EXPLOITING PARALLELISM IN PRIMITIVE OPERATIONS ON BULK DATA TYPES: SOME RESULTS

S H Lavington, M E Waite, J Robinson and N E J Dewhurst

Internal Report CSM-177, October 1992

This is an invited paper for publication in a forthcoming issue of the journal *Computers and Artificial Intelligence.* It is an extended version of CSM-166 (December 1991).

Department of Computer Science

University of Essex

Colchester

CO4 3SQ

UK

Tel: 0206 872 677

e-mail: lavington@uk.ac.essex

EXPLOITING PARALLELISM IN PRIMITIVE OPERATIONS ON BULK DATA TYPES: SOME RESULTS

S.H. Lavington, M.E. Waite, J. Robinson and N.E.J. Dewhurst

Dept. of Computer Science
University of Essex.

Abstract.

Structures based on tuples may be used to represent the common bulk data types such as sets, relations and graphs, as found in symbolic applications. Few languages support primitive operations that handle such structures explicitly, and fewer support them as persistent objects. Even when relevant data types are accommodated at the language level, implementations are very inefficient because of the lack of underlying systems architecture support. The result is over-complex software and missed opportunities to exploit the inherent data- parallelism present in these structures. In this paper we propose a representational framework, a repertoire of primitive operations, and a systems architecture for handling the bulk data types relevant to (deductive) databases and knowledge-based applications. We indicate the opportunities for exploiting inherent parallelism, and briefly describe a prototype novel parallel hardware unit called the IFS/2 which supports the structure primitives. We present performance figures for the IFS/2 when supporting pattern-directed searching and relational ***join***.

1. Introduction.

The twin problems of slow and complex software occur in most computing applications. It is paradoxical that whilst device technology provides ever faster and cheaper processing elements, computer applications get ever more demanding so that slowness and complexity of software remain issues. To quote C.A.R.Hoare: "The increasing speed of hardware creates the illusion among the users that speed is going to solve the problems of complexity in software, but in fact, it's quite the reverse".

One way to address complexity is to identify generic, frequently-occurring tasks and single these out for special 'built-in' support. Where tasks can be formalised and interfaces agreed upon, this support can even extend down to the hardware. This has been the case with floating-point arithmetic, virtual memory management, graphics primitives, and FFT algorithms. For the non-numeric, i.e. symbolic, world of information systems and AI, it has so far proved difficult to identify generic primitives. Nevertheless, non-numeric applications account for the majority of real- world computing activity so the incentive to identify and formalise primitives is compelling. A start has been made in the field of database languages by describing a theory of bulk types that identifies operations and properties common to structures such as sets, bags, lists, certain graphs, relations, and finite mappings [1]. It is in the spirit of this effort that we use the collective phrase *bulk data types.*

There is another reason to seek generic primitives within symbolic computation. Many non-numeric applications are data-intensive; many bulk data types are observed to have a regular (though complex) representation based on well-understood notions of sets, relations and graphs. There is consequently the possibility of exploiting the high degree of parallelism inherent in many of the operations over these types (e.g. set intersection), when providing support for generic primitives. If this natural parallelism could be successfully exploited, then the provision of high- level generic primitives would not only simplify the user-level software but could also yield dramatic improvements in execution times.

However, a note of caution must be introduced. It is doubtful whether this form of bulk data parallelism can be successfully exploited within the framework of current parallel distributed architecture proposals. Most research in this area has concentrated on models (e.g. graph reduction) in which the *computation* can be dynamically partitioned over several processor-memory pairs. The problem of partitioning large *data* structures in such architectures is difficult, especially when the data is complex, as in knowledge-based systems. In the case of a set intersection, for example, the overhead of copying data between processors rapidly outweighs the advantages of being able to perform many element-to-element comparisons simultaneously.

In the rest of this paper, which is an extended version of a talk given by us at the PARLE-92 Conference in Paris, we take the view that generic primitives at the language level must be accompanied by suitable architectural support. In Section 2 we discuss the functional requirements of symbolic i.e. non-numeric, applications, concentrating on the needs of smart information systems. We then present an underlying formalism for representing bulk data structures, and a formalisation for the various modes of pattern-directed search over these structures. Search is shown to underpin many of the generic primitives. Section 5 presents a suitable architectural framework, based on the notion of *active memory.* In Section 6 we present a repertoire of high-level language primitive operations on bulk data types, together with the corresponding low-level systems programming procedural interface which mechanises them. This low-level interface is, in turn, supported by an add-on active memory unit which both stores and manipulates bulk data. A hardware prototype of this unit, called the IFS/2, is described in Section 7. This uses an array of SIMD search modules under transputer control. Finally, in Section 8 we give some results obtained for the IFS/2 hardware, when supporting operations on bulk data types.

2. Functional requirements.

2.1 Definitions and scope.

The domain of interest may informally be termed knowledge-based, or ‘smart’, information systems. This includes those non-numeric applications which incorporate some non-trivial data-manipulation feature such as the ability to adapt, or to deal with dynamic heterogeneous information, or to carry out inferencing. We use ‘inference’ loosely as a synonym for reasoning, inference, and deductive theorem proving. Practical examples of smart information systems thus include Management Information Systems, deductive databases, Expert Systems, AI planners, Intelligent Information Retrieval, etc.

The necessary software for a smart information system incorporates a knowledge-representation formalism together with knowledge-processing algorithms. There are many representational schemes, not all of which (alas) have a formal basis. Examples include relational, object-oriented, frame-based, production rules, clausal logic, semantic nets, neural nets, non-standard logics (e.g. for belief systems), etc. The algorithms which act upon the knowledge base perform such tasks as pattern- matching/selection/ recognition; making inferences of some sort (including deduction, reasoning, theorem proving, and constraint solving); handling beliefs and uncertainty; learning/adaptation; and data and object management (including sharing, protection, persistence, versioning, integrity, etc.).

2.2 Generic tasks.

For a given knowledge-manipulation task, there is usually a choice of practical techniques and strategies which depend partly on the knowledge representation adopted and partly on the computational platform (hardware and software) available. For example, marker-propagation algorithms can also be expressed as relational operations such as closure and intersection; the condition-elements in production rule systems can either be represented as a tree structure or as an n-tuple relation; logic clauses may be given a linear representation based on the applicative formal system of Schoenfinkel. Each approach may be suitable for a different platform. For example, the architecture of the Connection Machine [2] was inspired by address-induced, xector representations which suggest marker-propagation algorithms; the architectures of the DADO and NON-VON machines were inspired by the tree-structures of the RETE and TREAT matching algorithms for production rule systems [3].

With all the above variety, analysis and quantification of generic tasks is not easy. As a starting point, important activities may be grouped under four (somewhat overlapping) functional headings:

representation and management of knowledge
pattern recognition (including selection)
inference (including reasoning and deduction)
learning.

A little thought will show that pattern-matching may often be an important component of all four activities, not just the second one. The data structures over which pattern-matching could be required include lists, trees, sets, relations, graphs, etc. Lists and other recursive data structures, favoured for good reasons by the constructive set theoretic approach to formal software development, have an inherent sequentiality/ordering. AI programmers often seem to use lists to represent sets - (because of a language-culture?). At some level of detail, sets and lists are of course inter-definable. However, the set is the more fundamental mathematical notion. Since ordering can in any case be modelled in the relational paradigm, we favour a set-based approach to bulk data types. Graphs can also be represented conveniently as relations - (see section 6.1) - as can the data structures of production rule systems - (see section 8).

Taking a set-based approach seems an inherently more appropriate starting point for *parallel* programming paradigms. There is some sympathy for this view with respect to handling data structures in functional languages [4]. Logic languages, however, have usually followed Prolog's sequential resolution strategy. Indeed, most proposals for parallel Prolog architectures are based on replications of the Warren Abstract Machine's sequential, stack-based, control. Nevertheless, we show in Section 4 that pattern-directed search over sets (of clause-heads) can be taken as an effective form of pre-unification filtering.

Programming languages based on higher-level set primitives do exist. Perhaps SETL [5] is the best example. SETL has been used with good effect for software prototyping, but runs inefficiently because its higher-level primitives have to be mechanised by inappropriate (i.e. von Neumann) architectural support. Bearing in mind the importance of pattern-matching activities, which seem to imply *content*-addressability rather than locational (e.g. linear) addressability, a fresh architectural approach is called for. We start with a general look at representation and pattern- matching over set-based objects. We then consider some practical systems architectural requirements in Section 5, in the light of the generic tasks which occur in knowledge-based applications.

3. A representational framework.

We first introduce a general formalism for storing information which is intended to support a wide variety of knowledge-representation schemes. Let the basic elements of our formalism be a universe, D, of atomic objects, Ai. The atomic objects comprise the members of two infinite sets and one singleton set:

C,	the set of constants (i.e. ground atoms);
W,	the set of named wild cards (i.e. an abstraction of the variables of logic programming languages);
∇,	the un-named wild card (i.e. an individual distinct from all the members of the other two sets).

Thus:

$$D = \{C_1, C_2, \ldots\} \cup \{W_1, W_2, \ldots\} \cup \{\nabla\};$$

Within this formalism, the symbol A will be used to denote an atomic object of unspecified kind - (see below).

A word should be said about ground atoms. These may represent actual external entities such as a numerical constant or a lexical token. They may also represent some higher-level <type> information, or an abstract entity including a <label>. The notion of a <label> as a short-hand name for a composed object is mentioned again later.

Having established our domain of atomic objects, let information be represented as sets of tuples composed from this domain. That is, we assume the existence of a constructor *make-tuple*. Tuples may be of any length, and may consist of any choice of component atoms. The ith tuple thus has the general format:

$$Ti = <A_{i_1}, A_{i_2}, A_{i_3}, \ldots, A_{i_m}>,$$

where A_i , A_i , A_i , ..., $A_i \in D$. The m atoms are often referred to as the fields of the tuple. The scope of a wild card atom is the tuple and its extensions. If a tuple is required to be referenced within another tuple (or within itself, in self-referential systems), then a ground atom can be used as a <label>. This gives a straightforward method for representing structured information and complex objects. It is up to the higher-level knowledge modeller to ensure <label> uniqueness, and to enforce a strict (e.g. Gödel-number) or congruence semantics. In other words, we see no theoretical reason for singling out <labels> for special treatment at the lowest level of information representation - (but see Section 5.3 for considerations of efficiency).

Tuples may be grouped into tuple-sets, via a *make-tuple-set* constructor, according to typing and semantic information as described in Section 6. The tuple-set is the basic unit of information from the memory-management viewpoint (i.e. 'paging' and protection). Obviously, logical tuple-sets of varying granularity can be described, down to the single tuple. This suggests a mechanism for memory management, as discussed in Section 6.

Bulk data types such as sets, relations and graphs can clearly be built from tuples and tuple-sets. It is interesting to note that the Conceptual Object Manager used in the ESPRIT STRETCH project has four basic constructors: *tuple, set, dynamic vector,* and *sequence*. The STRETCH project aims to support large knowledge-based systems using both a rule-based language paradigm and an object-oriented language paradigm. It appears that the major justifications for having the extra *dynamic vector* and *sequence* constructors were considerations of efficiency in linearly-addressed memory. This justification could change if the memory were to be associative (i.e. content-addressable) - a topic we return to in Section 5.

Practical examples of data structures built using the *make-tuple* and *make-tuple-set* constructors are given in Section 6. Facilities must naturally exist for creating, deleting, modifying and retrieving the tuples that constitute a knowledge base. Retrieval is achieved as a result of pattern- directed searching. Indeed, many of the useful operations on bulk data structures involve matching of atoms, i.e. essentially searching. Because of its importance, we now consider searching over tuple-sets in some detail.

4. A formalisation of search.

Pattern-directed search is conceptually a single function with three arguments: the interrogand, the matching algorithm to be used, and the tuple-set to be searched. Its result is in general another tuple-set (being a sub-set of its third argument). Operationally, the search proceeds as follows. Each member of its third argument is compared with the interrogand, which is itself a <tuple>; if they match, as determined by the matching algorithm, then that tuple appears in the output set.

The matching algorithm may specify:

(a) search mode - (see below);
(b) a compare- operator, i.e. logical or arithmetic versions of $=, \neq, >, \geq, <$, or $\leq$;
(c) a compare mask, to inhibit a part or the whole of one or more fields from taking part in the comparison;
(d) a means, e.g. Hamming distance, of measuring nearness.

Topic (d) is a current research issue and is not considered further in this paper. For symbolic information, un-masked equality is generally the most relevant type of comparison. Confining ourselves to equality for the present, there are two senses in which a pair of tuples may be said to match: they may be unifiable or identical.

Two tuples are unifiable if they can be instantiated to some common value. We use the word 'instantiated' to mean: 'have constants or wild cards substituted for wild cards, subject to the condition that two wild cards with the same name must be replaced by the same atom.' Since an atom may be used as a <label>, the above definition implies label-dereferencing in the case of structure unification.

Two tuples are identical if they have the same components. Equivalently, a tuple is identical to itself, and to no other tuple. The algorithm for deciding identity is trivial. Unifiability of tuples is also decidable. Identity matching and unifiability matching actually represent extremes of the same process, in the sense that identity matching is in effect unifiability matching in which any wildcards occurring in a tuple are treated as though they were simply constants. This observation gives us a way of describing several useful searching modes.

Let us restrict ourselves to un-labelled tuples. Various modes of search are possible, depending upon whether un-named and named wild cards are given their full interpretation or are treated as if they were constants. We call these two cases 'interpreted' and 'uninterpreted'. Furthermore, either of these two possibilities can be applied to atoms in the interrogand or to atoms in the stored tuple. There are thus 16 possible modes of search (not all of which turn out to be useful). Denoting uninterpreted as 0 and interpreted as 1, we have the following description of five of the more obvious modes:

interpretation of wild cards in interrogand:		interpretation of wild cards in stored tuple:		description of search mode
un-named	named	un-named	named	
0	0	0	0	Identity matching
1	0	0	0	Simple matching
0	0	1	1	One-way matching (F)
1	1	0	0	One-way matching (D)
1	1	1	1	Unifiability matching

Table 1: A sub-set of possible search modes

In Table 1, identity matching is a meta-level mode used, e.g., for systems housekeeping. Simple matching is the basic search available in a conventional CAM. By considering named wild cards as a mechanism for encapsulating constraints at the tuple level, it is seen that two varieties of one-way pattern-matching may be described. Borrowing terminology from declarative languages, Variant (F) in Table 1 has the flavour of the functional paradigm in which constraints are propagated, as it were, from the stored tuple to the interrogand tuple. Variant (D) is the database paradigm in which the constraints, if any, are propagated from the interrogand tuple to the stored tuple. The last entry in Table 1 represents the logic programming paradigm in which the constraints, where these exist, are propagated in both directions.

We now discuss a suitable architectural framework in which to embed the tuple-set objects and pattern-directed search over such objects. The aim is to provide practical support for useful operations over bulk data types, whilst allowing the natural parallelism in these types to be exploited efficiently.

5. The active memory architecture.

5.1 Whole-structure processing

Current parallel architectures differ in the way that processor-store communications are organised. There is a range of possibilities. At one extreme there is the shared memory design, in which several processors share access to, and communicate via, a single memory (e.g. Encore Multimax). At the other extreme there is the fully distributed design, in which each processor only has access to its own local memory, and processors communicate directly with each other [6]. One perceived advantage of distributed memory designs is that the overall store bandwidth is increased linearly as more processors are added to the system. However, when handling large data structures, the overhead of inter-processor communication often outweighs the delays caused by contention in a shared memory design [7]. An alternative approach is to reduce the store bandwidth requirement by making the memory more active. The I-Structure store introduced by Arvind [8] is a step in this direction. Instead of holding an array in the dataflow graph itself, Arvind proposed that it be held in a separate store which is capable of performing array update operations in response to commands which are primitive to the source language. Extending the I-Structure notion somewhat, we might envisage a physically-bounded region of memory which contains all shared data and the means ('methods') for performing operations upon that stored data. If used in a multi-processor environment, the shared memory would accept one command at a time.

Using the regular form of representation for symbolic data described in the preceding section, it is possible to imagine an active form of memory that is capable of performing whole-structure operations such as set intersection in situ. This yields several advantages. The store bandwidth requirement is considerably reduced, since only high level commands and printable results cross the processor-memory interface. Given the regular format, a highly efficient SIMD approach can be taken to exploit the fine grain parallelism available in the majority of required operations. The need for a mapping from backing store formats (e.g. files) to more efficient RAM representations is eliminated. The architecture provides a natural framework for the notion of data persistence.

5.2 An active memory addressed by content.

In the spirit of Arvind's I-Structure store, we propose an add-on *active memory* unit which will both store and manipulate bulk data types. In addition to the general functional requirements of symbolic applications discussed in Section 2, there are some more specific operational features that are desirable. These include persistence, support for garbage collection, concurrent-user access, etc.

Object-based persistent languages, for example PS-alogol [9], allow data structures created in RAM to survive longer than the programs that created them. From an architectural point of view, the most important attribute of a persistent object store would appear to be the ability to access objects of various sizes without requiring a priori knowledge of their physical location or cardinality. In addition, a persistent object store should allow:

maintenance of structural relationships between objects;
protection of (logical sets of) objects;
ability to modify large structures in place.

These requirements imply some form of isolation from physical addressing, so that names used for objects at the applications programming level are carried through to the storage level, regardless of memory technology. This 'universal naming' may be mechanised by some form of one-level associative (i.e. content-addressable) memory. In particular, we might envisage that the tuple-sets of Section 3 are all held in one very large, associatively-accessed, table. When retrieving information from this table, the atoms in an interrogand are the same bit-patterns as the named atoms used by an applications programmer.

The programmer's notion of the type *map* may be implemented via an associatively-accessed table, provided that the semantics of search and equality are suitably defined for the table. The type constructor *map* has been suggested as the basis for representing most bulk types found in database systems or programming languages [18].

In an associative, i.e. content-addressable, memory, some of the problems of garbage-collection are reduced because physical slots which become vacant can be re-used without formality - (physical location is irrelevant to data retrieval). A more intriguing problem is how to manage data movement (i.e. 'paging') within a hierarchy of associative units. This is related to protection (i.e. locking). In [10] we present a scheme for memory management known as semantic caching which uses descriptors similar to the tuple interrogands of Section 3 to identify logical tuple-sets of varying granularity. Relatively straightforward logical tests on descriptors will determine whether a particular tuple-set is wholly, partly, or not at all contained in the fast cache section of an associative memory hierarchy. More details will be found in [10].

There are several examples of the application of CAM techniques to symbolic processing. At the disc level, there are database machines such as Teradata [11]. At the other extreme, there are special-purpose VLSI chips such as PAM [12]. We know of no affordable CAM technology that will offer the flexibility to store large quantities of tuple-sets of a variety of formats, as implied by the general bulk data type representation proposed in Section 3. Relying on disc alone tends to push the *processing* of data structures back into the locus of computational control (i.e. a CPU), which goes against the philosophy of whole-structure processing and the active memory. In Section 7 we describe prototype SIMD hardware that appears to offer direct support for a useful range of primitive operations on bulk data types, in the context of an associatively-accessed active memory unit. Before descending to the hardware level, however, we need to focus more clearly on the nature of the primitive operations and certain implementation decisions that have to be made.

5.3 Implementation considerations.

Physical realisations of conceptual architectures are always a compromise between generality and efficiency, and between cost and performance. In our realisation of the active memory principle (see also Section 7), we have made the following decisions.

(a) Support for label dereference. Using constant atoms for <labels>, as described in Section 3, means that label-dereference must be accomplished via a search command in which the interrogand has all but one field wild. In our SIMD implementation of associative memory, the more wild cards in the interrogand the slower the search. Our hardware thus introduces special <label> atoms which have a fast dereference time equivalent to a 'no wild card' search.

(b) Partitioning the tuple-set space. Each tuple is preceded by a <class- number>, which refers to an entry in a Tuple Descriptor Table (TDT) maintained by firmware. A new TDT entry is created each time a new tuple-set is declared. Each TDT entry gives the format (i.e. number and classification of atoms) for that tuple-set, together with other housekeeping information. The format information is used to control activity during unifiability search, etc. - (see Section 4). Including a <class-number> with each stored tuple ensures that many independent users can store many structures in the active memory without risk of these being confused during search commands, etc.

(c) Size of tuple-sets. In the present implementation, 16 bits are used for the <class-number>; each tuple-set format can specify up to 128 fields. Each field in the present implementation is represented by 32 bits, so that fields longer or shorter than this have to be modelled in terms of fixed-length quantities - (see also (d) below). Remember that a mask can be specified at search time - (Section 3).

(d) Variable-length lexemes. Lexical tokens such as ASCII character strings require to be mapped into fixed-length Internal Identifiers, via an arrangement similar to a compiler's symbol table. We provide hardware support for this, in a sub-unit known as a Lexical Token Converter (LTC). The discrete logic (i.e. non-transputer-controlled) version of the LTC is described in [13]. The LTC uses associative memory to perform the mappings in both directions for strings up to 120 bytes in length, with a fuzzy search facility provided for the translation Lexeme to ID. The use of the LTC to tokenise character strings is optional.

We now return to the topic of the primitive operations themselves, as seen both from the applications programmer's view and the active memory interface view. The intention, of course, is that there should be no mis-match between these views.

6. Procedural interfaces.

6.1 language primitives.

Based on Section 2.2, we take a set-oriented approach to the representation and manipulation of bulk data types. A list of candidate primitive operations may be drawn up as follows. In each case, the underlying representation is in terms of the tuple-set described in Section 3; thus, graphs are stored internally in the active memory unit as relations (see below). The matching algorithms implied by primitives such as *member, select* and *pattern-directed search* are based on appropriate search modes as discussed in Section 4. The relational algebraic primitives are straightforward. They are presented for convenience in three groups, as follows:

1) **Operations on data of type set:**
member, intersect, difference, union, duplicate removal, subset.
2) **Operations on data of type relation (a subtype of set):**
insert, delete, select, project, join, product, division, composition.
3) **Aggregate primitives for sets/relations:**
cardinality, maximum, minimum, average, sum, count, count unique.

The actual format for the corresponding low-level commands and their procedural parameters are discussed in Section 6.2, where sample C program fragments are given.

As far as operations on graphs are concerned, graphs are represented by relations. Nodes (or vertices) could be given names, where a <name> is an atomic object of characterisation *constant, named wildcard*, or *un-named wildcard*. Graph data types are declared as either *directed, undirected*, or *hybrid*. If an additional weight (or label) is to be attached to an arc, then this is represented by an extra field in the tuple. Similarly, if a node is to be given any additional property then this is represented by an explicit field. A graph may also be represented by any pair of columns in an n-ary relation, as in the case of predicate graphs composed in the query graph during deductive database evaluation. Except for the operations in groups 6 and 7 below, the results of graph operations take the form of tuple-sets which are each given a new <class-number> and stored in the active memory ready for further processing. Precise formats are still being considered.

4) **Whole-graph operations producing one or more graphs as result. Each resultant graph is a tuple-set of node-arc pairs.**
a) Transitive Closure (Relation Closure).
b) Find Component Partitions - (a graph component is a set of connected edges).
c) Find all maximum cliques - (i.e., each resultant graph is a maximum sized clique in the original graph).

5) **Graph operations producing either a set of discrete nodes or a set of discrete edge pairs as result.**
a) Find the set of vertices reachable from a set of one or more specified vertices.
b) Find the set of edges reachable from a set of one or more specified vertices (i.e. return a subgraph).
c) Find the set of vertices at distance N from a specified vertex. This is sometimes referred to as the Nth wave.
d) Find the path(s) between two specified vertices.

6) **Graph operations producing a boolean result.**
a) Is the graph connected? b) Is node b accessible from node a?
c) Is node b connected to node a? d) Is a graph cyclic or acyclic?
e) Sub-graph matching: is a specified graph a subgraph of another specified graph?
f) Say whether a specified set, N, of vertices in a specified graph, G, is a clique (i.e. every pair of nodes in N is connected by an edge in G).

7) **Labelled graph aggregate operations, producing an integer result.**
a) Give the shortest/longest path between nodes a and b.
b) For a graph having weighted arcs/nodes, give the arc/node average.

Graph traversal is breadth-first, with a *set* of vertices being produced at each step. These sets of nodes are recycled within the active memory, and they may also be stored - as in the operation to find the set of reachable vertices in a graph. This has applications in deductive databases, for example, where query evaluation can be seen as graph traversal. A chaining rule such as:

p(x,y) : - a(x,v), b(v,w), c(w,y).

defines a graph 'p' as a set of <x,y> instantiations derived from base relations a, b and c. A reachable node set in this *derived graph* is the answer to the query:

?-p(d,y). where ‘d’ is a constant, used as a start node for graph traversal.

Active memory can implement the query evaluation by chaining responder sets through the base relations a, b, c repeatedly. The task delegated to active memory hardware for each iteration of this variable binding set propagation is

$$\Pi_{1,4}((\Pi_{1,4}\ a \underset{2\,=\,1}{\bowtie} b)) \underset{2\,=\,1}{\bowtie} c)$$

For the hardware this simply entails two applications of its primitive *relation composition* (join/project) operation.

Allowing the active memory unit to manage intermediate sets, and remove duplicates, and to test for termination conditions during repeated operations has performance benefits. It also allows hardware to implement useful operations such as Least Fixed Point evaluation:

$$\text{LFP } (z = p \cup (p \circ z))$$

where ‘$\circ$’ denotes relation composition. This can be done even in cases where set p is a derived relation, such as p(x,y) defined in the chaining rule above, so that the LFP is

$$\text{LFP } (z = ((a \circ b) \circ c) \cup (((a \circ b) \circ c) \circ z)).$$

Further information is contained in [14].

There are several other possible primitives, some of which may be suitable for presentation as (high-level) language operations. Examples are *modify* and *sort.* Other primitives are more speculative. Examples are *nearness matching* according to a defined metric (e.g. Euclidean or Hamming distance), and *subgraph isomorphism.* These are the subject of current research and are not pursued further in this paper. Instead, we now turn to the practical issue of appropriate run-time support for the main operations on bulk data types. For this, we describe a particular implementation of the active memory principle.

6.2 Low-level commands.

The higher-level primitives of Section 6.1 are supported by a lower-level procedural interface. Rather than invent yet another systems implementation language, the active memory commands are embedded as library procedures in C. To give the flavour of the active memory interface, we present fragments of C code to illustrate the way a C systems program running on a host computer can manipulate persistent data held in the active memory. As is discussed in Section 7 below, a prototype active memory unit known as the IFS/2 is under development at Essex. The IFS/2 is connected by a transputer link to a host.

IFS/2 C library procedures exist in the host computer which give access to the following groups of hardware/firmware facilities in the IFS/2 add-on unit itself:

(a) housekeeping functions such as opening and closing the active memory unit, managing buffers in the host, etc.;

(b) descriptor management functions concerned with the *make-tuple* and *make-tuple-set* activities. These allow structures to be declared and destroyed; they cause entries to be inserted or deleted in the Tuple Descriptor Table - (see Section 5.3);
(c) Lexical Token Converter functions and label-management functions - (see Section 5.3);
(d) the set and graph structure-manipulation primitives, as listed in Section 6.1.

By way of example, the main pattern-directed search command has the C procedure format:

ifs_search*(matching_algorithm, code, cn, query, &result),*

where:

matching_algorithm specifies one of the search modes of Table 1;

code specifies three parameters for each field in the interrogand, namely a bit-mask, a compare-operator (=, >, etc.), and whether this field is to be returned in the responder-set;

cn is the tuple-set identifier (or 'class-number'), specifying the tuple-set to be searched;

query holds information on the interrogand, in the form of a tuple-descriptor giving the characterisation of atoms (see Section 3) and the actual field values;

&result is a pointer to a buffer in the host which contains information on the result of the search. This buffer contains a header giving: (i) a repeat of the *query* parameter; (ii) the descriptor of the responder tuple-set (including a new class_number allocated to it by the IFS/2); (iii) the cardinality of this responder tuple-set. In the software simulator version of the IFS/2, the header is then followed by a buffer containing the first n fields of the responder- set itself; in the actual IFS/2 hardware, the responder-set is held in the unit's associative memory, according to the active memory's default of treating all information (including derived 'working' sets) as persistent.

When manipulating persistent structures, e.g. via the active memory's relational algebraic operations, the present IFS/2 procedural interface builds on the existing C file-handling syntax. This is illustrated by the following fragment of host C program. We assume that three structures called Alf, Bill and Chris are being manipulated and that each structure is stored as a single base relation. Of these, let us assume that Alf already exists in the IFS/2's persistent associative memory, that Bill is to be created during the execution of the present program, and that Chris is the name we wish to give to the result of *joining* Alf and Bill. In other words:

Chris := *join* (Alf, Bill),

according to specified, compatible, join fields. The following program fragment assumes that the types IFS_ID, IFS_BUFFER, and IFS_TUPLE are defined in the included library file ifs.h; the last two are structures and the first is a 32-bit integer holding a structure's <class-number>. Assume that we already know from a previous program that the <class-number> of Alf = 1. The program loads tuples from a host input device into the new structure Bill, and then performs the required join:

```
#include "ifs.h"
main()
{
    IFS_ID          Bill,
                    Alf = 1,
                    Chris;
    IFS_BUFFER      *Bill_buf;
    TUPLE           t1;

    Bill = ifs_declare(<type>);
    if ((Bill_buf = ifs_open_buf(Bill, "w")) ! = NULL);
    {
        while ( <there are more tuples to be written> )
        {
            <set t1 to next tuple>

            ifs_write (Bill_buf, t1);    /* see note below */
        }
        ifs_close _buf (Bill_buf);

        Chris = ifs_filter_prod (Alf, Bill, <join parameters>);  /* the join command */
    }
}
```

ifs_filter_prod is a generalised relational command which has the following C procedural format:

ifs_filter_prod (cn1, cn2, expr e, expr_list el)

This forms the cartesian product of tuple-sets cn1 and cn2, then filters the result by (e, el) as follows:

expr and *expr_list* are structures defined in the header file "ifs.h". A structure of type *expr* represents an expression constructed from the usual boolean and arithmetic operators. The third argument in *ifs_filter_prod* should represent a boolean expression; the fourth should be a list of integer-valued expressions which define the contents of the output relation. These two arguments together specify a combined *selection* and *projection* operation, which in IFS/2 terminology we call a *filter*. The various relational *join* operations are special kinds of *ifs_filter_prods.*

Note that tuples are written one at a time to the buffer Bill_buf, but sent a block at a time to the active memory unit. Note also that a new class- number is automatically assigned to Bill and Chris by the IFS/2's firmware.

The IFS/2 supports two kinds of class: persistent and transient. Classes formed by relational procedures, such as Chris in the above example, are by default transient. Classes declared explicitly by the user, for example Bill above, are by default persistent. IFS/2 procedures exist for making a transient class persistent, and vice versa. The two kinds of class differ principally in their behaviour when passed as arguments to relational operations. In particular, a transient class is automatically removed from the IFS/2's memory once it has been used in the relational operation for which it was an argument.

It is important to note that all *ifs* commands in the above program fragments are sent down a communications-link to an attached active memory unit, where they cause hardware actions that proceed independently of the host CPU. The result is a reasonably direct route between a high-level primitive and its corresponding hardware support. This hardware can be parallel, as is now described.

7. A prototype parallel hardware mechanisation.

From the top-down requirements presented in Sections 2.3 and 5, it is clear that support for primitive operations on bulk data types depends upon the provision of large volumes of low-cost associative (i.e. content- addressable) memory. Fortunately, a scheme has been developed which uses SIMD techniques, hardware hashing, and conventional components to provide pseudo associative memory at no more than twice the cost per bit of normal RAM. A knowledge-base server known as the Intelligent File Store (IFS/1) using these techniques has been in operation for some years [15]. The techniques have been extended to support relational algebraic operations [14]. Based on this experience, we have built a prototype active memory unit known as the IFS/2 [16], which is now being evaluated.

Briefly, the IFS/2 is an extensible architecture based on nodes. Each node has an array of SIMD search modules under the control of a transputer, and an associatively-accessed SCSI disc. The SIMD search modules each contain 1Mbyte of RAM, a hardware comparator, registers for masking, etc., and some PAL control logic. The present IFS/2 implementation has nine search modules and 700Mbytes of disc per node, and three nodes. The 27Mbytes of semiconductor associative memory are actually used for two purposes: half is used as a cache for the disc (thus implementing a one-level associative memory); the other half acts as a number of dynamically-reconfigurable relational algebraic buffers used in set and graph operations within the active memory unit. The transputer node-controllers are linked to other transputers which look after tuple-descriptor housekeeping, the implementation of complex functions on tuple fields, presentational tasks such as the sorting of responders, and communication with a host computer.Connection to the host (normally a Sun Workstation) is via a standard SCSI channel.

The node controllers are T425 transputers, each with 2Mbytes of local RAM and operating at a clock speed of 25 MHz. The external access time to the group of SIMD search engines, which depends partly upon bus characteristics and search logic, but principally on the 80nsec. DRAM employed, is set at 160 nsec. Apart from the SIMD search modules themselves, the only other section of tailor-made (as opposed to bought-in) logic is a hardware hasher sub-unit attached to each node controller. The byte-manipulation facilities in the T425 transputer's instruction set are somewhat limited, and it was found that a software OCCAM routine for deriving a hash value from a tuple-field was taking about 10 microseconds. Our hardware hasher, consisting of nine GAL chips and two PROM look-up tables, can do the same job in 40 nsec. - ie well within the 160 nsec. SIMD access time. The IFS/2 employs a fixed number of logical hashing bins - (usually 512, but variable for experimental purposes). Each search module contains a 'vertical slice' of all 512 hashing bins. After deriving logical hashing-bin numbers, all data is distributed 'horizontally' across all SIMD nodes in an applications-independent manner.

Choice of the optimum number of SIMD search modules per node is a compromise. With the given T425 node-controlling transputers and 80 nsec. DRAM, it is not cost-effective to have less than two search modules per node. The more SIMD modules per node, the higher the overall search rate. However, each cluster of search modules shares a 32-bit wide local bus which is used for returning responders which successfully match an interrogand. The more responders, the greater the chance of bus contention at a node. Apart from cost considerations, sharing a given number of search modules between more nodes would reduce bus contention - at the risk of increasing inter-node communication overheads during the dyadic reational operations such as ***join***. We are currently investigating all these effects with our three-node, 27 search-module, IFS/2 prototype.

8. IFS/2 performance

The IFS/2 project is still in the development stages, so the results presented below must be regarded as preliminary. Some idea of the basic search capabilities may be obtained from the following figures, which relate to the storage of a single relation (class) consisting of 276,480 3-field tuples. This curious cardinality was arrived at by arranging for the IFS/2's associative cache to be divided into 512 logical hashing bins, and limiting each bin on each of 27 search modules to hold a maximum of 20 tuples. Tuples were generated synthetically so that the values hashed evenly over all bins. Thus, the total number of tuples was: (27 x 512 x 20), = 276,480. Each field for this synthetic data was a 32-bit integer. Each tuple is preceded by a 32-bit system control word giving class number, etc. Thus, the total volume of data was: (276,480 x 4 x 4) = 4.4 Mbytes. This left over 23 Mbytes of search module capacity for the associative relational algebraic buffers.

For this 276,480 tuple relation, the following times were observed:

insert a tuple:	~ 339 microsecs.
member	~ 117 microsecs on average.
delete a tuple:	~ 113 microsecs.
search with one wild card:	~ 688 microsecs. (no responders).

The above times, and indeed all the IFS/2 figures quoted in this Section, were for a version of the prototype which had its 32-bit word SIMD search time set to 200 nanoseconds, rather than the more up-to-date value of 160 nanoseconds which we currently use - (see Section 7).

It is interesting to compare the IFS/2's search rates with another SIMD architecture, namely the Distributed Array Processor (DAP) manufactured by Cambridge Parallel Processing (formerly Active Memory Technology Ltd.). The DAP 510 [19] has a 32 x 32 array of one-bit processing elements, each equipped with a one-bit wide memory of up to 1 Mbit. The DAP would conveniently store a relation as 'layers' of 1024 tuples. Coincidentally, it happens that the time for a DAP to inspect all 1024 3-field tuples is about the same time as it takes the IFS/2 to inspect one tuple - (a little over 10 microseconds). Thus, for small-cardinality relations the DAP and the IFS/2 return similar search times since the IFS/2 is able to narrow its area of search via hardware hashing. This is illustrated in Figure 1, which shows the search times for the *member* operation (i.e. no wild cards in the interrogand), when plotted against relation cardinality. The relation used for this test had three integer fields, generated synthetically. For the test, the tuple being searched for was always placed at the end of the un-sorted relation - (i.e., the 'worst case'). As may be expected, the IFS/2's performance relative to the DAP gets progressively worse as the number of wild cards in an interrogand is increased.

The potential of the IFS/2 hardware is perhaps better illustrated by evaluating the time taken to do a simple relational join, when compared with several commonly-used software systems. All the software systems were normalised to run on a technology roughly comparable with that of the IFS/2's hardware, the nearest easy equivalent being a Sun Sparc Workstation operating at a clock rate of 24 MHz and having 16 Mbytes of RAM.

For the tests, two equally-sized relations of various cardinalities were joined. Each relation's arity was 3, each field being a 32-bit integer. The integer values, which were unique, were chosen so that the output cardinality after the join was about 10 percent of the input relations' cardinality. This simple synthetic benchmark is more fully described in [20]. Basically, two joins are performed, as follows:

Test (a): $R \underset{3\ =\ 1}{\bowtie} S$ Test (b): $R \underset{3\ =\ 2}{\bowtie} S$

The following software systems were evaluated against the IFS/2 hardware:

1. A general-purpose C program.
2. MEGALOG (formerly known as KbProlog)
3. Quintus PROLOG.
4. Kyoto Common LISP.
5. The INGRES relational DBMS.
6. Another well-known relational DBMS, labelled X in Figure 2 for reasons of commercial confidentiality.

The source code for each of these programs is discussed in [20]. The C, MEGALOG, INGRES and X systems each gave approximately equal run-times for join test (a) and (b), indicating that the indexing strategies for these four programs were not sensitive to the position of the join fields. However, tests (a) and (b) gave run-times which differed by three orders of magnitude in the case of LISP and two orders of magnitude in the case of PROLOG. For example, Quintus PROLOG yielded the following run-times, measured in seconds:

Input relation cardinality, n	join test (a)	join test (b)
1,000	0.1 secs	18.867 secs
3,375	0.317 secs	219.284 secs
8,000	0.767 secs	1,231.317 secs
15,625	1,483 secs	too long for comfort
27,000	2.533 secs	too long for comfort
42,875	3.867 secs	too long for comfort
64,000	6.100 secs	too long for comfort

Table 2

Ferranti International has recently announced a memory-resident Prolog database management system written in C, known as the Ferranti Prolog Database Engine [21]. This greatly enhances the database performance of Quintus Prolog, producing *join* results similar to those of the IFS/2 in Figure 2.

In Figure 2 we have plotted the average of the times (a) and (b), and in the case of Quintus PROLOG and Kyoto Common LISP.

Figure 2 shows elapsed times in seconds versus relation cardinality (i.e. number of tuples in each input relation), plotted on a log/log scale. Besides the six software systems, we plot two sets of results for the IFS/2 hardware: the 27 search-module prototype machine and an IFS/2 containing an infinite number of search modules.

As far as conventional software is concerned. Figure 2 supports the general view that the common AI implementation languages are unlikely to be efficient at manipulating structures containing realistic volumes of data. In contrast, the MEGALOG system appears to perform very well.

It is seen from Figure 2 that the IFS/2 may be expected to perform *join* operations between 2 and 5,000 times faster than conventional software. Note that the times given in Figure 2 for the IFS/2 hardware are known to be capable of improvement; the prototype design is currently being refined.

The following general observations may also be made about the IFS/2:

(a) For small relations, the IFS/2 hardware may not be efficient. The break-even point between IFS/2 hardware and conventional software depends on the nature of the whole-structure operation and the particular software system being evaluated. For example, Figure 2 shows that the hand-coded C program performs joins faster than the IFS/2 hardware for relations smaller than about 1000 tuples. The cross-over points for the other