Guest Editor's Introduction

New Computers for Artificial Intelligence Processing

Benjamin W. Wah
University of Illinois at Urbana-Champaign

New computer architectures can be used to improve efficiency in the processing of some time-consuming AI tasks, but cannot overcome the combinatorial complexity of AI processing.

This special issue of *Computer* is about recent efforts to produce hardware and software architectures that support artificial intelligence (AI) applications. The earliest computer design to support AI processing was the implementation of Lisp on the PDP-6 computer and its successors, the PDP-10 and PDP-20, all made by the Digital Equipment Corporation. The half-word instructions and the stack instructions of these machines were developed with Lisp's requirements in mind. Since then, we have been seeing a proliferation of special-purpose computers that can support symbolic and AI processing. These efforts include the implementation in hardware of primitive operations fundamental to applications in AI, the design of a collection of primitive architectures to support more complex functions, and the design of system-level architectures to support one or more languages or knowledge-representation schemes.

Characteristics of AI computations

To develop a special-purpose computer to support AI processing, the requirements of AI applications must be fully understood. Conventional numerical algorithms are usually well analyzed, and bounds on computational performance can be established. In contrast, many AI applications are characterized by symbolic processing, nondeterministic computations, dynamic execution, large potential for parallel and distributed processing, knowledge management, and open systems.

Symbolic processing. AI applications generally process data in symbolic form. Primitive symbolic operations, such as comparison, selection, sorting, matching, logic set operations (union, intersection, and negation), contexts and partitions, transitive closure, and pattern retrieval and recognition are frequently used. At a higher level, symbolic operations on patterns such as sentences, speech, graphics, and images may be needed.

Nondeterministic computations. Many AI algorithms are *nondeterministic*, that is, it is impossible to plan with the available information the procedures that must be executed and terminated. This comes from a lack of knowledge and from an incomplete understanding of the problem, and results in the need to enumerate all possibilities exhaustively when the problem is solved.

0018-9162/87/0100-0010$01.00 © 1987 IEEE

Dynamic execution. Owing to this lack of complete knowledge and to the uncertainty of the solution process, the capabilities and features of existing data structures and functions may be defined, and new data structures and functions may be created, when the problem is actually solved. Further, the maximum size for a given structure may be so large that it is impossible to allocate the necessary memory space ahead of time. As a result, memory space may have to be dynamically allocated and deallocated when the problem is solved.

Large potential for parallel and distributed computing. In parallel processing of deterministic algorithms, a set of necessary and independent tasks must be found and processed concurrently. This class of parallelism is called *AND- parallelism*. In AI processing, the large degree of nondeterminism offers an additional source of parallel processing. Tasks at a nondeterministic decision point can be processed in parallel. This latter class is called *OR-parallelism*.

Knowledge management. Knowledge is an important component in reducing the complexity of solving a given problem: More useful knowledge means less exhaustive searching. However, many AI problems may have a very high degree of inherent complexity, hence the amount of useful knowledge may also be exceedingly large. Further, the knowledge acquired may be fuzzy, heuristic, and uncertain in nature. The management, representation, manipulation, and learning of knowledge are, therefore, important problems to be addressed.

Open systems. In many AI applications, the knowledge needed to solve the problem may be incomplete because the source of the knowledge is unknown at the time the solution is devised, or because the environment may be changing and cannot be anticipated at design time. AI systems should be designed with an open concept and allow continuous refinement and acquisition of new knowledge.

Design issues

The essential issues in designing a computer system to support a given AI application can be classified into the *representation level*, the *control level*, and the *processor level* (see Table 1).

- The representation level deals with the knowledge and methods to solve the problem, and the means to represent it.
- The control level is concerned with the detection of dependencies and parallelism in the algorithmic and program representations of the problem.
- At the processor level, the hardware and architectural components needed to evaluate the algorithmic and program representations are developed.

Many current designs start with a given language or knowledge-representation scheme; hence, the representation level is already fixed. Research has been focused on automatic methods to detect parallelism, as well as on providing hardware support for time-consuming operations. However, the representation level is an important element in the design process and dictates whether the given problem can be solved in a reasonable amount of time. At this time, little has been done in providing tools (a) to aid users in collecting and organizing knowledge or (b) to aid them in designing efficient algorithms.

Hierarchy of meta-knowledge. *Domain knowledge* refers to objects, events, and actions per se, while *meta-knowledge* includes the extent and origin of the domain knowledge of a particular object, the reliability of certain information, and the possibility that an event will occur. In other words, meta-knowledge is knowledge about domain knowledge. Meta-knowledge can be considered as existing in a hierarchy. That is, meta-knowledge is involved in deciding the appropriate domain knowledge to apply, while *meta-meta-knowledge* is the control knowledge about the meta-knowledge. Higher level meta-knowledge is commonsense knowledge known to humans.

The use of meta-knowledge allows one to express the partial specification of program behavior in a declarative language, hence making programs more aesthetic, simpler to build, and easier to modify. Moreover, it facilitates incremental system development; that is, one can start from a search-intensive algorithm and incrementally add control information until one obtains an algorithm that may be search-free. Lastly, many knowledge-representation schemes and program paradigms, such as logic, frame, semantic network, and object-oriented languages, can be integrated with the aid of meta-knowledge.

There are many open problems related to the use of meta-knowledge: its unambiguous specification, its consistency verification, the learning of new meta-knowledge, and the use of appropriate statistical metrics.

Table 1. Design issues in AI architectures.

Representation level
Hierarchy of meta-knowledge
Domain-knowledge representation
AI languages and programming
Control level
Truth maintenance
Partitioning and restructuring
Synchronization
Scheduling
Processor level
Micro-level architectures
Macro-level architectures
System-level architectures

Domain-knowledge representation. *Declarative representations* specify static knowledge, while *procedural representations* specify static knowledge as well as the control information that operates on the static knowledge.

Declarative representations are *referentially transparent*; that is, the meaning of a whole can be derived solely from the meaning of its parts and is independent of their historical behavior. Declarative representations offer a higher potential for parallelism than procedural representations, but are usually associated with a large search space that may partly counteract the gains of parallel processing.

In contrast, procedural schemes allow the specification and direct interaction of facts and heuristic information, hence eliminating wasteful searching. However, they may over-specify the precedence constraints and restrict the degree of parallel processing. When one chooses the appropriate representation scheme, trade-offs must be performed as regards the amount of memory space required to store the knowledge, the time allowed for making inferences, the expected usage of the knowledge, and the underlying computer architecture and technological limitations.

AI languages and programming. Conventional imperative languages are inade-

quate to AI programming owing to its inefficiency in symbolic and pattern processing and its unacceptable programming complexity. New AI languages feature large declarative power, symbolic processing constructs, representation of information by lists, and use of recursion as the only control mechanism. Function-, logic-, and object-oriented languages are the major programming paradigms used for AI today, and hybrids of these paradigms have been developed. These languages differ in their expressive power, their ease of implementation, their ability to specify parallelism, and their ability to include heuristic knowledge. A language-oriented AI computer will inherit all the features and limitations of the language it implements.

Truth maintenance. Many AI applications are characterized by a lack of consistent and complete knowledge at the representation level. Hence, it may be necessary to modify the existing knowledge base continually and to maintain its consistency as new knowledge is acquired. *Truth maintenance* consists of recognizing an inconsistency, modifying the state to remove the inconsistency, and verifying that all inconsistencies are detected and corrected properly. The process of removing inconsistencies may be inconsistent itself, and may introduce further inconsistencies into the knowledge base.

Partitioning and restructuring. These refer to the reorganization and decomposition of the knowledge base and the AI program to achieve more efficient processing. The issues that need to be considered are similar to those considered for conventional multiprocessing and parallel processing systems, namely, granularity, static and dynamic detection of parallelism, and restructuring. However, the methods to resolve these issues are different. Owing to the nondeterminism encountered in AI processing, AI tasks may be decomposed into a large number of smaller tasks, which will influence the design of a special-purpose computer system to support AI processing. Many of the proposed AI systems have a collection of simple processing elements to execute tasks of small granularity and another collection of more complex processing elements to execute tasks of larger granularity.

The detection of parallelism is also complicated by the nondirectionality of the modes of variables, the dynamic creation and destruction of data, and the nondeterminism. In many languages designed for AI processing, the input/output modes of variables and the necessary data structures are defined at run time. Static analysis, allocation, and scheduling are impossible here. Dynamic detection and scheduling do not give satisfactory performance because of their relatively high overheads for tasks of small granularity. One popular solution is to require users to supply additional information in order to allow compile-time analysis. The amount of speedup that parallel processing of nondeterministic tasks will provide is not clear, although the potential for processing these tasks in parallel is great. Without appropriately guiding the search, restructuring the program, and detecting redundant computations, much of the power of parallel processing may be directed toward fruitless searches.

Synchronization. There are two levels of synchronization: *control-level synchronization* and *data-level synchronization*.

In procedural languages, if a statement precedes another statement in the program, the implication is that this statement precedes the second statement if the two statements share common variables; that is, control-level synchronization is implicit when data-level synchronization is needed. This implicit execution order may overspecify the necessary control-level synchronization in the problem.

On the other hand, if the tasks are specified as a set over a number of declarative languages, then control-level synchronization is absent, and the set of tasks can be processed in parallel if the tasks do not share common variables.

If the tasks have common variables but are semantically independent, then they have to be processed sequentially in an arbitrary order to maintain data-level synchronization. The difficulty of specifying control-level synchronization when tasks are semantically dependent is a major problem in declarative languages, such as Prolog. For example, the decomposition of a set into two subsets in Quicksort must be performed before the subsets are sorted. Hence, the tasks for decomposition and for sorting the subsets are both semantically dependent and data dependent. To overcome this problem, programmers are provided with additional primitives, such as the input/output modes of variables in a Prolog program, to specify the necessary control-level synchronization. These primitives may have side effects and may not be able to specify completely all control-level synchronization in all situations. These problems may have to be dealt with at run time until sufficient information is available to solve them.

In short, there is a trade-off between the expressive power of the language and the implementation difficulty in designing a special-purpose computer system to support an AI language. New languages that combine the ability of functional languages to specify parallel tasks and that of logic languages to specify nondeterminism are evolving.

Synchronization is needed when the system is *message-based*, but may not be needed in systems that are *value-based* or *marker-based*. In a value-based system, multiple values arriving at a processor simultaneously are combined into a single value, hence contention will not happen and synchronization is not necessary. Neural networks and the Boltzmann machine are examples of this class. In systems supporting marker-passing, such as NETL and the Connection Machine, markers in a set represent entities with a common property and are identified in a single broadcast, hence synchronization is not necessary.

Scheduling. *Scheduling* is the selection of ready tasks to assign to available processors. It is especially important when there is nondeterminism in the algorithm. Scheduling can be static or dynamic. *Static scheduling* is performed before the tasks are executed, while *dynamic scheduling* is carried out when the tasks are executed.

The difficulty in designing a good scheduler lies in the heuristic metrics to guide the nondeterministic search. The metrics used must be efficient and accurate. Trade-offs must be made among the dynamic overhead incurred in communicating the heuristic-guiding and pruning information, the benefits that would be gained if this information led the search in the right direction, and the granularity of tasks.

In practice, the merits of heuristic guiding are not clear, since the heuristic information may be fallible. As a result, some AI architects do not schedule nondeterministic tasks in parallel. The excessive overhead coupled with the fallibility of heuristic information also leads some designers to apply only static scheduling to AI programs.

Micro-level, macro-level, and system-level architectures. The VLSI technology

that has flourished in the past 10 years has resulted in the development of many special-purpose computers.

• *Micro-level architectures* to support AI processing consist of architectural designs that are fundamental to applications in AI.

Examples of basic computational problems that are solved efficiently in VLSI are set intersection, transitive closure, contexts and partitions, best-match recognition, recognition under transformation, sorting, string and pattern matching, dynamic programming, selection, and proximity searches. Special features in AI languages that are overhead-intensive can also be supported by hardware. Examples of these architectures include the unification hardware, tag bits for dynamic data-type checking, and hardware stacks.

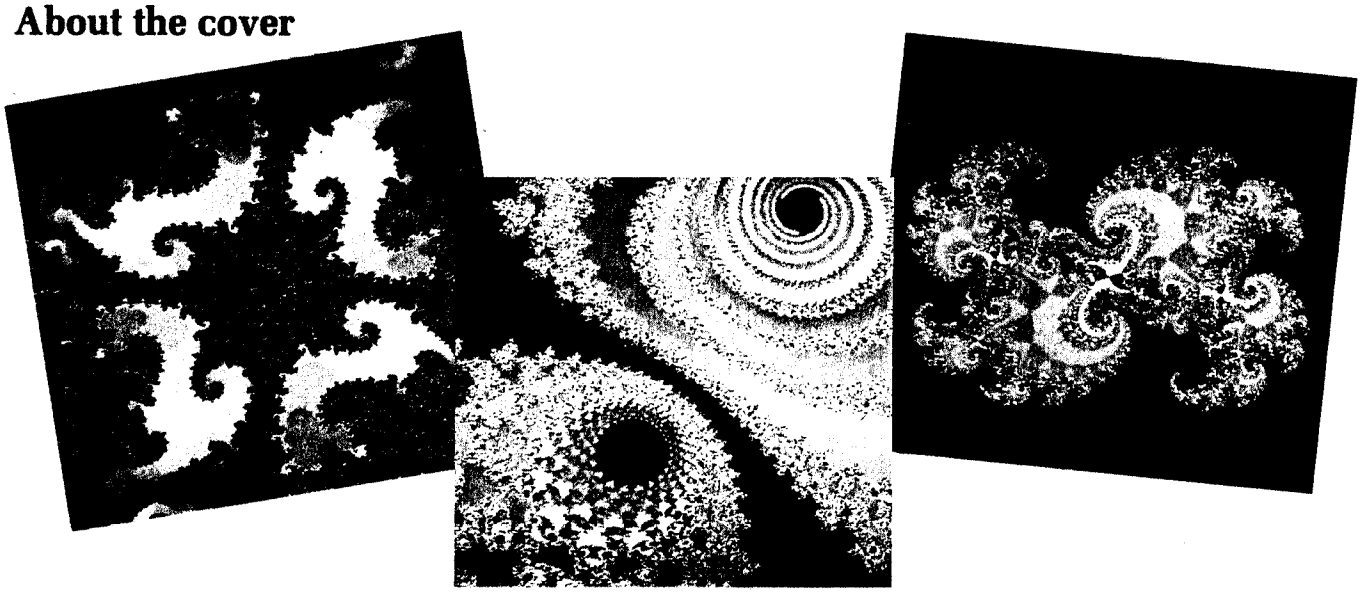
• The macro-level is an intermediate level between the micro-level and the system level. *Macro-level architectures* can be made up of a variety of micro-level architectures and can perform more complex operations. Examples include the dictionary and database machines, architectures for searching, and architectures for managing dynamic data structures (such as the garbage-collection hardware).

• The *system-level architectures* available today are generally oriented toward one or a few of the languages and knowledge-representation schemes and designed to provide architectural support for overhead-intensive features in these languages and schemes. Examples include systems to support functional programming languages, logic languages, object-oriented languages, production rules, semantic networks, and special applications, such as robotics and natural-language understanding.

Design methodology

The issues classified in Table 1 provide a view to the design methodology for special-purpose computers that support AI processing. The various approaches can be

About the cover



The explosion of form and color here and on the cover symbolizes the complexity of AI processing—its combinatorial complexity. The enumerative nature of many algorithms used in AI applications can lead to endless searching for the correct combinations. The key to harnessing this unbridled combinatorial stampede is to establish good heuristics and efficient computers.

The graphics were created by a graphics program of an iterative process in which the output of the polynomial $y = z^2 + c$ provides the input for the same equation, whose output in turn becomes input, and so on. The images, in an infinite regress, are made up of smaller and smaller clones of the parent image. The colorless areas of the images locate a set of numbers in the complex plane known in the field of fractal geometry as the Mandelbrot set—named after Benoit Mandelbrot of the IBM T. J. Watson Research Center.[1-4]

The images were generated at the Cornell National Supercomputer Facility in conjunction with the university's Mathematics Department and Theory Center. They were computed on Floating Point System's Models 164 or 264 Array Processors attached to an IBM 3090/400 Quad Processor mainframe running VM/370 Extended Architecture with 999 million bytes of virtual address space. They were displayed on an Advanced Electronics Design model 512 graphics tube. The AED 512 was attached to the IBM 3081 via a 9600-baud RS-232 port and a Device Attachment Control Unit 7170 high-speed communications line.

They were photographed with a Canon A1 Camera equipped with a Vivitar 70-150 zoom lens with Fujicolor or Fujichrome 100 film at F9.5 and 1 second.

The photos were provided courtesy of Homer Wilson Smith of Art Matrix, Ithaca, New York, whose stock of photos of the Mandelbrot set are remarkable for their complex beauty.

1. H. W. Smith, *Mandelbrot Sets and Julia Sets*, Art Matrix Corp., PO Box 880, Ithaca, NY 14851-0880.
2. A. K. Dewdney, "Computer Recreations," *Scientific American*, August 1985, p. 16.
3. B. B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman and Co., New York, 1983.
4. François Robert, *Discreet Iterations*, translated by Jon Rokne, Springer-Verlag, Berlin, in press.

classified as *top-down, bottom-up,* and *middle-out.*

Top-down design methodology. This approach starts by defining, specifying, refining, and validating the requirements of the application; devising methods to collect the necessary knowledge and meta-knowledge; choosing an appropriate representation for the knowledge and meta-knowledge; studying problems with the given representation scheme that are related to the control of correct and efficient execution; identifying functional requirements of components; and mapping these components, subject to technological and cost constraints, into software and micro-level, macro-level, and system-level architectures.

The process is iterative. For example, the representation of knowledge and the language features may be changed or restricted if it is discovered that the functional requirements cannot be mapped into a desirable and realizable system with the given technology and within the set cost. In some projects, the requirements may be very loose and span many different applications. As a result, the languages and knowledge-representation schemes used may be oriented toward general-purpose usage. The Japanese Fifth-Generation Computer System project is an attempt to use a top-down methodology to design an integrated, user-oriented, intelligent system for a number of applications.

Bottom-up design methodology. In this approach, the designers first design the computer system; the design is based on a computational model (such as dataflow, reduction, or control-flow) and the technological and cost limitations. Both possible extensions of existing knowledge-representation schemes and languages developed for AI applications are implemented on the given system. Finally, AI applications are coded by means of the representation schemes and languages provided. This is probably the most popular approach to applying a general-purpose or existing system to AI processing. However, it may result in inefficient processing, and the available representation schemes and languages may not satisfy the application requirements.

Middle-out design methodology. This approach is a short-cut to the top-down design methodology. It starts from a proven and well-established knowledge-representation scheme or AI language (most likely a scheme or language developed for sequential processing) and develops both the architecture and the modifications to the language or representation scheme that are necessary to adapt to the application requirements.

This is the approach taken by many designers in designing special-purpose computers for AI processing. It may be subdivided into *top-first* and *bottom-first,* although both may be iterative.

In a *top-first middle-out methodology,* studies are first performed to modify the language and representation scheme to make them more adaptable to the architecture and computational model. Primitives may be added to the language to facilitate parallel processing. Useful features from several languages may be combined. The design of the architecture follows.

In the *bottom-first middle-out methodology,* hardware support for the overhead-intensive operations enables the chosen language or representation scheme to be mapped directly into architecture and hardware. Applications are implemented by means of the language and representation scheme provided. Lisp computers are examples of computers designed with the bottom-first middle-out methodology.

The future

To support efficient processing of AI applications, research must be done in developing better AI algorithms, better AI software-management methods, and better AI architectures.

The development of better algorithms could lead to significant improvement in performance. Many AI algorithms are heuristic in nature, and upper bounds on performance to solve AI problems have not been established as they have been in traditional combinatorial problems. As a consequence, the use of better heuristic information, based on commonsense or high-level meta-knowledge and on better representation of the knowledge, could lead to far greater improvement in performance than an improved computer architecture could provide. Automatic learning methods to aid designers to acquire and manage the new knowledge in a systematic manner are very important.

Better AI software-management methods are essential in developing more efficient and reliable software for AI processing. AI systems are usually open systems and cannot be defined on the basis of a closed-world model. The language must be able to support the acquisition of new knowledge and the validation of existing knowledge. Probabilistic reasoning and fuzzy knowledge may have to be supported. The verification of the correctness of AI software is especially difficult owing to the imprecise knowledge involved and the random way of managing knowledge in a number of declarative languages and representation schemes. Traditional software-engineering design methodologies must be extended to accommodate the characteristics of AI applications.

The role of parallel processing and innovative computer architectures lies in improving the processing time needed to solve a given AI problem. It is important to realize that parallel processing and better computer architectures cannot be used to overcome the exponential complexity of exhaustive enumeration (unless an exponential amount of hardware is used) and are not very useful in extending the solvable problem space. It is unlikely that a problem too large to be solved today by a single computer in a reasonable amount of time can be solved by parallel processing alone, even if a linear speedup can be achieved. The decision to implement a given algorithm in hardware depends on the complexity of the problem the algorithm solves and the frequency of the problem's occurrence. Problems of low complexity can be solved by sequential processing or in hardware if they are frequently encountered; problems of moderate complexity should be solved by parallel processing; and problems of high complexity should be solved by a combination of heuristics and parallel processing.

In many AI systems being developed today, tasks and operations implemented in hardware are those that are frequently executed and have polynomial complexity. These tasks and operations are identified by means of the languages or the knowledge-representation schemes supported. The architectural concepts and parallel-processing schemes applied may be either well-known conventional concepts or new concepts for nondeterministic and dynamic processing. The role of the computer architect lies in choosing a good representation, recognizing tasks for maintaining and learning meta-knowledge that are overhead-intensive, identifying primitive operations in the languages and knowledge-representation schemes, and supporting these tasks and operations in hardware and software.

In this issue

This special issue of *Computer* is a collection of articles describing a number of projects in this active area called AI computers.

The first article, "Computer Architectures for Artificial Intelligence Processing," by K. Hwang, J. Ghosh, and R. Chowkwanyun, is a survey of computers for AI processing. Existing efforts are classified as multiprocessors supporting MIMD operations, multicomputers supporting multiple SISD processing, and multipurpose computers operating in an SIMD, or multiple-SIMD, or MIMD fashion. The architecture, languages, execution paradigms, and principal applications of various AI computers are summarized.

The second article, "Software Development Support for AI Programs," by C. V. Ramamoorthy, S. Shekhar, and V. Garg, presents the problems faced in designing the software-development environment so that it will support all phases of the software-development cycle of an AI program: requirement specification, design, implementation, testing, and maintenance. The evolution of support for development of AI programming is described with respect to discrete tools, toolboxes, life-cycle support, knowledge-based tools, and intelligent life-cycle support environments.

The third article, "Symbolics Architecture," by D. A. Moon, details the design philosophy of and trade-offs in the Symbolics Lisp computers. Three levels of the architecture—system architecture, instruction architecture, and processor architecture—are discussed.

The next two articles discuss systems for the support of object-oriented programming.

The fourth article, "The Architecture of FAIM-1," by J. M. Anderson, W. S. Coates, A. L. Davis, R. W. Hon, I. N. Robinson, S. V. Robison, and K. S. Stevens, presents the design of FAIM-1, a concurrent, general-purpose, symbolic accelerator for parallel AI symbolic computations. The OIL language supported by FAIM-1 has object-oriented, logic-programming, and procedural-programming features.

The fifth article, "What Price Smalltalk?" by D. Ungar and D. Patterson, discusses the design of a Reduced Instruction Set Computer for Smalltalk-80. The requirements of the Smalltalk-80 programming environment and the valuable lessons learned by implementing clever ideas in hardware that does not significantly improve overall performance are presented.

The sixth article, "Initial Performance of the DADO2 Prototype," by S. J. Stolfo, presents the design trade-offs of, improvements achieved by, and measured performance of DADO2, a parallel-processing system for evaluating production rules and other almost-decomposable search problems.

The last two articles are concerned with architectures for support of knowledge-representation schemes.

The seventh article, "Applications of the Connection Machine," by D. L. Waltz, discusses the architecture and applications of the Connection Machine, a system with massive parallelism. A number of applications, including document retrieval, memory-based reasoning, and natural-language processing, are presented.

The eighth article, "Connectionist Architectures for Artificial Intelligence," by S. E. Fahlman and G. E. Hinton, presents the designs of both NETL, a marker-passing system implementing semantic networks, and value-passing systems (which the authors exemplify by the Hopfield and Boltzmann iterative networks) for constraint satisfaction.

Owing to page limitations, we were unable to include two other articles originally accepted for this special issue. One, "The Architecture of Lisp Machines" by A. R. Pleszkun and M. J. Thazhuthaveetil, enumerates the runtime requirements of a Lisp system and identifies architectural requirements that must be met for good machine performance. The solutions to these requirements in a number of Lisp machines are also discussed. The other, "Computer Architecture for the Processing of a Surrogate File to a Very Large Data/Knowledge Base," by P. B. Berra, S. M. Chung, and N. I. Hachem, shows the design and performance of a proposed back-end system to support the use of surrogate files in data/knowledge bases. The articles will appear in an upcoming issue in the near future.

Despite the large number of articles in this special issue, it was not possible to cover many major projects in this area. I realize that there are many researchers, too numerous to mention individually, who have made notable contributions to the development of this area of research, and I apologize for any inadvertent omissions. Two collections of articles[1,2] that I have compiled also provide reference sources for some of the published articles in this exciting area. □

Acknowledgments

I would like to thank the authors and reviewers for helping to make this special issue a reality. Without them, there would be no special issue. The editor-in-chief of *Computer*, Mike Mulder, and *Computer* Editorial Board member Franklin Kuo helped me through the formalities of publication. I am also grateful to G. J. Li for his comments and to K. Lindquist for her secretarial support. I would like to acknowledge the support of National Science Foundation Grant DMC 85-19649 for this project.

References

1. B. W. Wah and G. J. Li, *Computers for Artificial Intelligence Applications*, IEEE Computer Society Press, Washington, DC, 1986.
2. B. W. Wah and G. J. Li, "A Survey of Special-Purpose Computer Architectures for AI," *ACM SIGART Newsletter*, Number 66, Apr. 1986, pp. 28-46.

Benjamin W. Wah is an associate professor in the Dept. of Electrical and Computer Engineering and in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.

He was on the faculty of the School of Electrical Engineering at Purdue University between 1979 and 1985.

His current research interests include parallel computer architectures, artificial intelligence, distributed databases, computer networks, and theory of algorithms.

Wah was an IEEE-CS Distinguished Visitor between 1983 and 1986. He is an editor of the *IEEE Transactions on Software Engineering* and the *Journal of Parallel and Distributed Computing*. He is the program chairman of the 1987 IEEE International Conference on Data Engineering.

He has authored *Data Management in Distributed Systems* (University Microfilm International Research Press, 1982), and has coedited a *Tutorial on Computers for AI Applications* (IEEE-CS Press, 1986).

He received a PhD in computer science from the University of California at Berkeley in 1979.

Readers may write to Benjamin Wah about this special issue at the University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, 1101 W. Springfield Ave., Urbana, IL 61801.