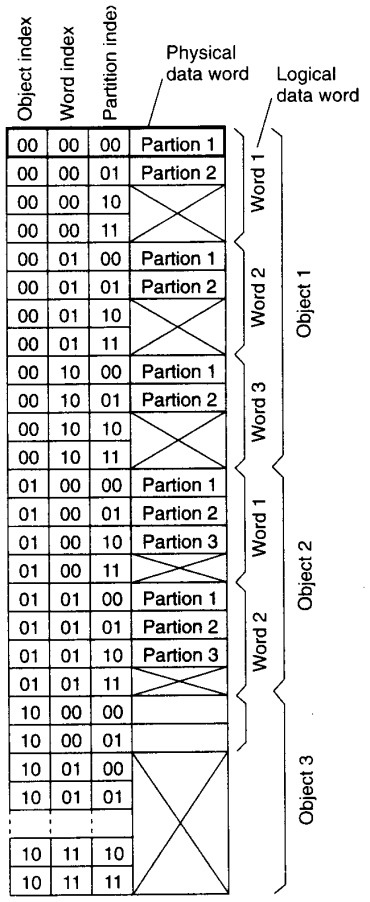


Figure 7. Indexed associative memory.

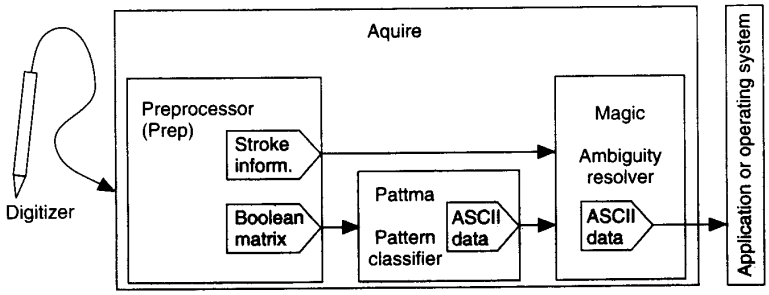


Figure 8. Software architecture of the Aquire recognition engine.

The MHD method revisited

Originally introduced by Klauer and Waldschmidt,[1] the MHD function can indicate the similarity of two Boolean patterns. Proven suitable for handprinted character recognition, it functions within the recognition engine (PenStar HWR) distributed with the INFOS 386-SX notepad computer.[2] (INFOS is German for personal digital assistant.)

Hamming distance. Hamming[3] originally proposed a distance of two patterns as an error correction mechanism for binary codes. In the following equations, let **A** and **B** be Boolean vectors. HD(**A**,**B**) is the total of all bits a_i, b_i with $a_i \neq b_i$. Within a discussion of similarity measures, Kohonen describes it to be "perhaps the best known measure between digital representations."[4]

$$HD(\mathbf{A},\mathbf{B}) = \sum (a_i \oplus b_i)$$

where $\oplus$ is the Boolean EXOR function

Masked Hamming distance. We define the MHD of two patterns **A** and **B** and a mask **M** as follows:

$$MHD(\mathbf{A},\mathbf{B},\mathbf{M}) = \sum ((a_i \oplus b_i) \wedge \neg m_i)$$

where $\wedge$ is the Boolean AND function and $\neg$ is the Boolean NOT function

MHD behaves as HD if $\mathbf{M} = \underline{0}$. In this case, MHD computes the total of all different elements a_i, b_i with $a_i \neq b_i$. These elements are not hidden by m_i. HD as well as MHD can serve as similarity indicators in recognition systems. We have shown[1] that MHD is much more suitable for the recognition of handprinted characters than HD.

MHD method. Let an image $I = (BP,s)$ be a tuple consisting of a Boolean pattern BP and the semantics s of the pattern. We can represent all well-known symbols in a database as the following example shows. ϕ indicates that the semantics of a symbol is unknown. We can use the following method to find the semantics of an unknown image $U = (X, \phi)$: 1) Compute the mask argument **M**; 2) compute MHD(X, P_i, **M**) for all i; and 3) replace ϕ with s_i if MHD (X, P_i, **M**) is minimal for all i.

Masking argument for the MHD method. The MHD method differs from the pure HD method simply because the masking argument **M** can hide specific areas of the patterns to be compared. The MHD classifier computes the Hamming distance of all unmasked bits, a useful feature in fault-tolerant classification. We have demonstrated an efficient method to compute the mask argument[1] and have further proven its suitability for character recognition. We are currently testing it for speech recognition.

Figure A1 shows two symbols that are similar but not equal. A recognition system holding one of both symbols in its database should classify them to be similar. The Hamming distance between patterns **A** and **B** is 28, indicating that the patterns are different—even if they are not from a human point of view.

of the AM³. (A virtual base class is an incomplete base class containing at least one function that has been declared but not implemented.) We also scaled all components of the AM³ to 32-bit data.

Therefore, the patterns of all images in the database and the patterns of images to be recognized must be disassembled into fragments. Let $I = (Y,s)$ be an image. A triple $f = (y_j,j,s)$ consisting of a subimage y_j, the index j of the subimage, and the semantics s of the corresponding image is then a fragment from I. The CAM must store all fragments from all known images.

Assume $u = (X, \phi)$ to be an unknown image. Let **Z** be a vector containing one counter for each known image. Each element of **Z** must be initialized by the total F of all fragments. Let Ω be the set of data in the associative memory, and let T be the set of all identified fragments after an MSO request. We can use the following informal procedure to determine the most similar pattern in the database contained in the associative memory. For all j

$$T = \text{MSO}\{(x_j,j,\phi), (\underline{m},\underline{0},\underline{1}), \Omega\}$$

for all f in T decrement ($Z[s]$).

(x_j,j,ϕ) is a fragment in the unknown image. $(\underline{m},\underline{0},\underline{1})$ is the mask argument. The m at the first place of the mask contains the mask for the pattern information. $\underline{m}$=0 at the first place indicates that the complete pattern-related information is significant. The index is also important, as indicated by the 0 in the second position. Only the semantics ϕ of the image to be identified is masked by a 1 because it is unknown.

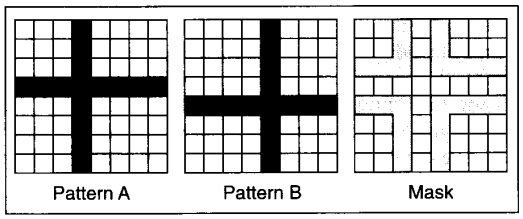
The semantics s must complete the triples in the returned set. After identifying all matching fragments, this operation uses the semantics to indicate which counters to decrement. Fragments in the memory can only match if they have the same index j as the search argument and matching data in the image part. Finally, Z contains MHD between all patterns in the database and the search and mask arguments if the size of the pattern related information is 1 bit per fragment.

AM³ performance parameters for the MHD pattern recognition method. One goal of the AM³ implementation was to show that the proposed method is suitable for computation and acceleration by associative hardware. The academic intention was to show a sublinear runtime behavior with a growing database size. The memories used within the AM³ enabled us to measure a logarithmic runtime behavior.

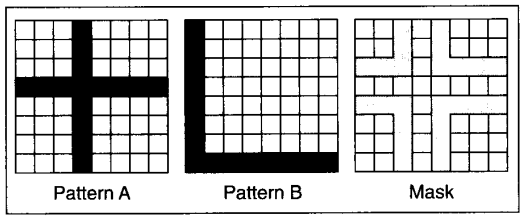
The MHD method revisited (continued)

The masked Hamming distance between **A** and **B** and a mask argument as shown in Figure 5 is 13, indicating that both patterns are similar. This result is more adequate to the human observer. The next example (Figure A2) shows that the MHD classifier also works when both symbols are different. In this case, MHD(**A,C,M**) = 24, indicating that both symbols are different.

Implementing the MHD method. We have implemented this MHD pattern recognition method as the core method of the Aquire recognition engine for handprinted characters on personal computers and the AM³ processor. Using the experience gained especially from the AM³ implementation, we have developed a special, fully parallel coprocessor called the MHD memory[5] to replace the complete Pattma module with a specific hardware component. We have also implemented a fast emulation of the MHD memory using a TMS 32C031 RISC processor. Implementing the MHD memory as a fully parallel hardware circuit remains a future task.



(1)



(2)

Figure A. Pattern comparison example: symbols similar but not equal (1); symbols quite different (2).

References

1. B. Klauer and K. Waldschmidt, "Pen-Based Recognizing of Handprinted Characters," *Proc. Euromicro*, Vol. 38, Elsevier, Amsterdam, 1993, pp. 803-809.
2. B. Klauer and K. Waldschmidt, "An Object-Oriented Character Recognition Engine," *Proc. Euro-ARCH*, Springer, Berlin, 1993, pp. 187-198.
3. R.W. Hamming, *Bell Systems Tech. J.*, Vol. 29, No. 147, 1950.
4. T. Kohonen, *Content-Addressable Memories*, Springer, Berlin, 1987.
5. B. Klauer, "The MHD Memory," *Proc. 15th DAGM Symp.* Mustererkennung, Springer, Berlin, 1993.

Single-source problems. Many applications require efficient computation of the shortest path from a vertex in a weighted graph to another vertex. The best known algorithm for that problem is Dijkstra's,[13] which computes the best paths from a single vertex to all other vertices. (Other than the single-source algorithm, no known algorithm computes only the best path between two vertices.) For example, intelligent routing systems for information networks or vehicle guidance systems require computation of the shortest or cheapest path.

Dijkstra's algorithm. Let $G = (V, E)$ be a directed graph, $v_0 \in V$ a source in G, and l a function from edges to nonnegative floating-point numbers. Let $l(v_i, v_j)$ be $+\infty$ if (v_i, v_j) is not an edge for $v_i \neq v_j$, and if $v_i = v_j$, $l(v_i, v_j) = 0$. The algorithm is

```
    BEGIN
1.      S := {v0};
2.      D[v0] := 0;
3.      FOR EACH v in V-{v0}DO D[v] := l(v0, v);
4.      WHILE S ≠ V DO
        BEGIN
5.          choose a vertex w in V-S such that D[w] is a
                minimum;
6.          add w to S;
7.          FOR EACH v in V-S DO
8.              D[v] := min( D[v], D[w] + l(w,v) )
        END
    END
```

AM³ implementation. To implement Dijkstra's algorithm, we chose an AM³ processor configuration with a large associative memory based on a tree structure. The data in this specific memory is always sorted. The processor can directly access minimum and maximum elements in constant time. We implemented the method just described on the AM³ processor using this special feature to accelerate the minimum search in line 5 of the algorithm shown.

To demonstrate the application, we embedded the complete system into a simulated vehicle guidance system and included the character recognition system for a comfortable user interface. The implementation of Dijkstra's algorithm lets us find the best path on a road map, taking various road conditions as weights on the graph (weather, accidents,

building sites). A human driver can issue data and control inputs by painting symbols on a touch-panel sensor device.

To understand the complexity and associative acceleration, let n be the total of vertices in the directed graph. Dijkstra's algorithm computes the shortest path from a source vertex to all other vertices in $2n^2 + n$ steps (worst case). Using the associative "find minimum" computation provided, for example, by ARAM memories, we can reduce the selection in line 5 to constant complexity. The total complexity then reduces to $n^2 + n$.

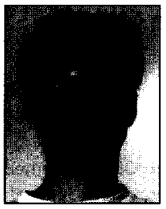
ALTHOUGH WE HAVE SEEN tremendous work on parallel computer architectures in recent years, parallel hardware components are still restricted to special classes of problems. As a parallel machine of the single-instruction, multiple-data type, the AM³ processor architecture combines special-purpose associative components with an all-purpose machine.

In general, the AM³ can compute all algorithms sequentially. It provides parallel support for searching and sorting procedures appearing in a variety of algorithms. As we have shown, the AM³ architecture can map special algorithms for pattern matching and shortest path computations with intensive searching and sorting.

References

1. M. Schulz et al., "An Associative Microprogrammable Bit-Slice Processor for Sensor Control," *Proc. Third CompEuro*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 87-95.
2. M. Darianian, C. Schœnfeld, and K. Waldschmidt, "A High-Capacity Associative Memory Array in the AM³," *Proc. ITG/GI Symp. Computer Architecture*, [in German], IEEE, Berlin, 1990.
3. J. Mick and J. Brick, *Bit-Slice Microprocessor Design*, McGraw-Hill, New York, 1980.
4. R. Schuster, *Interfacing of Neural Nets and von Neumann Architectures*, masters thesis, [in German], J.W. Goethe University, Frankfurt am Main, Germany, 1991.
5. D. Tavangarian, *Associative Memories and Processors Based on Flag Algebra*, [in German], Springer, Berlin, 1987.
6. D.R. Morrison, "Patricia—Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Trans. ACM*, Vol. 15, No. 4, Association of Computing Machinery, New York, 1968.
7. M. Darianian and K. Waldschmidt, "A Fault-Tolerant and Easily Testable Associative Memory with Optimized Storage," *Proc. EIS (Development of Integrated Circuit) Workshop*, [in German], Dresden, Germany, 1991.
8. B. Klauer, "The MHD Memory," *Proc. 15th DAGM Symp. Mustererkennung*, Springer, Berlin, 1993, pp. 568-575.
9. A. Bleck, "Design of an Instruction Set for a Microprogrammable Associative Processor," [in German], master's thesis, J.W. Goethe University, Frankfurt am Main, Germany, 1990.
10. J.L. Potter, *Associative Computing: A Programming Paradigm for Massively Parallel Computers*, Plenum Press, New York, 1992.
11. J.T. Schwartz et al., *Programming with Sets: An Introduction to SETL*, Springer, Berlin, 1986.
12. T. Kohonen, *Content-Addressable Memories*, Springer, Berlin, 1987.
13. E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik (Numerical Mathematics)*, Vol. 1, 1959, pp. 269-271.

Bernd Klauer is a research assistant at the Technical Information Department at the J.W. Goethe University in Frankfurt. His main fields of research are the implementation and application of associative systems. He received an MSc in computer science from J.W. Goethe University. He is a member of the German National Association for Computer Sciences (GI) and the German Electrical Engineering Association (VDE).



Andreas Bleck is a research assistant at the Technical Information Department at the J.W. Goethe University. His main field of research is the VLSI implementation of associative hardware components. He received an MSc in computer science from J.W. Goethe University.

Klaus Waldschmidt heads the Technical Information Department at the J.W. Goethe University. His research and teaching interests include computer architecture, especially associative memories and processors, and CAD, especially of mixed analog/digital circuits. He received the Dipl Ing and Dr-Ing degrees in electrical engineering from the Technical University of Berlin. He is a member of Society for Information Technologies (ITG), GI, and Euromicro.

Direct questions concerning this article to Bernd Klauer, J.W. Goethe University, Professur für Technische Informatik, Robert Mayer Strasse 11-15, D-60054, Frankfut Am Main, Germany; klauer@ti.informatik.uni-frankfurt.de.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 168 Medium 169 High 170