1. INTRODUCTION

Many of today's computers are single-processor von Neumann machines designed for sequential and deterministic numerical computations [23, 25, 358, 289], and are not equipped for artificial intelligence (AI) applications that are mainly parallel, nondeterministic, symbolic manipulations [16, 49, 127, 387]. Consequently, efficient computer architectures for AI applications would be sufficiently different from traditional computers [50, 104, 105, 134, 153, 187, 274, 281, 326, 407]. These architectures have the following requirements.

1. Symbolic processing. In the microlevel, AI applications require symbolic processing operations such as comparison, selection, sorting, matching, logic set operations (union, intersection, and negation), contexts and partition, transitive closure, pattern retrieval, and recognition. In a higher level, these applications may require the processing of nonnumerical data such as sentences, speech, graphics, and images. Efficient computers designed for these applications should possess hardware for symbolic processing functions [60, 191, 285]. The most important ones are tagged mechanisms [87, 191] and hardware stacks [117].
2. Parallel and distributed processing. Most AI problems are complex [84, 305] and must be evaluated by high-performance computers. Due to technological limitations of physical devices, parallelism is perhaps the only promising mechanism to further improve the performance of computers [182, 187, 198, 249, 315, 387]. To prevent the bottleneck of a centralized controller, intelligence in such a system should be decentralized. In applying multiprocessing and distributed processing to solve problems with exponential complexity, which is typical for problems in AI, one must realize that multiprocessing is useful in improving the computational efficiency, and not in extending the solvable problem size [404]. To extend the solvable problem space of such problems, the key is to find better models and more efficient heuristics.

3. Nondeterministic processing. Most AI algorithms are nondeterministic [78], that is, it is impossible to plan in advance the procedure to execute and to terminate with the available information. Therefore, dynamic allocation and load balancing of computational resources are essential in AI architectures [29, 187]. Further, an efficient interconnection network is needed to disseminate information for the scheduler. The tradeoff between the overhead of distributing the scheduling information and the overhead for the extra work needed without the scheduling information must be made. Moreover, efficient garbage collection is important for AI architectures, owing to be dynamically allocated storage [29, 79, 248].

4. Knowledge-base management. Because a very large amount of information has to be stored and retrieved in AI applications, large knowledge bases are inevitable [20, 34, 319, 412]. An implementation using a common memory is inappropriate due to access conflicts. A decentralized memory system with distributed intelligence and capabilities for pattern matching and proximity search is required.

5. Software-oriented computer architectures. The efficiency of a computer system for an AI application depends strongly on its knowledge representation and the language used. An efficient AI architecture should be designed around the knowledge representations of the problems to be solved and the high-level AI languages to be supported. Further, the designed architectures should adapt to changes in granularity and data formats of various applications. Examples of these architectures are the dataflow machines [41, 205], object-oriented architectures [204, 381], Lisp machines [87, 105], and Prolog-like machines, such as the Fifth Generation Computer System [153].

Currently, extensive research is underway to design efficient AI architectures. Many existing concepts in computer architecture such as dataflow processing [107, 388], stack machines [117], tagging [191], pipelining [198], direct execution of high-level languages [68, 144, 424], data base machines [237], multiprocessing, and distributed processing can be incorporated into future AI architectures. New concepts in computer architectures are also expected.

---

2. A.

On the other hand, the idea that AI can be implemented using a Neural Net model has not yet been explored. Thus, it appears that a type of learning in neural nets which is close to true learning is impossible, even in these simple models.

Further, it is important to develop algorithms for learning in neural nets. In a general situation, one would like to consider all the possible learning algorithms that the neural nets should exhibit.

In this section, we consider an important question of how to design algorithms that are good for learning in neural nets. We show that neural nets can be constructed which are capable of learning in neural nets.

Namely, neural nets can be designed to learn in neural nets. Such nets can be used to solve a wide variety of problems, some of which, however, may be difficult to solve using traditional algorithms. The neural nets described in this book use only simple, closed formulas to perform their computations.

A single neural net can be used to perform a wide variety of tasks, as long as it is trained properly. A neural net is trained by adjusting its weights to minimize error. A neural net is trained in a manner similar to that of a human brain, in which the error is reduced by changing the weights of the connections between neurons.

Generally, a neural net is trained using a set of input data and a set of desired output data. The input data is fed into the neural net, and the output data is compared to the output produced by the neural net. The error between the desired output and the actual output is used to adjust the weights of the connections between neurons.

The training process can be simplified by using back-propagation, a technique that updates the weights in a way that reduces the error at each step. The back-propagation algorithm is an iterative process that adjusts the weights in the neural net to minimize the error.

In the training process, the weights are adjusted in the direction of the error gradient. The weights are updated in the direction that reduces the error, so that the error at the next iteration is smaller than the error at the current iteration.

The back-propagation algorithm is a powerful tool for training neural nets. It allows the neural net to learn from its mistakes and improve its performance over time. The algorithm is computationally efficient, as it requires only a few iterations for convergence.

The back-propagation algorithm has been successfully applied to a wide range of problems, including pattern recognition, classification, and function approximation. It is a widely used method for training neural nets, and has been shown to be effective in many different applications.
2. AI LANGUAGES AND PROGRAMMING

One goal of computer scientists working in the field of AI is to produce programs that imitate the intelligent behavior of human beings [30, 44, 67, 313, 419]. Von Neumann-style programming that uses imperative languages, such as Fortran and Pascal, is inadequate due to its inability to specify parallel tasks and its unacceptable complexity [21, 224, 417]. To enhance programmers' productivity, a type of problem-oriented languages called declarative languages have been developed and widely applied in AI programming [121]. Functional programming [22, 183, 417] and logic programming [72, 226-228, 316] are the major programming paradigms of declarative languages.

Functional programming does not contain any notion of the present state, program counter, or storage. Rather, the "program" is a function in the true mathematical sense: it is applied to the input of the program, and the resulting value is the program's output. The terms functional language, applicative language, dataflow language, and reduction language have been used some-what interchangeably [57, 95, 199, 390]. Examples of functional languages are pure Lisp [26, 170, 260, 261, 321], Backus' FP [21], Hope [58], Val [264], and Id [15].

Interest in functional programming is steadily growing because it is one of the few approaches that offer a real hope of relieving the twin crises of AI-oriented computing today: the absolute necessity to reduce the cost of programming, and the need to find computer designs that make much better use of the power of very large scale integration (VLSI) and parallelism.

In its modest form, a logic program refers to the procedural interpretation of Horn clauses or predicate logic [226, 227]. The computer language Prolog [73, 75, 80, 81, 116, 406] is based on logic programming. Generally speaking, logic programming is reasoning-oriented or deductive programming. In fact, some ideas of logic programming, like automatic backtracking, have been used in early AI languages QA3 [30], PLANNER, MICROPLANNER, and CONNIER [44, 363]. Logic programming has recently received considerable attention because of its choice by the Japanese as the core computer language for the Fifth Generation Computer System Project [284]. Although it seems on the surface that logic programming is an independent and somewhat separate notion from function programming, an ideal AI-programming style should combine the features of both languages and may be called "assertional programming" [316].

New languages and programming systems are being developed to simplify AI programming radically. It is expected that object-oriented programming [312] will be as important in the 1980s as structured programming was in the 1970s. The language Smalltalk [2, 160] is an example of object-oriented programming. Some ideas of object-oriented programming have been used in existing languages and systems, such as Simula, B5000, Lisp-AI notion of frame, ADA, and CLI. Other new object-oriented programming systems have also been developed [204, 272, 365, 381, 397].

AI programming languages have had a central role in the history of AI research. Frequently, new ideas in AI are accompanied by a new language in
which it is natural for the ideas to be applied. Except for the widely used language Prolog, Lisp and its dialects, Maclisp [277], Interlisp [338, 376], Qlisp [318], Common Lisp [351], Franz Lisp [413], etc., many other AI languages have been designed and implemented. Examples include IPL [296, 297] PLANNER [184], CONNIVER [363], KRL [46], NETL [126], SAIL [311], POP-2 [93], FUZZY [131], and first-order logic. In general, three capabilities, namely action, description, and reasoning, are needed for an AI language. Historically, languages strong in one of these capacities tended to be relatively weak in others. Prolog is a reasoning-oriented language that is limited by its inefficiency of description and action. Lisp, the second oldest programming language in general widespread use retains some features of von Neumann programming. Some new languages, such as Loglisp [316] and QUTE [325], which amalgamate Prolog and Lisp in natural ways, have been developed. On the other hand, to explore parallelism, parallel versions of Prolog and Lisp, such as Parlog [70], Concurrent Prolog [333, 335], and Concurrent Lisp [361, 367], have been proposed. Recent efforts have been aimed at automatic programming that will allow the program to be generated from a simple specification of the problem [31, 32, 242, 256, 343].

It has also been apparent to the AI community since the mid 1960s that inferences alone, even those augmented with heuristics, were often inadequate to solve real-life problems. To enhance the performance of AI programs, they must be augmented with knowledge of the problem domain rather than formal reasoning methods. This realization gave birth to knowledge engineering or the knowledge-based system, the field of applied AI [13, 240].

A knowledge-based expert system, or in short, expert system, is a knowledge-intensive program that solves problems in a specific domain normally requiring human expertise [36, 99, 157, 179–181, 294, 407]. An expert system consists of two parts: knowledge base and inference procedure. The knowledge base contains the facts and heuristics, while the inference procedure consists of the processes that search the knowledge base to infer solutions to problems, form hypotheses, and so on. What distinguishes an expert system from an ordinary computer application is that, in a conventional computer program, pertinent knowledge and the methods for utilizing it are all intermixed, whereas in an expert system, the knowledge base is separated from the inference procedure, and new knowledge can be added to the system without programming.

Contemporary expert-system development techniques are shifting towards the use of software development tools that resemble a programming language, but include internal user-accessible data bases and other high-level strategies for using knowledge to solve a class of problems [88, 157, 181, 220]. Each tool suggests some additional design properties, such as rule-base and backward reasoning, for the knowledge-system architecture. Three of the most popular families of expert-system tools are: (1) MYCIN [267, 268], KS:300, and S:1; (2) HEARSAY-III [125] and AGE [299]; and (3) OPS that incorporates the R1 (XCON) expert-system families [145]. Other expert-system tools include LOOPS [354], ROSIE [130], RLL [167] MRS, and KMS. Some of these tools...
aim to provide a mixture of representations and inference techniques. Knowledge-acquisition tools such as TEIRESIAS [97], EXPERT [411], KAS [120], and learning tools such as META-DENDRAL [55] and EURISKO [241] have also been developed.

3. MICRO-AND MACROLEVEL AI ARCHITECTURES

The VLSI technology has flourished in the past ten years [1, 54, 266, 329, 382], resulting in the development of advanced microprocessors [384], semiconductor memories [415], and systolic arrays [147, 148, 168, 231, 330].

The microlevel architectures consist of architectural designs that are fundamental to applications in AI. In the design of massively parallel AI machines [129], some of the basic computational problems recognized are set intersection, transitive closure, contexts and partitions, best-match recognition, Gestalt recognition, and recognition under transformation. These operations may not be unique in AI and may exist in many other applications as well. Due to the simplicity of some of these operations, they can usually be implemented directly in hardware, especially in systolic arrays using the VLSI technology. Many other basic operations can also be implemented in VLSI. Examples include sorting [35, 43, 51, 197, 229, 309, 378, 379, 395, 418] and selection [401, 425], computing transitive closure [171, 231], string and pattern matching [8, 13, 14, 162, 174, 192, 285, 374], selection from secondary memories [119, 161], dynamic programming evaluations [39, 64, 171], proximity searches [427], and unification [118, 146, 380, 399, 400].

Some AI languages such as Lisp differ from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of numbers or bit fields of a fixed size. The instruction set must be designed according to the storage structure [353]. Additional concepts that are well suited for list processing are the tagged-memory [137, 191] and stack architectures [117].

The macrolevel is an intermediate level between the microlevel and the system level. In contrast to the microlevel architectures, the macrolevel architectures are (possibly) made up of a variety of microlevel architectures and perform more complex operations. However, they are not considered as a complete system that can solve problems in AI applications, but can be taken as more complex supporting mechanisms for the system level. The architectures can be classified into dictionary machines, data-base machines, architectures for searching, and architectures for managing data structures.

A dictionary machine is an architecture that supports the insertion, deletion, and searching for membership, extremum and proximity of keys in a data-base [18, 37, 61, 141, 238, 303, 348]. Most designs are based on binary-tree architectures; however, designs using radix trees and a small number of processors have been found to be preferable when keys are long and clustered [141].
A data-base machine is an architectural approach that distributes the search intelligence into the secondary and mass storage and relieves the workload of the central processor. Extensive research has been carried out in the past decade on optical and mass storage [270, 271], back-end storage systems [150], and data-base machines [19, 52, 156, 177, 196, 217, 236, 251, 336, 339]. Data-base machines developed earlier were mainly directed towards general-purpose relational database management systems. Examples include the DBC, DIRECT, RAP, CASSM, associative array processors, text retrieval systems [196, 236], and CAFS [19]. Nearly all current research on data-base machines to support knowledge databases assumes that the knowledge data-base is relational, hence research is directed towards solving the disk paradox [52] and enhancing previous relational data-base machines by extensive parallelism [287, 319, 340, 373]. Commercially available data-base and backend machines have also been applied in knowledge management [213, 214, 295].

Searching is an essential to many applications, although unnecessary combinatorial searches should be avoided. The suitability of parallel processing to searching depends on the problem complexity, the problem representation, and the corresponding search algorithms. Problem complexity should be low enough such that a serial computer can solve the problem in a reasonable amount of time. Problem representations are very important because they are related to the search algorithms. Parallel algorithms have been found to be able to dramatically reduce the average-time behavior of search problems, the so-called combinatorially implosive algorithms [222, 223, 408].

A search problem can be represented as searching an acyclic graph or a search tree. According to the functions of nodes in the graph, the problem is transformed into one of the following paradigms: (a) AND-tree (or graph) search: all nonterminal nodes are AND-nodes, (b) OR-tree (or graph) search: all nonterminal nodes are OR-nodes, and (c) AND/OR-tree (or graph) search: the nonterminal nodes are either AND- or OR-nodes. A divide-and-conquer algorithm is an example algorithm to search AND-trees; a branch-and-bound algorithm is used to search OR-trees, and an alpha-beta algorithm is used to search (AND/OR) game trees. Parallel algorithms for divide-and-conquer [195], branch-and-bound [10, 24, 106, 149, 234, 235, 245, 247], and AND/OR-graph search [140, 148, 258] have been developed. Various parallel architectures to support divide-and-conquer algorithms [307, 341] and branch-and-bound algorithms [108, 122, 139, 175, 201, 202, 359, 402, 404] have been proposed.

Extensive research has been carried out in supporting dynamic data structures in a computer with a limited memory space. Garbage collection is an algorithm that hierarchically relays memory space no longer needed by the users [27, 29, 33, 45, 79, 110, 114, 136, 188, 230, 248, 259, 298, 349, 352]. This is usually transparent to the users and could be implemented in hardware, software, or a combination of both. For efficiency reasons, additional hardware such as stacks and reference counters are usually provided.
4. FUNCTIONAL PROGRAMMING ORIENTED ARCHITECTURES

The origin of functional languages as a practical class of computer languages can perhaps be traced to the development of Lisp by McCarthy [261] in the early 1960s, but their ancestry went directly back to the Lambda Calculus developed by Church in the 1930s. The objective of writing a functional program is to define a set of (possibly recursive) equations for each function [91]. Data structures are handled by introducing a special class of functions called constructor functions. This view allows functional languages to deal directly with structures that would be termed "abstract" in more conventional languages. Moreover, functions themselves can be passed around as data objects. The design of the necessary computer architecture to support functional languages thus centers around the mechanisms of efficient manipulation of data structures (list-oriented architectures) and the parallel evaluation of functional programs (function-oriented architectures).

List-oriented architectures are architectures designed to efficiently support the manipulation of data structures and objects. Lisp, a mnemonic for list-processing language, is a well-known language to support symbolic processing. There are several reasons why Lisp and list-oriented computers are really needed. First, to relieve the burden on the programmers, Lisp was designed as an untyped language. The computer must be able to identify the types of data, which involves an enormous amount of data-type checking and the use of long strings of instructions at compile- and run-times. Conventional computers cannot do these efficiently in software. Second, the system must periodically perform garbage collection and reclaim unused memory at run-time. This amounts to around 10 to 30% of the total processing time in a conventional computer. Hardware implementation of garbage collection is thus essential. Third, due the nature of recursion, a stack-oriented architecture is more suitable for list processing. Lastly, list processing usually requires an enormous amount of space, and the data structures are so dynamic that the compiler cannot predict how much space to allocate at compile-time. Special hardware to manage the data structures and the large memory space would make the system more cost-effective and efficient [105, 132, 143, 290].

The earliest implementation of Lisp machines were the PDP-6 computer and its successors the PDP-10 and PDP-20 made by the Digital Equipment Corporation [261]. The half-word instructions and the stack instructions of these machines were developed with Lisp's requirements in mind. Extensive work has been done for the DEC-System 10's and 20's on garbage collection to manage and reclaim the memory space used.

The design of Lisp machines was started at MIT's AI Laboratory in 1974. CONS, designed in 1976 [36, 165, 219, 328], was superseded in 1978 by a second-generation Lisp machine, the CADR. This machine was a model for the first commercially available Lisp machines [22, 257, 291], including the Symbolics LM2, the Xerox 1100 Interlisp work station, and the Lisp Machine Inc. Series III
CADR, all of them delivered in 1981. The third-generation machines were based on additional hardware to support data tagging and garbage collection. They are characterized by the Lisp Machines Inc.’s Lambda, supporting Zetalisp and LMLisp [87, 257, 291]. The Symbolics 3600, supporting Zetalisp, Flavors, and Fortran 77 [19, 278, 405, 409], the Xerox 1108 and 1132, supporting Interlisp-D and Smalltalk [47, 280, 337], and the Fujitsu FACOM Alpha Machine, a backend Lisp processor, supporting Maclisp [9, 178]. Most of the Lisp machines support networking using Ethernet. The LMI Lambda has a NuBus developed at MIT to produce a modular, expandable Lisp machine with multiprocessor architecture.

A single-chip computer to support Lisp has been implemented in the MIT SCHEME-79 chip [350, 353, 364]. Other experimental computers to support Lisp and list-oriented processing have been reported [111, 164, 166, 169, 292, 310, 322–324, 370]. These machines usually have additional hardware tables, hashing hardware, tag mechanisms, and list-processing hardware, or are microprogrammed to provide macroinstructions for list-processing. Experimental multiprocessor systems have been proposed to execute Lisp programs concurrently [173, 186, 265, 273, 361, 362, 414]. Dataflow processing is suitable for Lisp because these programs are generally data driven [422, 423]. Other multiprocessors and dataflow architectures to support list-processing have been proposed and developed [11, 12, 77, 113, 159, 385].

Besides specialized hardware implementations, software implementations on general-purpose computers are also popular. The earliest Lisp compilers were developed on the IBM 704 and later extended to the IBM 7090, 360, and 370. Various strategies for implementing Lisp compilers have been proposed [60, 103, 170, 320, 321, 376], and conventional microcomputers have been used to implement Lisp compilers [216, 368]. Lisp is also available on various general- and special-purpose work stations, typically based on multiple 68000 processors [216, 368]. Lisp has been developed on Digital Equipment Corp.’s VAXStation 100, a MC68000-based personal graphics work station, and clusters of 11/782s running several dialects of Lisp and Common Lisp [360]. One dialect of Lisp, Franz Lisp, developed at the University of California, Berkeley, was written in C and runs under Unix and is available on many general-purpose work station.

Architectures have also been developed to support object-oriented programming languages which have been extended from functional languages to additionally implement operations such as creating an object, sending and receiving messages, modifying an object’s state, and forming class-superclass hierarchies [185, 381, 421]. Smalltalk, first developed in 1972 by the Xerox Corp., is recognized as a simple but powerful way of communicating with computers. At MIT, the concept was extended to become the Flavors systems. Special hardware and multiprocessors have been proposed to directly support the processing of object-oriented languages [204, 308, 365, 397].

In function-oriented architectures, the design issues center on the physical interconnection of processors, the method used to “drive” the computation, the
representation of programs and data, the method to invoke and control parallelism, and the optimization techniques [398]. Desirable features of such architectures should include a multiprocessor system with a rich interconnection structure, the representation of list structures by balanced trees, and hardware supports for demand-driven execution, low-overhead process creation, and storage management.

Architectures to support functional-programming languages can be classified as uniprocessor architectures, tree-structured machines, data-driven machines, and demand-driven architectures. In a uniprocessor architecture, besides the mechanisms to handle lists, additional stacks to handle function cells and optimization for redundant calls and array operations may be implemented [6, 38, 63, 350, 390]. Tree-structured machines usually employ lazy evaluations, but suffer from the bottleneck at the root of the tree [94, 210, 251, 253, 300]. Dataflow machines are also natural candidates for executing functional programs and have tremendous potential for parallelism. However, the issue of controlling parallelism remains unresolved. A lot of the recent work is concentrated on demand-driven machines, which are based on reduction machines on a set of load-balanced (possibly virtual) processors [74, 90, 151, 194, 211, 212, 218, 221, 344, 383, 385].

Owing to the different motivations and objectives of various functional programming-oriented architectures, each machine has its own distinct features. For example, the Symbolics 3600 [278] was designed for an interactive program development environment where compilation is very frequent and ought to appear instantaneous to the user. This requirement simplified the design of the compiler and results in only a single-address instruction format, no indexed and indirect addressing modes, and other mechanisms to minimize the number of nontrivial choices to be made. On the other hand, the aim in developing SOAR [397] was to demonstrate that a Reduced Instruction Set Computer could provide high performance in an exploratory programming environment. Instead of microcode, SOAR relied on software to provide complicated operations. As a result, more sophisticated software techniques were used.

5. LOGIC- AND KNOWLEDGE-ORIENTED ARCHITECTURES

In logic- and knowledge-oriented architectures, the ideal goal is for the user to specify the problem in terms of the properties of the problem and the solution (logic or knowledge), and the architecture exercises the control on how the problem is to be solved. This goal is not fully achieved yet, and users still need to provide small but undue amounts of control information in logic programs, partly by ordering the clauses and goals in a program, and partly by the use of extra-logical "features" in the language.

Knowledge- and logic-oriented architectures can be classified according to the knowledge representation schemes. Besides incorporating knowledge into a program written in a functional programming language, some of the well-known
schemes are logic programs and semantic networks. According to the search strategy, logic programs can further be classified into production systems and logical inference systems [48, 50, 118, 146, 326, 407].

Substantial research has been carried out on parallel computational models of utilizing AND-parallelism, OR-parallelism, and stream parallelism in logical inference systems [41, 62, 66, 69, 82, 83, 102, 215, 246, 250, 293, 302, 342, 396, 420, 426], production systems [301, 317, 377], and others [154]. The basic problem on the exponential complexity of logic programs remains open at this time.

Sequential Prolog machines using software interpretation [7, 200], emulation [76, 429], and additional hardware support such as hardware unification and backtracking [207, 372, 380] have been proposed. Single-processor systems for production systems using additional data memories [239] and a RISC architecture [146] have been studied.

New logic programming languages suitable for parallel processing have been investigated [193]. In particular, the use of predicate logic [123], extensions of Prolog to become Concurrent Prolog [53, 233, 331, 334, 366, 375, 393], Parlog [71], and Delta-Prolog [306], and parallel production systems [394] have been developed. Concurrent Prolog has also been extended to include object-oriented programming [331] and has been applied as a VLSI design language [366]. One interesting parallel language is that of systolic programming, which is useful as an algorithm design and programming methodology for high-level-language parallel computers [332].

Several prototype multiprocessor systems for processing inference programs and Prolog have been proposed, some of which are currently under construction. These systems include multiprocessors with a shared memory [53], ZMOB, a multiprocessor of Z80's connected by a ring network [65, 208, 314, 389, 410], AQUARIUS, a heterogeneous multiprocessor with a crossbar switch [109], and MAGO, a cellular machine implementing a Prolog compiler that translates a Prolog program into a formal functional program [225]. Techniques for analyzing Prolog programs such that they can be processed on a dataflow architecture have been derived [40, 176, 203, 205]. DADO is a multiprocessor system with a binary-tree interconnection network that implements parallel production systems [172, 355, 357]. An associative processor has been proposed to carry out propositional and first-order predicate calculus [115].

It has been recognized that a combination of Lisp, Prolog, and an object-oriented language such as Smalltalk may be a better language for AI applications [369]. A computer of this type that implements a combination of the AI languages may use microprogramming to emulate the various functions. Prolog is also available as a secondary language on some Lisp machines. A version of Prolog interpreter with a speed of 4.5 KLIPS has been developed for Lisp Machine's Lambda [257]. Some of the prototype multiprocessors, such as ZMOB [65, 208, 314, 389, 410] and MAGO [225] were developed with a flexible architecture that can implement object-oriented, functional, and logic languages. FAIM-1, a multiprocessor connected in the form of a twisted hex-plane topology,
implements the features of object-oriented, functional, and logic programming in the OIL programming language [96].

Besides being represented in logic, knowledge can also be represented in terms of semantic nets. Proposed and experimental architectures have been developed. NETL [126, 128, 138], and its generalization to THISTLE [129], consists of an array of simple cells with marker-passing capability to perform searches, set-intersections, inheritance of properties and descriptions, and multiple-context operations on semantic nets. Thinking Machine's Connection Machine is a cellular machine with 65,536 processing elements. It implements marker passing and virtually reconfigures the processing elements to match the topology of the application semantic nets [86, 186]. Associative processors for processing semantic nets have also been proposed [275, 276].

Some AI architectures are based on frame representations and may be called object-oriented architectures. For example, the Apiary developed at MIT is a multiprocessor actor system [186]. Actor is an object that contains a small amount of state and can perform a few primitive operations: sending a message, creating another actor, making a decision, and changing its local state. An efficient AI architecture also depends on the problem-solving strategy. The basic idea of the Boltzmann machine developed at the Carnegie-Mellon University is the application of statistical methods to constraint-satisfaction searches in a parallel network [190]. The most interesting aspect of this machine lies in its domain-independent learning algorithm [17].

With the inclusion of control into stored knowledge, the resulting system becomes a distributed problem-solving system. These systems are characterized by the relative autonomy of the problem-solving nodes, a direct consequence of the limited communication capability [98, 100, 133]. With the proposed formalism of the Contract Net, contracts are used to express the control of problem solving in a distributed processor architecture [345–347]. Related work in this area include Petri-net modeling [304], distributed vehicle-monitoring testbed [84, 243], distributed air-traffic control system [59], and modeling the brain as a distributed system [152, 158].

6. FIFTH GENERATION COMPUTER SYSTEM
The Fifth Generation Computer System (FGCS) project was a project that started in Japan in 1982 to further the research and development of the next generation of computers. It was conjectured that computers of the next decade will be used increasingly for nonnumeric data-processing such as symbolic manipulation and applied AI. The goals of the FGCS project are

1. To implement basic mechanisms for inference, association, and learning in hardware
2. To prepare basic AI software in order to utilize the full power of the basic mechanisms implemented
3. To implement the basic mechanisms for retrieving and managing a knowledge base in hardware and software
4. To use pattern recognition and AI research achievements in developing user-oriented man-machine interfaces
5. To realize supporting environments for resolving the "software crisis" and enhancing software production

The FGCS project is a marriage between the implementation of a computer system and the requirements specified by applications in AI, such as natural-language understanding and speech recognition. Specific issues studied include the choice of logic programming over functional programming, the design of the basic software systems to support knowledge acquisition, management, learning, and the intelligent interface to users, the design of highly parallel architectures to support inferencing operations, and the design of distributed-function architectures that integrates VLSI technology to support knowledge data-bases [42, 153, 209, 282, 284, 386].

A first effort in the FGCS project is to implement a sequential inference machine, or SIM [392, 428]. Its first implementation is a medium-performance machine known as a personal sequential inference, or PSI, machine [371, 430]. The current implementation is on the parallel inference machine, or PIM [163, 205, 206, 283, 287, 288, 391]. Another architectural development is on the knowledge-base machine. Delta [286-319, 340]. Lastly, the development of the basic software system acts as a bridge to fill the gap between a highly parallel computer architecture and knowledge information processing [155, 263, 269]. Currently, all the projects are progressing well; however, the struggle is still far from over [255].

The Japanese FGCS project has stirred intensive responses from other countries [3-5, 89, 90, 92, 101, 124, 279, 344, 416]. The British project is a five-year $550 million cooperative program between government and industry that concentrates on software engineering, intelligent knowledge-based systems, VLSI circuitry, and human-machine interfaces. Hardware development has focused on ALICE, a Parlog machine using dataflow architectures and implementing both Hope, Prolog, and Lisp [89, 90, 92, 101, 279, 344]. The European Comission has started the $1.5 billion five-year European Strategic Program for Research in Information Technologies (Esprit) in 1984 [4]. The program focuses on microelectronics, software technology, advanced information processing, computer-integrated manufacturing, and office automation. In the United States, the most direct response to the Japanese FCQS project was the establishment of the Microelectronics and Computer Technology Corp. in 1983 [3]. The project has an annual budget of $50 million to $80 million per year. It has a more evolutionary approach than the revolutionary approach of the Japanese and should yield technology that the corporate sponsors can build into advanced products in the next 10 to 12 years. Meanwhile, other research organizations have

...for more information on the content of this page.
formed to develop future computer technologies of the United States in a broader sense. These include DARPA's Strategic Computing and Survivability, the semiconductor industry's Semiconductor Research Corporation, and the Microelectronics Center of North Carolina [3].

7. CONCLUSIONS
This survey briefly summarizes the state of the art in AI architectures. Conventional von Neumann computers are unsuitable for AI applications because they are designed mainly for deterministic numerical processing. To cope with the increasing inefficiency and difficulty in coding algorithms in AI, declarative languages have been developed. Lambda-based and logic-based languages are two popular classes of declarative languages.

One of the architect's starting point in supporting applications in AI is the language. This approach has been termed the language-first approach. A possible disadvantage of this approach is that each language may lead to a quite distinct architecture which is unsuited to other languages, a dilemma in high-level language computer architectures. In AI applications, the lambda-based and logic-based languages have been considered seriously by novel architects. Recent research lies in integrating the logic and lambda languages, and the work on lambda and logic oriented architectures provides useful guidelines for parallel architectures that support more advanced languages. On the other hand, AI architectures are also related to knowledge representations. This approach has been called the knowledge-first approach. Several architectures have been designed to support multiple knowledge representations.

An appropriate methodology to design an AI architecture should combine the top-down and bottom-up design approaches. That is, we need to develop functional requirements based on the AI problem requirements and map these requirements into architectures based on technological requirements. Parallel processing is a great hope to increase the power of AI machines. However, parallel processing is not a way to overcome the difficulty of combinatorial explosion. It cannot significantly extend the solvable problem space on problems that we can solve today. Hence the problem complexity is an important consideration in designing AI machines. Problems of lower complexity may be solved by sequential computation; problems of moderate complexity may be solved by parallel processing; while problem of high complexity should be solved by heuristic and parallel processing. Since the complexities of most AI problems are high, an appropriate approach should start by first designing good heuristics to reduce the serial-computational time, and using parallel processing to pursue a near-linear speed-up.

Although many AI architectures have been built or proposed, the Lisp machines are the only architectures that have had widespread use for solving real AI problems. Most underlying concepts in AI architectures are not new and have been used in conventional computer systems. For example, hardware stack and
tagged memory were proposed before they were used in Lisp machines. On the other hand, some popular architectural concepts in current supercomputers will have restricted use in some AI applications. For example, the large amount of branch and symbolic processing operations in AI programs reduces stream parallelism in pipelining.

The question of how AI programs can be executed directly in hardware efficiently is still largely unanswered. The following are some key issues in designing AI architectures:

1. Identification of parallelism in AI programs
2. Tradeoff between the benefit and the overhead on the use of heuristic information
3. Efficient interconnection structure to distribute heuristic-guiding and pruning information
4. Granularity of parallelism
5. Dynamic scheduling and load balancing
6. Architecture to support the acquisition and learning of heuristic information
7. Prediction of performance and linear scaling
8. Management of the large memory space.

REFERENCES


159. W. K. Giloi and R. Gueth, “Concepts and Realization of a High-Performance Data Type


350. G. Steele and G. Sussman, Design of Lisp-Based Processor, or SCHEME: A Dielectric Lisp or Finite Memories Considered harmful, or LAMBDA: The Ultimate Opcode, AI Memo 514, MIT, Cambridge, MA, March 1979.


