AFOSR-TR- 86-2213

OPTICAL SYMBOLIC PROCESSOR FOR EXPERT SYSTEM EXECUTION

QUARTERLY TECHNICAL REPORT

June 1, 1986 to August 31, 1986

Sponsored by

Advanced Research Projects Agency (DOD)
ARPA Order No. 5794
Monitored by AFOSR Under Contract #F49620-86-C-0082

Prepared by

Matthew Derstine
Aloke Guha*
Raja Ramnarayan*

Honeywell Physical Sciences Center
10701 Lyndale Avenue South
Bloomington, MN 55420

*Honeywell Corporate Systems Development Division

Approved:

Anis Husain
Section Head

Norman Foss
Department Manager

AD-A174955

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): Honeywell Physical Sciences Center	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 86-2213
6a. NAME OF PERFORMING ORGANIZATION: Honeywell Corporation; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: AFOSR / NE
6c. ADDRESS (City, State and ZIP Code): (Physical Sciences Ctr) 10701 Lyndale Ave., Bloomington, Minnesota	7b. ADDRESS (City, State and ZIP Code): Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: same as 7b; 8b. OFFICE SYMBOL (If applicable): NE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: #F49620-86-C-0082
8c. ADDRESS (City, State and ZIP Code): same as 7b	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2305	B1 (DARPA)	

11. TITLE (Include Security Classification): Optical Symbolic Processor for Expert System Execution

12. PERSONAL AUTHOR(S): Matthew Derstine

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Quarterly Progress	FROM 6/1/86 TO 8/31/86	Nov 86	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

It was found that the computational requirements of logic languages and functional languages are primitive operations which involve manipulation of complex data structures such as graphs and trees, and that the execution of the languages can be described as manipulations of those data structures. The representation of the complex data structures imply that the representations must be exact (digital) and that some means to denote connections between data items, such as pointers, is required. Since the representation between data items is more important than the actual items stored, the most important functions involve the manipulation of the data structures. Examination of the optical architectures available to represent and implement the functions identified showed that some way to perform location addressable memory was needed. One technique, matrix representation, was identified and a technique to construct addressable optical memories was invented. It was found that these methods do not perform computational primitives. It was found functional languages & logic require primitive operations.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	UUUUU

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. C. Lee Giles	202-767-4933	NE

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS

SUMMARY

The goal of the Optical Symbolic Processor for Expert System Execution program is to develop concepts for optical computers which can perform real-time symbolic processing. The program is divided into two sections, architecture development and development of a device for reconfigurable interconnects. In the first quarter of the program, only architecture development work was performed.

The approach for this phase of the program has been to examine computational models of computer languages and determine the primitive operations required. Possible optical implementations of these primitives were then examined and evaluated. In general, a top down approach was taken with the goal of a direct optical implementation of the desired primitive operations.

It was found that the computational requirements of logic languages and functional languages [Section III] are primitive operations which involve manipulation of complex data structures such as graphs and trees, and that the execution of the languages can be described as manipulations of those data structures. The representation of the complex data structures imply that the representations must be exact (digital) and that some means to denote connections between data items, such as pointers, is required. Since the representation between data items is more important than the actual items stored, the most important functions involve the manipulation of the data structures.

Examination of the optical architectures available [Section IV] to represent and implement the functions identified showed that some way to perform location addressable memory was needed. One technique, matrix representation, was identified and a technique to construct addressable optical memories was invented. By examination of a possible architecture, it was found [Section V], however, that these methods do not adequately perform the computational primitives. Moreover, it was found that while functional languages and logic languages require similar primitive operations, implementation of logic languages in parallel optical environments is more difficult.

In the next phase of the research, optical architectures will be examined to determine how they can perform the required functions through combinations of lower level operations.

I. INTRODUCTION

One long-term goal of the Optical Symbolic Processor for Expert System Execution program is to develop a computer for real-time symbolic computing. The chosen approach was to investigate concepts for the implementation of an existing computer language on an optical computer architecture. Many researchers have suggested that the parallelism of optics might be exploited for symbolic processing applications.[1-8] These proposals have been made because of the ease with which optics performs correlations and because of the large communications bandwidth of two-dimensional interconnects. This report describes the first examination of how the optics could be used within the framework of implementing a well-specified parallel language, PARLOG[9], on a parallel optical computer. It also suggests other types of languages which may be better suited to optical computer architectures.

PARLOG was initially selected because of the belief that the large amount of matching and searching required in logic languages would lend itself to optical computing. This report examines how logic languages, of which PARLOG is one, and functional languages could be implemented on an optical computer. The approach taken was to examine the problem from the top, that is, examine how each of the required functions can be directly implemented in an optical computing structure. This report documents functions and capabilities needed to develop a symbolic optical computer architecture.

Section II of the report examines the languages which could be used for expert system development. The third section describes the computational model of PARLOG and the computational primitives required for both PARLOG and functional languages. The fourth section describes how optics can address some of the functions required for symbolic processing. The final section describes our conclusions for the period June 1 - August 31 and future plans.

II. LANGUAGES FOR EXPERT SYSTEM SHELLS

An expert system shell (e.g., KEE, ART, 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 exclude them from consideration here.

Imperative languages, particularly LISP (as it is 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. The impact of this is that the implementation of such languages requires the use of stacks. As will be seen in a later section, the large size of the state space and the need for stacks do not augur well for the optical implementation of imperative 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. Examples of functional languages include pure LISP, FP, and dataflow languages such as VAL and Id. LISP, as it is generally used today in expert systems and other applications, is an imperative rather than a functional language; great use is made of side effects. Computing by effect implies incremental changes made to variables by successive assignment. Pure functional languages[10], such as pure LISP, thus compute by value and not by effect, and functions are used to compute new values from old.

The best-known logic programming language is Prolog, which has received widespread attention as a result of the Japanese Fifth Generation effort. The

problem with Prolog is its sequential semantics, which 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, 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 formulas in the body are executed in parallel, as concurrent processes, with the shared variables acting as communication channels.

As part of another effort at CSDD, PARLOG was identified as the language that best meets the requirements for implementing high performance expert system shells. Therefore, as far as logic programming languages go, we will restrict ourselves to investigating optical implementations for PARLOG. This investigation will enable us to identify issues involved in the optical implementation of other logic programming languages as well.

In order to develop techniques for the optical implementation of logic languages and functional languages, we need to understand their operational semantics. With this in mind, we briefly describe computational models for these languages in the next section.

III. COMPUTATIONAL MODELS FOR EXPERT SYSTEM LANGUAGES

The operational semantics of PARLOG are best understood in terms of the AND/OR process model. 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 commitment, the literals in the body of the chosen clause are evaluated. During query evaluation, the AND/OR process tree grows and shrinks dynamically.

As part of another effort at CSDD, a parallel abstract machine has been designed for PARLOG. 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.

There are basically two computational models for functional languages: dataflow and reduction. In a dataflow 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 dataflow 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, to 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[11] 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.

In summary, the basic functions required for PARLOG and for functional languages are similar. Both can be expressed in terms of graph reduction models. This implies that the representation of the connections between the data is more important than the actual data items. This has important consequences for implementation of these languages. Most importantly, it requires the storage of the data structures to be exact. This is because errors in representation of the data can easily and completely destroy its meaning.

Exact representation of data structures can be most easily accomplished with digital representation. Analog representation could be employed if the probability of error was sufficiently low, but in practice, digital systems

are the only choice. This choice will then limit the types of optical computing structures to those which represent the data in digital form. This does not, however, imply that all of the computation must necessarily be digital. Very low level (node-to-node) matching might be able to use analog methods, but manipulation of the data structures must use digital computation. Nevertheless, since the most important and time consuming part of the task is the data structure manipulation, the use of analog optical structures to do the matching may not provide any performance increase.

Another implication of the representation of the data by graphs and trees is the need for some way to express the connections. Traditional computer designs handle this problem through the use of pointers. Pointers are typically addresses of locations where other data items are stored. This approach to the representation of complex data structures is attractive because it allows complicated relationships to be efficiently stored without having to be specified at the time the program was developed. It does, however, require that the machine possess addressable memory.

In the next section, we will examine the primitive operations readily available in optical computing. Specifically, we examine digital optical architectures with the goal of expressing and manipulating data structures.

IV. PRIMITIVES IN OPTICAL COMPUTING

An examination of the optical computing primitives, and the fundamental and practical limitations of optics will provide us with data as to how well various computational models can be supported and what changes must be made to make them amenable to optical computation.

The features of optics that we would like to exploit are its parallelism, parallel write and read capability, high-bandwidth interconnects, and data representation in more than one dimension, such as arrays.

The limitations of many proposed optical computing schemes that need to be avoided are the method of input and output to the computer and the need for photo-electronic and electro-photonic conversions. These are typical bottlenecks in hybrid computing systems since indiscriminate conversion of optical to electrical signals results in a degradation of performance due to the need for conversion of the data representation in each system and power/speed considerations. These inefficiencies occur because symbol representations in optical and electronic computers can be expected to be different, and electron-photon conversion requires excess power.

Although traditional optical computing in signal processing uses analog data representation, because of noise problems and the requirement of precise representation of data structures in expert systems, digital representation is preferred.

All approaches to provide the primitive operations required (Section II, III) must take into account the overall nature of the task. From the optical device point of view, these primitive operations are the macro-functions which must be performed. All of the macro-functions operate on data structures, not on simple data items. Moreover, makeup of these data structures are not known until the macro-function is executing. The required operations cannot be

performed by operations like simple correlations since the structure of the data must be examined.

Operations like searching and matching of digital data items could, perhaps, still be performed using correlations. However, since the macro-functions are all manipulations on data structures, correlations cannot be employed unless the data structure can be represented as an entity rather than as items connected together. At present, this type of representation is difficult to achieve in an optical computer because the data structures change, requiring a means for selecting, adding, deleting, splitting, and joining.

The first item that must be investigated in developing ways to perform the macro-functions is the representation of the graphs and lists using optical architectures. As stated previously, this type of representation typically requires the use of pointers and location addressable memory. This need for addressable memory can be tackled in two ways: by developing another type of memory structure or by developing a way to implement addressable memory with optical devices.

The first method we examined to implement a different memory and computing structure is the optical finite state machine (OFSM).[1,2] The feature of this architecture which is unlike conventional electronic computers is that the memory is not separated from the processor. Computing systems have been proposed which are composed of parallel planes of 1000x1000 optical gates which perform the logic operations of the finite state machine.[2]

The conventional way to design a finite state machine is to enumerate all the possible inputs, outputs, and next states, and then develop some combinatorial logic to perform that function. However, design of a system with over a trillion (1×10^{12}) states is practically impossible when done in this manner. Such an effort would be tantamount to specifying all of the possible data structures and all the values of the data items at the time the machine is designed. It also would require specifying the answers for each possible case, in other words, enumerating all of the answers for all of the possible computations before the machine is ever constructed.

The other approach to developing a finite state machine would be to specify the transition rules for the states in such a way as to avoid specifying all of them explicitly. Symbolic Substitution is such a method[2], but it has the disadvantage that the machine is no longer massively interconnected. Only pixels within a certain neighborhood can communicate directly. This will eventually limit the speed at which a computation can occur, since many cycles will be required to transfer data around the plane.

Symbolic Substitution does, however, have the advantage of being easily implemented[2] and may be able to employ high-speed (gigabit) optical components.[8] We will be investigating Symbolic Substitution in the future to determine if it can perform the required primitive and macro-functions.

Another method for representing data structures is the use of adjacency matrices[4, 7]. 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 rows to

indicate the node the connection is from and the columns to indicate the node that is the destination. Figure 1 illustrates the case of a simple directed graph. This scheme has the disadvantage that memory is used very inefficiently; only a few connections are made between nodes, while there is memory allocated for any of the possible connections.

In this scheme, 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 is required to address and set/reset the elements of the matrix. This is made even more difficult when the elements to be added to the existing matrix make up another graph. The new subgraph must be rearranged to be added as rows and columns to the existing graph. If elements were to be removed from the graph, some means would be needed either to keep track of the empty rows and columns or to rearrange the graph so that the empty rows and columns are no longer in the interior of the data structure. Both of these methods require other data structures, such as linked lists, to keep track of the altered data. Thus, to perform nontrivial operations on data stored in matrix format, some form of addressing must be used at some point.

One possible way to use a matrix memory would be to use masks stored in a holographic memory.[6] Only a few masks could be stored in this way, but it might be possible to set up primitive select and add operations. Such a memory system would be controlled by an OFSM acting as a sequencer. The holographic memory, the OFSM, and the matrix memory would make up a special kind of addressable memory in that it would be optimized to store data structures rather than just data items.

The other solution to location addressable memory is to actually construct memory which has binary addresses. The problem with this approach has been the difficulty in generating the decoding addresses. Figure 2 shows a new concept for an address decoder. This device makes up one stage of the address decoder and consists of a phase conjugate mirror combined with two switchable absorbers(SA1 and SA2). At the beginning of the decoding process, a single beam is incident upon the phase conjugate mirror. If SA1 is open, the beam passes to output 1 and if SA2 is open, the beam passes to output 2. By connecting the switchable absorbers to one of the address lines so that SA1 is open while SA2 is closed and vice versa, the device becomes a one of two selector. Since the device is not limited to the position of the input beam, it can be cascaded. Thus, at the input to the second stage, the beam could be at one of two possible positions. Then after going through one or the other switchable absorber, the beam is at one of four positions. By cascading four stages, the device would be a 1 of 16 selector, which is just what is required to address 1 of 16 memory locations. Similarly, eight stages could be cascaded to form a selector for 256 locations, and if a square geometry is used, two eight-stage decoders could decode a 64k by 1 memory. Such a geometry would require arrays of memory elements like edge-addressed bistable devices demonstrated in InSb.[12]

In summary, it is possible for optics to perform most of the memory functions needed for symbolic computing, but it is unclear that optics has any clear advantage over electronics. In the next section, we will examine the issues involved in implementing logic languages and functional languages using optical primitives by examining an example optical computer architecture.

V. ISSUES IN OPTICAL IMPLEMENTATION OF EXPERT SYSTEM LANGUAGES

Issues in optical implementation of PARLOG

Based on the AND/OR graph reduction computational model [9], which appears most appropriate for PARLOG, we defined a broad optical architecture for PARLOG. This corresponds to a distributed architecture consisting 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. Furthermore, the shared memory must also contain the terms that are constructed during evaluation of queries. The dedicated memory of each PE contains the complete compiled program and 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 that are known at compile time. By using shared memory for runtime-generated objects, the problem of linking different DAGs or subDAGs can be avoided.

The PEs in the parallel abstract machine for PARLOG consist of different agents. In optics, such agents are best realized in terms of a cluster of OFSMs which executes the algorithms comprising 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, matching, and arithmetic operations, are done external to the basic finite state machine. Another approach would be to use OSFMs which employ Symbolic Substitution and perform all of the logic, arithmetic, and matching functions.

The crucial design of the optical architecture, however, is the data representation of the DAGs. Given the severe limitations of optical memory, linked lists structures cannot be implemented as easily as in electronics. The alternative solution is to represent the DAG as an adjacency matrix (not as an adjacency list since that requires linked lists). However, although space inefficient adjacency matrices can be conveniently represented as two-dimensional arrays, they only contain the topology information and not information about the node value itself. More important than the necessity to represent values of the nodes in the DAG is the fact that a node in a DAG may be uninstantiated, i.e., it possesses no value at all and possibly waits to be unified with some other variable not yet evaluated. Such a situation may require waiting in a demand list of another variable to avoid busy waiting on a variable by a PE. Moreover, when the node does become instantiated, it may be instantiated with some structure already present in the memory. This would then require that the matrix memory be modified to point to the data structure and that the connections to the demand list be removed. This removal would then require either the rearrangement of the total memory or the addition of the freed up space to some list. In any case, representing uninstantiated variables, common to logic programming languages, implies extensive use of pointer structures and location addressable memory. Similarly, describing the processes and the variables in the descriptor and term memory, respectively, also requires manipulating pointers and list structures.

Because of the difficulty in feasibly implementing location-based addressing by using OFSMs and separate memory structures, it becomes untenable to directly implement optical processors for executing logic programming languages.

Issues in optical implementation of functional programming languages

One of the major disadvantages of logic programming language execution in optics was the presence of uninstantiated variables. Functional languages, on the other hand, do not present this problem since in their computational model, all variables are instantiated[10]. To evaluate functional languages for optical implementations, both reduction and dataflow models were used since both have been implemented in electronics for high performance. The data flow computational model is considered first followed by a discussion on reduction.

A dataflow machine [13] requires multiple processors to operate on different portions of the dataflow graph if and when input arguments for different operators become available. This requires each PE to maintain the complete graph and possess the capability of recognizing input arguments and their context with the help of tags. Since an operation may require multiple arguments, a PE in the multiprocessor architecture has to maintain a tag-matching unit while waiting for all arguments of an operation. The typical topology of the architecture is an nxn routing network connecting the PEs [13].

The level of granularity of the parallelism, from the point of view of optics, is high since each PE, implemented in terms of OFSMs, has to be cognizant of the complete dataflow graph. However, the overhead at the level of data object management is also substantial since tags must be generated, maintained, and matched at runtime. In a broad sense, the complexity of the optical architecture for this computational mode will be as complex as that in a logic language except that the variables will not be uninstantiated during evaluation.

The next computational model considered will be that of reduction. The type that appears most promising for parallelism is that of combinator graph reduction.[11] In combinator 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. 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 [14]. 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.

The critical design challenge in optics is how to realize the combinator reduction process in parallel. 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 becomes attractive. Another instance 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, 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 [15]. Given these issues, it would therefore appear more attractive to examine a non-distributed architectures, where graph mutations are managed in a common memory. Such architectures would be designed to exploit low-level parallelism in optics.

General issues in implementing parallel 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 architectures. [An important issue that is difficult to address in optics is that a multiprocessor architecture has a finite number of processors but may have many more processes created.] This implies that some processor will have to be responsible for the execution of more than one process. Handling more than one process is equivalent to handling more than one context and requires context switching. Context switching, in turn, requires maintenance of stacks and other indirect addressing schemes which are difficult to implement without location addressable memory. Consequently, context switching and recursion in general are difficult to implement in optics.

VI. CONCLUSIONS AND PLANS

We have examined different computational models for languages for expert systems and found that it is not possible to directly exploit optical primitives. 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 SKI combinator graph reduction in a non-distributed architecture targeted towards exploiting low-level parallelism.

Examination of the models has also elucidated the need for a way to represent complicated relationships between data items, specifically, the need for pointers. Since optical memories are not typically addressable, this requirement is not easily filled. Future investigation will center on the examination of proposed optical computing architectures to determine if they can be modified to perform data handling operations. Specifically, we will be investigating symbolic substitution[2] to see if it can exploit non-distributed, low-level parallelism. We have selected symbolic substitution because it does not require space variant interconnects, since other Honeywell-funded results[16] indicate optics cannot achieve general interconnects of the density required for non-distributed architectures.

In summary, the efforts of the second quarter will be to examine, from the bottom up, how optics can provide the identified macro-functions and determine if they can be performed with enough speed to provide for real-time symbolic processing.

VII. REFERENCES

[1] Alexander A. Sawchuk and Timothy C. Strand, 'Digital Optical Computing', Proceedings of the IEEE, Vol. 72, No. 7, July 1984, pp. 758-779.

[2] Alan Huang, 'Architectural Considerations Involved in the Design of An Optical Digital Computer', Proceedings of the IEEE, Vol. 72, No. 7, July 1984, pp. 780-786: Karl-Heinz Brenner, Alan Huang, and Norbert Streibl, "Digital optical computing with symbolic substitution", Applied Optics, Vol. 25, 15 September 1986, pp. 3054-3060.

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

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

[5] 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.

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

[7] C. Warde and J. Kottas, "Hybrid optical inference machines: architectural considerations," Applied Optics, Vol. 25, 15 March 1986, pp. 940-947.

[8] P.W. Smith and W.J. Tomlinson, "Bistable optical devices promise subpicosecond switching," IEEE Spectrum, Vol. 18, June 1981, pp. 26-33.

[9] James Richardson, et al., 'Interim Report on Very Large Parallel Dataflow Program', Honeywell Corporate Systems Development Division, May 1986.

[10] 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.

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

[12] Dror Sarid, Ralph S. Jameson, and Robert K. Hickernell, "Optical bistability on reflection with an InSb etalon controlled by a guided wave," Optics Letters, Vol. 9, May 1984, pp. 159-161.

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

[14] 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.

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

[16] M.W. Derstine, 'Fundamental Geometrical Limitations of Free Space Optical Interconnects,' Unpublished Seminar (1986).

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

[18] 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.

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

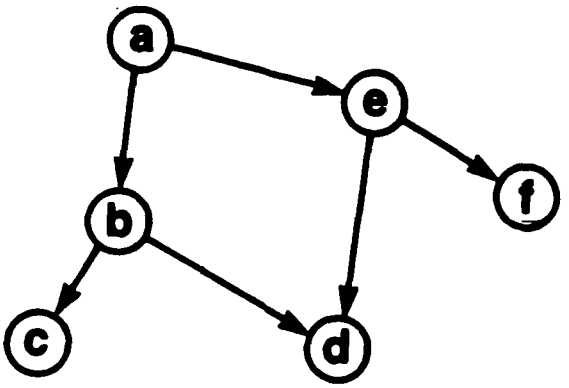
[20] Simon L. Peyton Jones, 'An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions', Proc. of the 1982 ACM Symposium on LISP and Functional Languages, pp. 150-158.

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

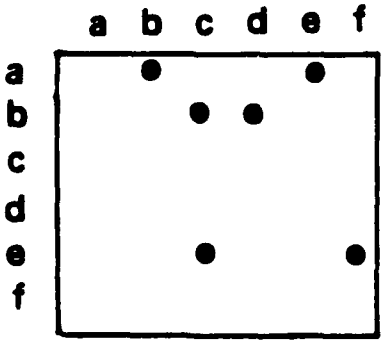
[22] Paul Hudak and Benjamin Goldberg, 'Experiments in Diffused Combinator Reduction', Proc. of the 1984 ACM Symposium on LISP and Functional Languages, pp. 167-176.

[23] H. Richards, Jr., 'An Overview of the Burroughs NORMA', Burroughs Austin Research Center, January 1985.

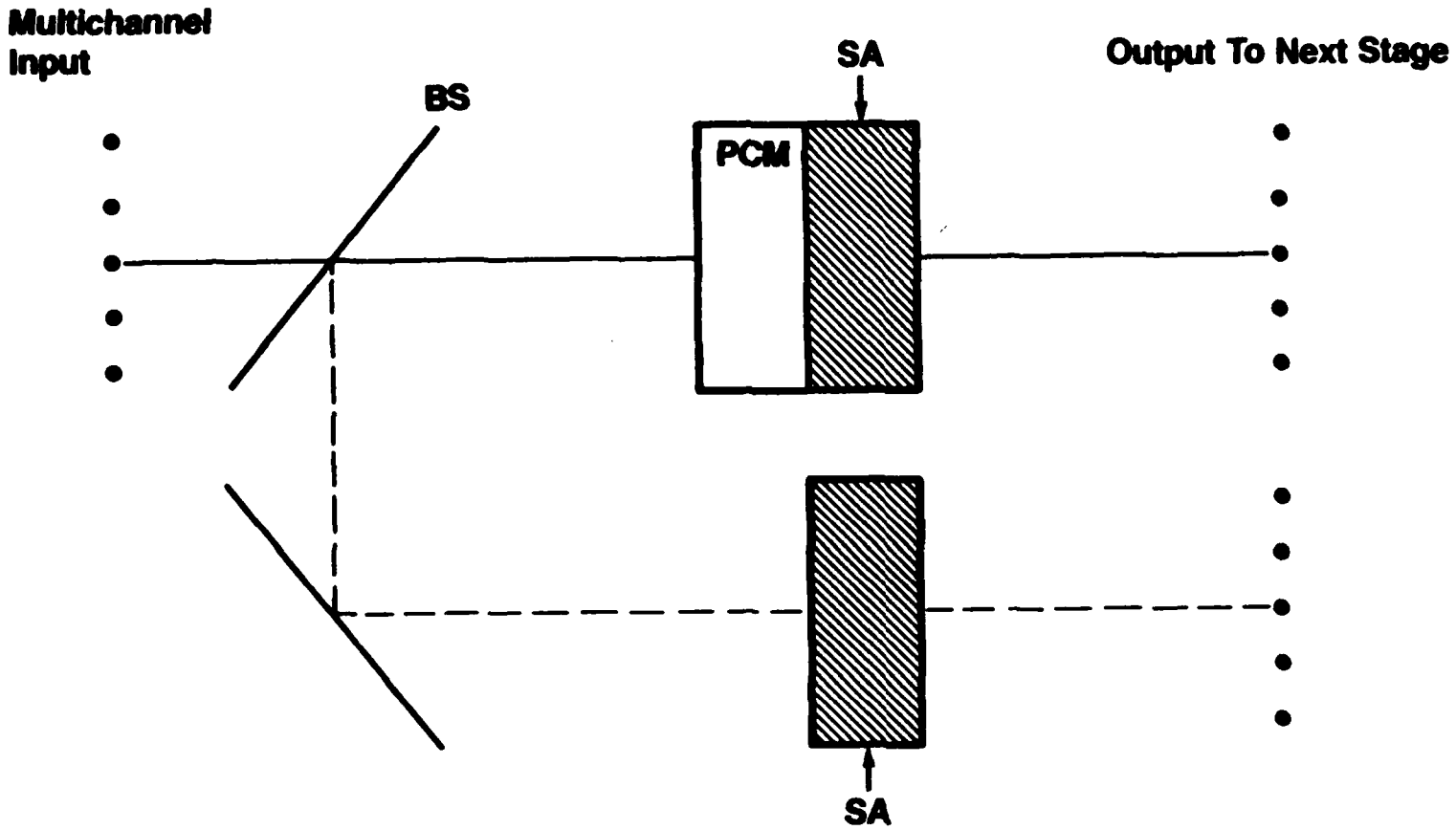
Directed Acyclic Graph



Matrix Representation



Optical Memory Addressing Device



861217