ARCHITECTURAL ISSUES IN DESIGNING SYMBOLIC PROCESSORS IN OPTICS

Aloke Guha and Raja Ramnarayan

Honeywell Corporate Systems Development Division
1000 Boone Ave. N., Golden Valley, MN 55427

Matthew Derstine

Honeywell Physical Sciences Center
10701 Lyndale Ave. S., Bloomington, MN 55420

ABSTRACT

This paper analyzes potential optical architectures for AI applications (such as knowledge-based systems). Our goal was to investigate architectures most suitable for implementation completely in optics. While optical computing appears to hold much promise because of its inherent parallelism and speed, constructing a symbolic processor or even a general purpose computer in optics requires examining many issues never before addressed. This paper presents these issues and discusses those architectures which appear most feasible in optics. We take into account fundamental physical limitations as well as the state-of-the-art optical device research. We conclude that, unlike in electronics, large-grained parallelism is not suitable for implementation in optics. We also find that functional languages, rather than logic languages, are better candidates for optics. Finally, we show that implementing an optical symbolic processor warrants the need for a real, or at least an emulated, addressable memory in optics.

INTRODUCTION

There has been very little research in developing optical architectures for general purpose processors, and even less for symbolic processors. Therefore, we chose a two-step approach for meeting our long-term goal to design an optical processor for real-time symbolic computing. First, we abstracted the implementation requirements for existing AI languages by examining their execution models. Second, we determined suitable optical computing architectures, based on the state-of-the-art optics technology, that would match these requirements. A number of researchers, however, have suggested that the parallelism of optics might offer a significant advantage for implementing AI architectures [2-9]. Ward and Kottas [9] have even proposed a hybrid optical design to this end. However, to understand the difficulty of implementing purely optical processors, one must first understand the unique advantages and disadvantages that using optics presents.

Optics and optical systems have several characteristics [2, 4] which make them advantageous over electronics.

- Optics can perform very well pattern matching and correlations, operations used frequently in unification processes in logic programming.
- Optical systems can provide very high space-bandwidth and time-bandwidth, thus offering potential for high throughput.
- Optical processors are inherently two-dimensional and parallel.
- Optical signals can propagate through each other with essentially no interaction and crosstalk, a feat that electronics cannot match.

While optics can provide a quantum leap over electronics in high-speed communications and in potentially massive parallelism, there are certain important disadvantages [4] that must be kept in mind.

- Because digital optical devices are a new field of research, the state-of-the-art is a far cry from the levels of complexity and integration seen in today's electronics.
- Implementing optical logic is difficult because materials which provide the essential nonlinearities are only beginning to be developed. Optical nonlinearities are small compared to their electronic counterparts. This means that at present optical devices consume more power than functionally similar electronic devices.
- To date, the concepts of optically addressable memories (where addresses are decoded optically), or optical storage materials as in the case of semiconductors, have not been developed. The use of electronic or electronically addressable memories means slow electro-photonic and photo-electronic conversions.

An important guideline, based on the properties of optical devices, for designing optical architectures is to avoid mimicking electronic designs to achieve the same or better throughput. To compete with electronics, one must somehow exploit the parallelism and non-planar propagation capability of optics [4].

In our architectural analysis we examined both logic and functional symbolic processing languages representative of languages typically used to write AI applications. Among logic languages our focus has been on a parallel logic language, PARLOG [11], which is a derivative of Prolog and Guarded Horn Clauses. However, the essential conclusions for PARLOG apply equally well to PROLOG. We emphasized PARLOG because unlike the more popular PROLOG, the PARLOG computational model was inherently parallel. We thought that the parallelism of this computational model would be an ideal vehicle to exploit the parallelism of optics. We considered pure functional languages because the computational models for

such languages have the potential for a high degree of parallelism.

In searching for the ideal optical architecture, we have looked for

- the appropriate symbolic processing language,
- the appropriate computational model of the chosen language, and
- the optical computing techniques most suitable to support the execution of the computational model.

While the steps involved in this approach look deceivingly sequential, they are really synergistic. Thus, the fundamental limits on optics and implementation technology determine the language and its computational model. The computational model in turn drives the next-generation of optics device research.

The remaining sections of this paper are organized as follows. Section 2 provides a brief outline of languages for AI applications. Both logic programming and functional languages are emphasized. A very brief survey of computational models for these languages is provided in Section 3. Section 4 outlines the building blocks that can be used in optical computing. The issues for implementing the optical computing architectures for the AI languages are dealt with in Section 5. Section 6 outlines the conclusions regarding optical symbolic architectures, and the recomendations that we make for further investigations into implementing optical processors.

LANGUAGES FOR AI APPLICATIONS

In this section, we discuss the types of languages available for implementing expert system shells. An expert system shell (e.g., KEE, ART, or LOOPS) is a tool that facilitates the development of expert systems. It provides facilities such as forward chaining (data-driven reasoning), backward chaining (goal-driven reasoning), procedural computation, object-oriented knowledge representation, evidential reasoning, models of time and hypothetical worlds, belief maintenance, nonmonotonic reasoning, and explanation facilities.

Languages for implementing expert system shells can be divided into three categories: imperative, functional, and logic. A fourth category combines logic and functional languages, but such languages are still under development, and so we excluded them from consideration.

Imperative languages

Imperative languages, particularly LISP (as it is typically used today), form the basis of current expert system shells. However, they permit uncontrolled side effects via the assignment operation. The presence of side effects makes it very difficult to exploit parallelism in such languages. Moreover, due to the assignment operation, the execution of an imperative language program can be viewed as a series of changes to a large state space. Finally, the implementation of such languages requires the use of stacks. As will be seen in a Section 5, the large size of the state space and the need for stacks do not augur well for the optical implementation of imperative languages.

Functional languages

Programs in functional languages are essentially definitions and applications of functions. There is no notion of operations on named objects, and therefore there are no side effects [12]. Examples of functional languages include pure LISP, FP, and data flow languages such as VAL and Id.

A functional program is usually expressed in terms of S-expressions (S for symbolic). S-expressions are defined as being either elementary items or atoms, or recursively defined as a sequence of S-expressions. The following are examples of S-expressions.

(EXP p q) representing the formula p^q
(ADD (MUL 2 x) 1) representing the formula $(2x + 1)$

The absence of side effects render functional languages more suitable than imperative languages for implementations exploiting parallelism.

Logic languages

Programs in most logic languages are composed of Horn clauses, which have the form

$$\text{head} \leftarrow \text{body}$$

where head is zero or an atomic formula (predicates with arguments supplied) and body is a conjunction of zero or more atomic formulas. The logical interpretation of a Horn clause is that the body implies the head. For example, the Horn clause

$$a(X, Y) \text{ :- } b(X, Z),\ c(Z, Y)$$

means that for all X, Y, and Z, b(X, Z) and c(Z, Y) imply a(X,Y). An empty Horn clause body is considered true. Therefore, a Horn clause with an empty body states the head is always true. Such a Horn clause is called a fact. Facts can be written with no implication sign. For example, parent(a, b), means that a is the parent of b. An empty Horn clause head is considered false. Therefore, a Horn clause with an empty head states that the conjunction of atomic formulas in the clause's body is false. This refutation of the body can be used to initiate a resolution-based proof that the body is in fact true. In the course of this proof, all variable instantiations that make the body true can be discovered. A Horn clause with no head is therefore called a goal or query.

While the best known logic programming language is Prolog, its sequential semantics render it inherently unsuitable for parallel processing. The semantics of Prolog are defined in terms of a sequential execution model. In this model, the order of the clauses in a Prolog "database" is significant; the database is scanned sequentially from top to bottom when attempting to satisfy a goal. Within a clause, the atomic formulas in the body are satisfied from left to right in order. Finally, the cut operator, when executed, prevents searching for later clauses to satisfy the goal that the current clause is attempting to satisfy.

Concurrent logic programming languages, e.g., Concurrent Prolog, PARLOG [11], and Guarded Horn Clauses, alleviate the problems posed by the sequential semantics of Prolog. Here, atomic formulas are executed as processes. A clause represents the expansion of a process (the predicate in the consequent) into a set of processes (the predicates in the body). Processes communicate with each other via shared variables. Synchronization mechanisms are provided to delay the consumer process when it attempts to reference a variable that the producer process has not yet bound. A goal is evaluated by checking the multiple clauses in parallel for applicability and non-deterministically choosing one of them. The atomic formula in the body are executed in parallel, as concurrent processes, with the shared variables acting as communication channels.

COMPUTATIONAL MODELS FOR AI LANGUAGES

To develop optical implementation techniques for logic and functional languages, we need to understand their operational semantics. With this in mind, we examined the computational models for these languages. In the logic language category, we concentrated on PARLOG, the language we

identified [1] as best meeting implementation requirements for high-performance expert system shells. This investigation will enable us to identify issues involved in the optical implementation of other logic programming languages as well. In case of functional languages, our focus was on those, such as pure LISP or SASL, that exhibited no side effects. Based on the study of the computational models, we summarized the computing requirements necessary for optical implementations.

Computational models for logic languages (PARLOG)

The operational semantics of PARLOG are best understood in terms of the abstract AND/OR process model [1]. In this model, a process is created for evaluating literals and for searching for a candidate clause during evaluation of a literal. The state of a PARLOG evaluation is represented by a process structure called the AND/OR process tree. The nodes in this tree are processes. The leaf processes are either runnable or suspended on some variable. The non-leaf processes are not runnable. They await results from their child processes. There are two types of non-leaf processes: AND processes and OR processes. A process assumes a type AND if it is to evaluate a conjunction of literals. A process assumes a type OR if it is to search for a candidate clause among the clauses defining a relation. A PARLOG query is evaluated by first searching for a candidate clause and then non-deterministically committing to one such clause. Upon committment, the literals in the body of the chosen clause are evaluated. During query evaluation, the AND/OR process tree grows and shrinks dynamically.

A graph reduction computational model based on the abstract AND/OR process model and a parallel abstract machine has been designed for PARLOG. This abstract machine is targeted towards electronic implementations [1]. This machine is a loosely coupled multiprocessor. Each processing element (PE) is a collection of computing agents that perform dedicated functions such as process tree growth, process tree management, and unification. The PARLOG data objects (or terms) are represented as directed acyclic graphs (DAGs) in this machine. Data objects and the AND/OR processes are distributed among the various PEs. However, the machine has a single virtual address space. This means that the DAGs and the process tree are linked across PEs. This linkage will be seen to have important consequences for optical implementation of PARLOG.

Computational models for functional languages

There are basically two computational models for functional languages: data flow and reduction. In a data flow model, the program is compiled into a graph representing the data dependencies. The nodes of such a graph are referred to as operators. They represent function applications, while the edges reflect the composition of the functions. The data flow graph is executed directly; an operator "fires" whenever its input arguments are present, sending any output to its direct descendants.

In a reduction model, the program is viewed as a set of rewrite rules. The left hand side of each rule corresponds to a function specification; the right hand side, the function definition. In order to evaluate a function, first a directed graph that captures the rewrite information, is built up. The nodes in this graph correspond to functions. The immediate descendants of a node correspond to the function's definition. The computation can proceed in either a demand driven or an eager manner. After a function is evaluated, it is replaced by its value, hence the name reduction. Eventually, the whole graph will be replaced by one value.

Reduction comes in two varieties: string reduction and graph reduction. In string reduction, every occurrence of a variable is treated as a distinct copy, while in graph reduction, all occurrences share the same copy.

A technique called combinator reduction is often used for implementing functional languages efficiently. In this technique, variables occurring in a function definition are "abstracted out" to produce a function definition consisting solely of operators called combinators. Combinators are higher order functions, that is, they can accept functions as arguments and return functions as results. The so-called S, K, and I combinators are sufficient to remove all variables from any function definition. Combinator reduction involves two steps. First, the program is transformed into combinator expressions (containing no variables). Second, these expressions are reduced as dictated by the definitions of the combinators.

Computing requirements to support computational models

An examination of the computation models, especially those that use graph reduction and are of interest to us, reveals that most computing requirements revolve around manipulating data structures such as graphs or lists. Representations in memory or an emulation of such data structures must therefore be efficient. Other operations that should be well supported are arithmetic and logic primitives, comparison or matching operations, efficient ways of traversing graphs and lists, and dynamic allocation of memory.

In the next section, we examine the issues involved in implementing logic languages and functional languages using optics.

COMPUTING PRIMITIVES IN OPTICS

Our approach to designing symbolic optical processors consisted of two steps. First, examining the implementation requirements of computational models of symbolic processing languages. Second, determining suitable optical computing architectures that best matches these requirements. The results of the first step were described in the previous section. In this section we present the analyses of the second step.

An examination of optical computing primitives and the fundamental and practical limitations of optics provides us with clues as to how well the computational models outlined earlier can be supported. The results of such an exercise will also influence the optical architectures that can be designed.

Desirable optical features include parallelism in implementation, parallelism in primitive operation such as parallel write and read, high-speed interconnects, large fanin and fanout compared to electronics, and data representation in more than one dimension, such as arrays.

On the other hand, proposed optical computing schemes have certain limitations. First, traditional I/O methods cause bottlenecks at the electro-optic interface due to the need for different symbol representation in optical and electronic computers [4]. In addition, electron-photon conversion requires excess power.

Second, use of discrete optical gates to implement the processing elements (PEs) is not feasible since it would result in increased complexity, size, and poor performance. Use of optical computing configurations such as optical finite state machines (OFSMs) (Figure 1) may be more attractive [2].

Our examination of the computational models revealed that data representation in optical memory is of paramount importance in designing the architecture. Most operations in symbolic processing require manipulating and performing macro-operations on data or memory elements. These include simple arithmetic and logic operations, input matching and unification, accessing elements of a data structure such as a tree

or graph or list, and dynamically allocating new memory cells. How memory structures are implemented and data represented is critical.

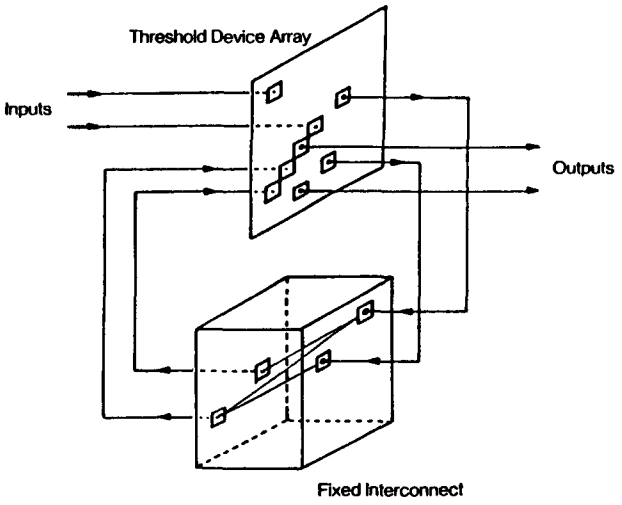

Figure 1. Functional block diagram of the Optical Finite State Machine (OFSM) [2]. The 2-D array and the interconnect act as the memory and combinatorial logic, respectively, of the OFSM.

The following subsections discuss the representations of data in optics. However, the representation of data cannot be independent of the optical computing structures or the memory implementations. For this reason, we examine data representation for proposed optical computing architectures.

Data representation in optics

In signal processing, traditional optical computing uses analog data representation. However, analog representations suffer from noise problems. Because of the requirement of precise data in programming languages, digital representation is preferred. For example, two-dimensional arrays of bit pixels as used in OFSMs [2, 4] would be more suited for representing complex data structures such as DAGs.

In Sections 2 and 3, we described how typical low-level operations in symbolic processing require manipulations of structured data such as graphs or lists. In conventional computers, such complex data structures are implemented using pointers. This approach to the representation of complex data structures is attractive because it allows complicated relationships to be stored without having to be specified at the time the program was developed. It does, however, require that the machine possess addressable memory. This requirement can be fulfilled in two ways: by developing another memory structure or by developing a way to perform addressable memory with optical devices.

Next we examine optical computing architectures and techniques for data representation. The architectural analysis presented later will be based on how these optical computing structures can be used to support the computational models described earlier.

Two optical computing architectures, optical finite state machine (OFSM) and symbolic substitution, are discussed in this subsection. The emphasis is on how to implement memory structures to represent data in each architecture.

One possible memory structure is the OFSM. Unlike conventional electronic computers, the OFSM memory is not separated from the processor. OFSM computing systems consisting of parallel planes with 1000 × 1000 optical gates have been proposed. The gates perform the logic operations of the finite state machine [2].

However, by using this type of structure, it is difficult to design a useful computer. The conventional way to design a finite state machine is to enumerate all the possible inputs, ouputs, and next states, and then design some combinatorial logic to perform that function. However, the design of a computer in this manner is practically impossible. Such an effort would involve specifying all of the possible data structures, all the values of the data items, and the answers at the time the machine is designed.

Another way to structure memory is to avoid explicitly specifying the transition rules. Unfortunately, with this approach, called symbolic substitution (Figure 2) [3, 4], the machine is no longer massively interconnected, although it can utilize the parallelism of optics [3]. The lower degree of interconnection limits the speed at which a computation can occur, since many cycles will be required to transfer data around the plane.

Symbolic substitution, however, is easily implementable [2, 3] and may be able to employ high-speed (gigabit) optical components [10], making it a candidate worth investigating to determine if it can perform the primitive functions that computational models require.

Matrices [6, 9] are a another way to represent data or memory structures. Graph structures can be represented in a matrix structure by assigning nodes of the graph to rows and columns. When there is a connection between nodes, an entry is made at the intersections of rows and columns of the two elements. A directed graph may be represented by using the

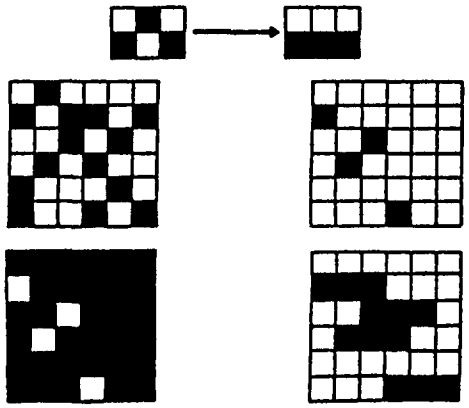

Figure 2. Symbolic Substitution usihg parallel symbolic recognition and substitution [4]. The rule (top) is applied to the input (top left) to produce the output (bottom right). Intermediate arrays represent steps in recognition of the rule pattern.

rows to indicate the node the connection is from and the columns to indicate the node that is the destination. In this scheme, however, the memory is used very inefficiently since most graphs are not fully connected; only a few connections are made between nodes, while there is space for any possible connections.

One advantage of the matrix representation scheme is that no addressing is required to check interconnections between data items; it is all present in the matrix. To set up the connections, however, some means are required to set the elements of the matrix. This is made even more difficult when the elements to be added to the existing matrix constitute another graph. Additions to the graph are necessary in logic languages since they feature partially instantiated data structures. The graph representation must be modified when rows and columns are added to the existing graph. Modifications are also necessary when elements are to be removed from the graph. In either case, means to keep track of where the rows and columns are to be added or deleted are required. The answer is to use pointer structures or indirect addressing. Thus to perform nontrivial operations on data stored in matrix format in logic languages, some form of location-based addressing must be used.

There is also the direct approach to implementing location addressable memory. This requires constructing memory which has binary addresses. With this approach it is difficult to generate the decoding addresses. While an address decoder has been designed at Honeywell, it does not appear that its inclusion as part of a memory would significantly improve overall system performance as compared to pipelined high-speed electronic memory.

In summary, it is possible to do most of the primitive functions required for symbolic computing in optics. However, it is unclear whether optics may have any clear advantage over electronics without examining implementations of memory in optics.

ISSUES IN OPTICAL IMPLEMENTATION OF EXPERT SYSTEM LANGUAGES

A review of the computational models for AI languages, and the computing primitives of optics provides the foundation for selecting appropriate architectures for an optical symbolic processor. Computational models for both logic and functional languages were considered. In this section, the critical issues of implementing either type of language, as well as the general issues of parallel implementation in optics are discussed.

Issues in the optical implementation of PARLOG

Based on the AND/OR graph reduction computational model [1] and the optical primitives and computing structures, we defined a broad optical architecture for PARLOG. This architecture corresponds to a distributed architecture that consists of multiple PEs connected point-to-point in optics, with both shared and dedicated memories. The shared memory must contain the AND/OR process descriptors since many AND/OR processes can be simultaneously evaluated by many PEs. Further, the shared memory must also contain terms that are constructed during evaluation of queries. The dedicated memory of each PE contains i) the complete compiled program, and ii) the template data objects that are used for matching or unification during runtime. Thus, the shared memory contains only those data objects that are constructed in runtime, while the dedicated memory contains all data objects known at compile time. By using shared memory for runtime-generated objects, the problem of linking different DAGs or subDAGs of a DAG across PEs can be avoided.

The PEs in the parallel abstract machine for PARLOG consist of different agents [1] that are responsible for dedicated functions required to execute the single-solution PARLOG programs. In optics such agents are best realized in terms of a cluster of OFSMs that execute the algorithms that comprise the function of the agent. The PEs, as well as the agents, communicate via messages. Different message types can be recognized by the use of a set-associative pattern matching on the message type. Since OFSMs are limited in complexity, all control operations such as logic and arithmetic operations, are done external to the basic finite state machine.

The crucial design issue in the optical architecture, however, is the data representation of the DAGs. In Section 4 we examined this issue, and concluded that representing DAGs and lists is difficult in optics. In the case of PARLOG, an added complexity is introduced by partially instantiated data structures, which may grow as processes execute, thus requiring the use of pointers. Furthermore, several processes may be suspended on some variable in a partially instantiated data structure. To avoid busy waiting, uninstantiated variables should have a demand list associated with them. This is a list of all processes that suspend on the variable. Demand lists necessitate pointer structures since their size is not known at compile time.

Another problem in distributed architectures is context switching, which is discussed below in the next sub-section.

Issues in the optical implementation of functional programming languages

One of the major disadvantages of logic programming execution in optics is the presence of partially instantiated variables. Functional languages, on the other hand, do not feature such data structures. In functional languages, a data structure must be fully instantiated before a function application can begin. Both data flow and reduction are suitable computational models for functional languages.

In gross terms, there are no differences in the complexity of implementing a distributed architecture for data flow or graph reduction. With both data flow and graph reduction, large data structures must be manipulated. In electronics, this is handled by using I-structures [21]. These data structures, however, require pointers and demand lists. To avoid extensive use of pointers in optical data flow implementations, shared structure memory is preferred. While data-driven graph reduction may not provide any advantage over data flow architectures, normal order graph reduction in optics may require less computation, thus making graph reduction more attractive than data flow.

Normal order graph reduction [13-18] is the reduction computational model type that appears most promising for parallelism. In this type of graph reduction, each step is an atomic step in which the graph is mutated in a manner consistent with the reduction rule of the corresponding combinator. An example of such graph mutation is illustrated in Figure 3.

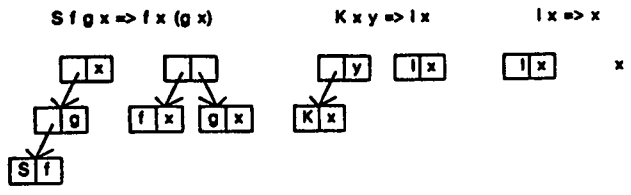

Figure 3. Combinator Graph Reduction: graph mutations on applying the S, K, and I combinators. f and g represent functions while x and y represent arguments.

In a distributed system, where the graph is distributed in a network of PEs, a message-passing strategy will allow each reduction to occur in piecemeal fashion [18]. The graph reduction evaluation model appears very well suited to parallel computing at a medium-level granularity. This level corresponds to evaluating reducible expressions (redexes) in parallel. Thus individual redexes that are available can be evaluated in parallel by different PEs.

One critical design issue in an optical implementation is how to exploit the parallelism of combinator graph reduction. Since the PE responsible for the combinator application is a simple combinational function, an OFSM implementation is not necessary. However, since the argument of the combinators can be a data structure, a list in the general case, of any size, the transformations may be difficult to handle if the data is moved every time into different nodes. In the electronic case, pointers can be used very conveniently without actual movement of data. Thus, unless the data is of simple structure, using pointers is essential.

Another instance in graph reduction where pointer structures are necessary is in the evaluation of common subexpressions. To avoid wasted computation, common expressions are shared in graph reduction (unlike in string reduction, which is similar in all other respects to graph reduction). However, the use of shared expressions in combinator graph reduction implies using indirection to ensure that argument values are not lost before all expressions involving the subexpression have been evaluated [11]. Given these issues, it would therefore appear more attractive to examine non-distributed architectures, where graph mutations are managed in a shared memory. Such architectures would be designed to exploit low-level parallelism in optics.

Common issues in implementing distributed architectures in optics

While the above issues relate to the specific computational models of symbolic processing languages, there are some aspects of computational support that are common to any multiprocessor architecture. Support for context switching is one such aspect. Context switching requires maintenance of stacks and indirect addressing schemes which are difficult to handle in optics. Another issue is that of communication between processors. If data structures are to be sent from one processor to another, then addresses or pointers will be required, thus necessitating location-based addressing.

CONCLUSIONS: WHAT MAKES SENSE

We have examined different computational models of symbolic processing languages and found that it is not possible to directly exploit existing optical primitives.

Because of the presence of partially instantiated data structures in logic languages and the difficulty in implementing location-based addressing, it is not feasible to implement optical processors for executing logic languages. Based on the comparison of the computational models of logic programming and pure functional languages, however, it is clear that functional programming languages are a better candidate for optics implementation. The computational model that may be most worthwhile investigating is that of normal order combinator graph reduction in a non-distributed architecure targeted towards exploiting low-level parallelism. Such parallelism can be found in processes of small granularity such as parallel word searches, writes or reads.

REFERENCES

[1] Raja Ramnarayan, et al, 'Interim Report on Very Large Parallel Data Flow Program', Honeywell Corporate Systems Development Division, May, 1986.

[2] Alexander A. Sawchuk and Timothy C. Strand, 'Digital Optical Computing', Proceedings of the IEEE, Vol. 72, No. 7, July '84, pp. 758 - 779.

[3] Karl-Heinz Brenner, Alan Huang, Norbert Streibl, 'Digital Optical Computing with Symbolic Substitution', Applied Optics, Vol. 25, September 1986, pp. 3054 - 3060.

[4] Alan Huang, 'Architectural Considerations Involved in the Design of an Optical Digital Computer', Proceedings of the IEEE, Vol. 72, No. 7, July '84, pp. 780 - 786.

[5] Keith B. Jenkins and C. Lee Giles, 'Parallel Processing Paradigms and Optical Computing', SPIE Vol. 625, Optical Computing (1986), pp. 22 - 29.

[6] Rodney A. Schmidt and W. Thomas Cathey, 'Optical Representations for Artificial Intelligence Problems', SPIE Vol. 625, Optical Computing (1986) pp. 226 - 233.

[7] J. Tanida and Y. Ichioka, 'Optical Logic Array Processor using Shadowgrams', Journal of Optical Society of America, Vol. 73, No. 6, June 1983, pp. 800 - 809.

[8] T. K. Gaylord et al, 'Optical Digital Truth Table Look-up Processing', Optical Engineering, January/February 1985, Vol. 24, No. 1.

[9] Cardinal Ward and James Kottas, 'Hybrid Optical Inference Machines: Architectural Considerations', Applied Optics, Vol. 25, March 1986, pp. 940 - 947.

[10] P. W. Smith and W. J. Tomlinson, 'Bistable Optical Devices promise Subpicosecond Switching', IEEE Spectrum, Vol. 18, June 1981, pp. 26 - 33

[11] Keith Clark and Steve Gregory, 'PARLOG: Parallel Programming in Logic', Research Report DOC 84/4, June 1985, Department of Computing, Imperial College of Science and Technology, University of London.

[12] Peter Henderson, 'Functional Programming Application and Implementation', Prentice-Hall International, 1980.

[13] David Turner, 'A New Implementation Technique for Applicative Languages', Software-Practice and Experience, Vol. 9, 1979, pp. 31 - 49.

[14] T.J.W. Clarke et al, 'SKIM - the S, K, I Reduction Machine', Proc. of the 1980 ACM LISP Conference, pp. 128 - 135.

[15] W.R. Stoye et al, 'Some practical methods for Rapid Combinator Reduction', Proc. of the 1984 ACM Symposium on LISP and Functional Languages, pp. 159 - 166.

[16] R.J.M. Hughes,'Super-Combinators', Proc. of the 1982 ACM Symposium on LISP and Functional Languages, pp. 1 - 10.

[17] Simon L. Peyton Jones, 'An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions', Proc. of the 1982 ACM Symposium on LISP and Functional Languages, pp. 150 - 158.

[18] Steven Tighe, 'A Study of Parallelism Inherent in Combinator Reduction' MCC Tech. report PP-140-85, November 1985.

[19] Paul Hudak and Benjamin Goldberg, 'Experiments in Diffused Combinator Reduction', Proc. of the 1984 ACM Symposium on LISP and Functional Languages, pp. 167 - 176.

[20] Paul Hudak and Benjamin Goldberg, 'Distributed Execution of Functional Programs Using Serial Combinators',

IEEE Transactions on Computers, Vol. C-34, No. 10, October 1985, pp. 881 - 891.

[21] Arvind and V. Kathail, 'A Multiple Processor Data Flow Machine that supports Generalized Procedures', Proc. of the 8th Annual Symposium on Computer Architecture, May 1981, pp. 291 - 302.

This research was supported by the Air Force Office of Scientific Research and the Advanced Research Projects Agency of the Department of Defense under Contract No. F49620-86-C-0082.