DESIGN METHODOLOGIES OF COMPUTERS FOR ARTIFICIAL INTELLIGENCE PROCESSING

Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

Many of today's computers are single-processor von Neumann machines designed for sequential, deterministic, and numeric computations and are not equipped for AI applications which are mainly parallel, nondeterministic, and symbolic manipulations. Consequentially, efficient computer architectures for AI applications would be significantly different from traditional computers. These architectures have the following requirements.

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. In 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 in advance the procedures to execute and to terminate with the available information. This is attributed to a lack of knowledge and a complete understanding of the problem and results in exhaustively enumerating all possibilities when the problem is solved.

Dynamic execution. With a lack of complete knowledge and anticipation 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 processing. 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 very high 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.

Research supported by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), a NASA-supported Center for Excellence.

Second International Conference of Supercomputing, San Francisco, CA, USA, May 4-7, 1987.

Open system. 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 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.

With these distinct features, the essential issues to design a computer system to support AI applications can be classified into the representation level, the control level, and the processor level.

The representation level deals with the knowledge and methods to solve the problem and the means to represent it. The essential issues to be studied are the (a) *hierarchy of meta-knowledge*, (b) *domain-knowledge representation*, and (c) *AI languages and programming*. 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 of an event to occur. In other words, meta-knowledge is the knowledge about domain knowledge. Meta-knowledge can be considered to exist in a hierarchy. 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. Domain knowledge can be represented in a declarative form or in a procedural form. A declarative representation offers a higher potential for parallelism 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 but may overspecify the precedence constraints and restricts the degree of parallel processing. In choosing the appropriate representation scheme and language, tradeoffs must be performed on the amount of memory space required to store the knowledge, the inference time, the expected usage of the knowledge, and the underlying computer architecture and technological limitations.

The control level are concerned with the detection of dependencies and parallelism in the algorithmic and program representations of the problem to result in correct and efficient execution. The important issues in this level are (a) *truth maintenance*, (b) *partitioning and restructuring*, (c) *synchronization*, and (d) *scheduling*. Truth maintenance consists of recognizing the inconsistency, modifying the state to remove the inconsistency, and verifying that all inconsistencies are detected and corrected properly. Partitioning and restructuring refer to the decomposition and reorganization of the knowledge base and the AI program to result in more efficient processing. Issues similar to those considered for conventional multiprocessing systems are involved. However, the methods to resolve each are different due to the nondeterministic and dynamic nature of exe-

cution. Synchronization is used to maintain the correct execution under semantic and data dependencies. Data dependencies are specified through shared variables. However, in a number of declarative languages such as PROLOG, it is difficult to specify semantic dependencies directly, as subgoals in a clause are specified as a set rather than as a sequence. New languages that combine the feature of functional languages to specify parallel tasks and that of logic languages to specify nondeterminism are evolving. Scheduling refers to the selection of ready tasks to assign to available processors. Scheduling can be static or dynamic. The difficulty in designing a good scheduler lies in the heuristic information to guide the nondeterministic search. In practice, the heuristic information may be fallible. As a result, some AI architects do not schedule nondeterministic tasks in parallel and only consider static scheduling.

The **processor level** consists of the *micro-level*, *macrolevel*, and *system-level* architectures. Microlevel architectures to support AI processing consist of primitive architectural designs that are fundamental to applications in AI. 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 macrolevel is an intermediate level between the microlevel and the system level. Macrolevel architectures are (possibly) made up of a variety of microlevel architectures and 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 towards one or a few of the languages or knowledge-representation schemes and are designed to provide architectural support to overhead-intensive features in these 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 image understanding.

Based on these issues, the various design methodologies can be 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 of the knowledge and meta-knowledge, studying problems related to the control of correct and efficient execution with the given representation scheme, identifying functional requirements of components, and mapping these components into software and microlevel, macrolevel and system-level architectures subject to technological and cost constraints. The process is iterative. 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. Since the requirements are loose and span across multiple applications, the language developed is oriented towards general-purpose usage.

Bottom-up design methodology. In this approach, the designers first design the computer system based on a computational model, such as dataflow, reduction, and control-flow, and the technological and cost limitations. Possible extensions of existing knowledge-representation schemes and languages developed for AI applications are implemented on the given system. Finally, the AI applications are coded using the representation schemes and languages provided. This is probably the most popular approach to apply a general-purpose or existing system for 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 developed for sequential processing) and develops the architecture and the necessary modifications to the language and representation to adapt to the application requirements and the architecture. 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 it more adaptable to the architecture and computational model. Primitives may be added to the language to facilitate parallel processing. Nice features from several languages may be combined. The design of the architecture follows. In the bottom-first middle-out methodology, the chosen language or representation scheme is mapped directly into architecture by providing hardware support for the overhead-intensive operations. Applications are implemented using the language provided. LISP computers are examples designed with the bottom-first methodology.

Some observations on supporting efficient processing of AI applications can be drawn.

(1) Better understanding and representation of meta-knowledge are required to design better parallel algorithms. Many AI algorithms are heuristic in nature, and upper bounds on complexity to solve these problems have not been established as in traditional combinatorial problems. Hence, the use of better heuristic information, based on common-sense or meta-knowledge and better knowledge representation can have far greater impact on performance than what improved computer architecture can provide.

(2) 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 based on 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. Traditional software design methodologies must be extended to accommodate the characteristics of AI applications.

(3) Parallelism and better computer architectures cannot overcome the exponential complexity of exhaustive enumeration and are not very useful to extend the solvable problem space. For a problem with a size that is too large to be solved in a reasonable amount of time, it is unlikely that it can be solved by parallel processing alone, even if a linear speedup can be achieved. In general, 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.

(4) In many AI systems developed today, tasks and operations implemented in hardware are those that are frequently executed and have polynomial complexity. These tasks and operations are identified from 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.

(5) It is not clear on the speedup that parallel processing of nondeterministic tasks will provide, although the potential to process these tasks in parallel is large. Without appropriately guiding the search and and detecting redundant computations, much of the power of parallel processing may be directed toward fruitless searches.

The role of the computer architects lies in choosing a good representation, recognizing overhead-intensive tasks to maintain and learn meta-knowledge, identifying primitive operations in the languages and knowledge-representation schemes, and supporting these tasks in hardware and software.