# The IFS/2: Add-on Support for Knowledge-Based Systems

**S H Lavington**

**Internal Report CSM-175, August 1992**

This is a draft Chapter for a book to be published by the IEEE Press in 1993, with the title:

> **Emerging trends in database and knowledge-base machines:** the application of parallel architectures to smart information systems.
>
> edited by M Abdelguerfi (University of New Orleans)
>
> and S H Lavington (University of Essex)

**Department of Computer Science**

**University of Essex**

**Colchester**

**CO4 3SQ**

**UK**

**Tel: 0206 872 677**

**e-mail: lavington@uk.ac.essex**

# THE IFS/2: ADD-ON SUPPORT FOR KNOWLEDGE-BASED SYSTEMS.

**Simon Lavington,**

**Department of Computer Science, University of Essex.**

## Abstract

Knowledge-based systems use a spectrum of programming paradigms and information models, drawn from database and AI technology. In this paper we identify underlying generic tasks, in terms of operations on data-structures such as sets, relations and graphs. We introduce an applications-independent formalism for the representation of knowledge-base data and a repertoire of whole-structure operations on this data. An add-on active memory unit, known as the IFS/2, is described. This uses SIMD-parallel hardware to achieve associative (i.e. content-addressable) storage and processing of sets, relations and graphs. The IFS/2 allows natural data parallelism to be exploited. Performance figures are given for a prototype IFS/2 hardware add-on unit.

## 1. Motivation.

Throughout the last 40 years of computer design, there has been a history of successful innovations which take the form of add-on performance accelerators. For example, hardware floating-point units, address- translation units (for virtual memory management), graphics units, and smart disc controllers have all in due course proved cost-effective for their target applications-domains. Eventually, many such units have become integrated into the main computational platform rather than remaining optional extras. In each case, success has come only after general agreement about standard interfaces and functionality.

In the field of database support, the ICL Content-addressable file store (CAFS, ref. 1) is an early example of an add-on performance accelerator which was then re-engineered and (since the early 1980s) integrated into every Series 3 ICL mainframe. Where SQL has been accepted as a standard, relational database machines such as the NCR/Teradata DBC 1012 (ref. 2, and described in this book) are now regarded as add-on performance accelerators. It is now time for manufacturers to be thinking ahead to the possibility of providing add-on hardware support for information models richer than the relational.

For knowledge-based systems and Artificial Intelligence applications - perhaps best summarised collectively as *smart information systems* - the approach to performance accelerators is subject to some debate. Research trends have been towards special-purpose computers, rather than add-on units to conventional computers. Developments have tended to focus either on particular language paradigms or on particular knowledge- representation formalisms. Examples of the former are the

PROLOG and LISP machines described in ref. 3; examples of the latter include Production Rule machines such as DADO (ref. 4) and semantic network machines such as SNAP (ref. 5).

Designers of (declarative) language machines have generally gone for computational speed, eg as measured in logical inferences per second via an *append* benchmark, and have given lower priority to any ability to handle the large amounts of data which occur in practical knowledge-based systems. The knowledge representation family of machines has produced some interesting technical innovations but regrettably little market take-up. This is no doubt partly because smart information systems themselves are not yet in widespread use. Thus, the Connection Machine, originally intended to support semantic networks (ref. 6), has since been equipped with floating-point capability and sold to the scientific and engineering community. Furthermore, the AI community employs a variety of knowledge- representation formalisms so that machines which only perform cost- effectively on a single formalism or a single declarative language may find it hard to gain acceptance. Finally, very few AI-oriented machines provide easy migration paths from conventional data processing platforms. In short, the pre-requisites mentioned in the first paragraph (ie agreement on standard interfaces and functionality) do not yet exist.

However, the challenge certainly does exist: smart information systems are strategically important; smart information systems run slowly on conventional computers, and their software is too complex. Can we design hardware support, preferably in the form of an add-on unit, which will solve the twin problems of slow and complex software?

In this paper we outline the top-down functional requirements of smart information systems in terms of generic activities, and propose an appropriate novel architecture which supports these activities. The philosophy behind this architecture is to exploit the natural parallelism in whole-structure operations, via an 'active memory' add-on unit which both stores and processes structures such as sets, relations and graphs. The whole-structure operations are then made available to high-level programmers as convenient data-processing primitives. We describe a prototype hardware add-on unit known as the IFS/2. We present the IFS/2's low-level command interface, higher-level programming environments, and the performance of the prototype IFS/2 when compared with conventional software.

## 2. The nature of smart information systems.

We define smart information systems, or knowledge-based systems, as computer applications which incorporate non-trivial data-manipulation features such as the ability to adapt, or to deal with dynamic heterogeneous information, or to carry out inferencing. We use the term 'inferencing' to include related techniques of reasoning and deductive theorem proving. Practical examples of smart information systems include deductive databases, Management Information Systems, Expert Systems, AI planners, Intelligent Information Retrieval, etc. More generally, such applications deal primarily with (large amounts of) symbolic, ie non-numeric, data which is usually complex and usually describable in terms of sets, relations, or graphs.

The phrase 'usually describable' begs comment. For a given knowledge- manipulation task, the

programmer makes a choice of practical data structures and processing strategies; this choice depends partly on the knowledge representation adopted and partly on the computational platform (hardware and software) available. 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 (eg for belief systems), etc. With the exception of neural nets and some higher-order logics, the commonly-used schemes can easily be described in terms of the super-type *relation* (which is taken to include operations on sets, relations and graphs).

Notions of knowledge representation often go hand in hand with processing strategies but, here again, it is believed that relational processing is generally of relevance. For example, marker-propagation algorithms can also be expressed as relational operations such as closure and intersection; also, some well-known matching algorithms in production rule systems can either be expressed in terms of tree traversals or in terms of flatter relational operations (ref. 7).

In spite of the above arguments in favour of a relational approach, relational processing is not a paradigm which some AI programmers find convenient. Particularly, AI programmers often 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, and indeed graphs and trees, can be represented in the relational paradigm, we promote a relational or set-based approach to the generic data structures encountered in *all* information systems. The strong pragmatic argument in favour of this common approach is that it promotes practical formalisms and architectures which may serve a very wide spectrum of applications, extending from conventional databases to the more advanced cognitive systems. Standard interfaces and convenient software migration paths are thus facilitated.

## 3. Functional requirements

Abstracting away from choice of programming language, etc., the important generic activities in knowledge-based systems may be grouped under four (somewhat overlapping) functional headings:

representation and management of knowledge
pattern recognition
inference
learning.

Actually, pattern-matching is often an important component of all four activities, not just the second one - an observation to which we return later. The data structures over which pattern-matching could be required are, as argued previously, relational in flavour. This allows us to call upon the accepted notions of **tuple** and **set**, as the common building blocks from which all relevant data types may be constructed. It also promotes a focussed debate on the desirable repertoire of data-manipulation primitives for smart information systems.

The functional requirements of smart information systems can be described abstractly in terms of operations on sets. Any novel architecture which aims to support smart information systems has to have three attributes:

(a) a practical memory scheme for the low-level representation, accessing, and management of (large amounts of) set-based information;
(b) a strategy for carrying out set manipulations rapidly, eg by exploiting inherent parallelism;
(c) an agreed procedural interface whereby the set operations may be presented to the high-level programmer as convenient data-processing primitives.

For (a), the memory system should ideally organise its accessing and protection mechanisms around the concept of *logical* (rather than physical) sets or objects of varying granularity. That is to say, technology-dependent physical storage boundaries and mappings should be decoupled from the shape and size of the logical objects as seen by an information system's software. Similarly, the maintenance of structural relationships between objects should be decoupled from physical addressing details. These requirements may be seen to be consistent with the properties required of a persistent object manager.

For (b), it would be advantageous if the set manipulations were to be carried out as whole-structure operations on sets referred to by system-wide logical names; this would once again help to decouple software from details of physical location or physical size. It could also remove from programmers any obligation to think about parallelisation strategies.

For (c), there exist a few high-level languages such as SETL (ref. 8) and SQL which already offer relational processing primitives. These languages should enjoy 'direct' hardware support from the candidate novel architecture. Users whose applications are better served by other languages, for example PROLOG or LISP, should be able to call upon a readily-understandable, general-purpose, set-based procedural interface to the novel architecture. This interface might conveniently take the form of a collection of C library procedures.

Returning to general functionality, the nature of both pattern-matching and set-operations implies the comparison (often in 'any' order) of a collection of data elements against one or more interrogands. These comparisons also constitute the basic action of an associative (ie content-addressable) memory. An associative memory has exactly the required properties of decoupling logical access from physical addressing, as required in attribute (a) above. Associative access is therefore seen as a desirable property in an architecture which supports smart information systems. It is shown in Section 7 that the IFS/2 provides a cost-effective mechanisation of large volumes of pseudo-associative memory. However, before going into details of the IFS/2 hardware, it is necessary to consider the desirable low-level representation of tuples and tuple-sets in an associative memory, and to be more precise about the pattern-matching capabilities of such a storage system - especially as it will be required to handle variables as well as ground data.

## 4. The IFS/2 representation of tuples and sets

We require a semantics-free, set-based formalism for low-level storage, suitable for use by a wide variety of knowledge-representation schemes. The scheme used by the IFS/2 is now introduced. The basic elements of the IFS/2 formalism is 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 (ie ground atoms);
W, the set of named wild cards (ie an abstraction of the variables of logic programming languages);
$\nabla$, the un-named wild card (ie 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 the IFS/2's domain of atomic objects, information is represented as sets of tuples composed from this domain. For this, the IFS/2 supports a *make-tuple* constructor. In general, tuples may be of any length - though there are some IFS/2 practical limitations mentioned later. Tuples may consist of any choice of component atoms. The ith tuple thus has the general format:

$$Ti = \langle Ai_1, Ai_2, Ai_3, \ldots, Ai_m \rangle,$$

where $Ai_1, Ai_2, \ldots, Ai_m \in D$. 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 (eg 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 ref. 16 for comments on efficiency).

Tuples may be grouped into tuple-sets, via an IFS/2 *make-tuple-set* constructor, according to typing and semantic information. The tuple-set is the basic unit of IFS/2 information from the memory-management viewpoint (ie '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 briefly in Section 6.

Practical use of the IFS/2 *make-tuple* and *make-tuple-set* constructors, and other data-manipulation facilities, is discussed in Sections 8 and 9. Facilities 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. As mentioned in Section 3, searching or matching activity lies at the heart of many operations on relational structures. Because of its importance, we now consider searching over tuple-sets in some detail. Of particular interest is the handling of variables (i.e. wild card atoms).

## 5. 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. There are many possible varieties of matching algorithm. These might include nearness-matching as implied by neural network paradigms, using a metric such as Euclidean distance or Hamming distance. For the purposes of this paper, we confine ourselves to matching in the context of the super-type *relation*.

In the IFS/2, the matching algorithm may specify:

(a) search mode - (see below);
(b) a compare- operator, ie logical or arithmetic versions of =, ≠, >, ≥, <, or ≤ ;
(c) a compare mask, to inhibit a part or the whole of one or more fields from taking part in the comparison;

For information systems containing variables, un-masked equality is generally the most relevant type of compare operator and mask. 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). Five of the more obvious modes are:

(i) identity matching: bit-patterns are compared, regardless of the kind of atom - (ie, wild cards in both interrogand and stored tuple are 'uninterpreted').
(ii) simple matching: un-named wild cards in the interrogand are interpreted, as in conventional content-addressable memory (CAM).
(iii) one-way matching (F): both kinds of wild card in the stored tuple are interpreted; this search mode is similar to the functional programming paradigm.
(iv) one-way matching (D): both kinds of wild card in the interrogand are interpreted - (the database paradigm).
(v) two-way, or unifiability, matching: all wild cards are interpreted.

The IFS/2 supports all the above modes. In Section 9 we give an example of a C procedure, which allows a programmer to specify a particular search option.

## 6. An architectural framework for active memory

Having presented an abstract view of the IFS/2's internal storage conventions, we now discuss the development of a practical architectural framework. The previous two Sections suggest that an appropriate memory scheme for smart information systems consists conceptually of a very large table for holding the variable-length tuples of Section 4. Furthermore, this table requires to be accessed associatively (ie by content), according to the pattern-directed search modes described in Section 5, to yield resultant tuple-sets. These tuple-sets may then be subject to further processing, for example via a relational *join* operation, during which there is much inherent parallelism. How can we exploit this by performing parallel 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 (eg the European Declarative System, EDS), ref. 9. The big issue for each of these machines is likely to be the strategy for distributing both data and work amongst nodes, in order to achieve an acceptably scalable performance for non-trivial information systems.

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. An alternative approach is to reduce the store bandwidth requirement and access contention by making a shared memory more active as a unit, instead of being passive. The I-Structure store introduced by Arvind (ref 10) 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 Section 4, 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, an efficient SIMD approach can be taken to exploit the fine grain parallelism available in the majority of required operations - (see later). 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.

In the spirit of Arvind's I-Structure store, the IFS/2 is an add-on **active memory** unit which both stores and manipulates the data structures used by knowledge-based systems. In addition to the general functional requirements of symbolic applications discussed in Section 3, 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 (ref. 11), 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.

As mentioned previously, this implies the isolation of software 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 (ie content-addressable) memory. In particular, the IFS/2 arranges that the tuple-sets of Section 4 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.

In an associative, ie content-addresable, 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 (ie 'paging') within a hierarchy of associative units. This is related to protection (ie locking). In ref. 12 we present a scheme for memory management known as semantic caching which uses descriptors similar to the tuple interrogands of Section 4 to identify logical tuple-sets of varying granularity. The IFS/2 will employ relatively straightforward logical tests on descriptors to 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 ref. 12.

There are several examples of the application of CAM techniques to symbolic processing. At the disc level, there are database machines such as Teradata (see ref. 2). At the other extreme, there are special-purpose VLSI chips such as PAM (ref. 13). 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 data type representation proposed in Section 4. Relying on disc alone tends to push the *processing* of data structures back into the locus of computational control (ie a CPU), which goes against the philosophy of whole-structure processing and the active memory. In the next Section we describe the IFS/2's prototype SIMD hardware that offers direct support for a useful range of primitive operations on data structures in the context of an associatively-accessed, one level, active memory unit.

## 7. IFS/2 hardware design.

The first technical challenge in designing the IFS/2 was to devise a modularly-extensible associative memory cache, which would provide several tens of megabytes of content-addressable storage at semiconductor speeds. An approximation to this goal has been achieved by employing

groups of simple SIMD search engines (essentially comparators), each with its own bank of DRAM. This technology was first used in our Intelligent File Store project, IFS/1 (ref. 14). The first production IFS/1 knowledge- base server went into operation at a customer site in December 1987, with 6 Mbytes of pseudo-associative memory and typical search times as follows:

38 microseconds (no wild cards in interrogand);
244 microseconds (one wild card);
2 milliseconds (two wild cards);
inserting a tuple took between 38 and 76 microseconds;
deletion took between 4 and 38 microseconds.

Although the IFS/1 appeared as a fully-parallel associative table to programmers, we call the technique 'pseudo-associative' because search times increase as the number of wild cards in the interrogand increases. This is because the IFS/1 uses hardware hashing to limit the area of SIMD search; the more unknown fields in the interrogand tuple, the less hashing information is obtainable and so more hashing bins have to be searched. Note, however, that there is no concept of key (or index) fields; associative access is also independent of tuple-ordering.

For the new IFS/2 unit, we have adapted the IFS/1 SIMD technology to suit the *active memory* concept introduced in the previous Section. The SIMD associative memory is now more flexible. It is used both as an associative cache to the IFS/2 associatively-accessed discs and as associative buffers for the whole-structure relational operations on tuple-sets. The basis for this approach to relational processing was investigated via a hardware test-bed described in ref. 23. The number of relational buffers can be arranged dynamically to suit the particular sequence of (relational algebraic) commands being executed by the IFS/2, as explained later. Thus, the IFS/2's SIMD search engines are used both for pattern-directed search and for the element-by-element comparisons inherent in most set and graph primitives. In this way, whole-structure operations are performed 'in situ' and 'in parallel' without intermediate transfer of commands or data between the IFS/2 and its host computer.

Figure 1 gives an overall view of the prototype IFS/2 add-on unit. Connection with the host computer, typically a Sun Workstation, is via a standard SCSI channel. The IFS/2 is based on modularly-extensible nodes, each node containing a transputer-based node controller, a group of SIMD search modules, and a SCSI disc. The prototype shown in Figure 1 has three nodes, each containing 9 Mbytes of associative cache and a 720 Mbyte associatively-accessed disc. Each 9 Mbytes of cache is arranged as three double-Euro sized PCBs, each containing three banks of DRAM and three search engines, connected by a local 32-bit wide bus to its node controller. This 32-bit bus is memory-mapped into the node-controller's address space. Apart from the four SCSI channels, all other interconnections in Figure 1 are transputer links.

The node controllers in Figure 1 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. The blocks labelled TRAM in Figure 1 each consist of an off-the- shelf small daughter-board containing a T801 transputer and 4Mbytes of local RAM. The SCSI TRAM modules are similar off-the-shelf modules containing a transputer-based SCSI controller and local buffering.

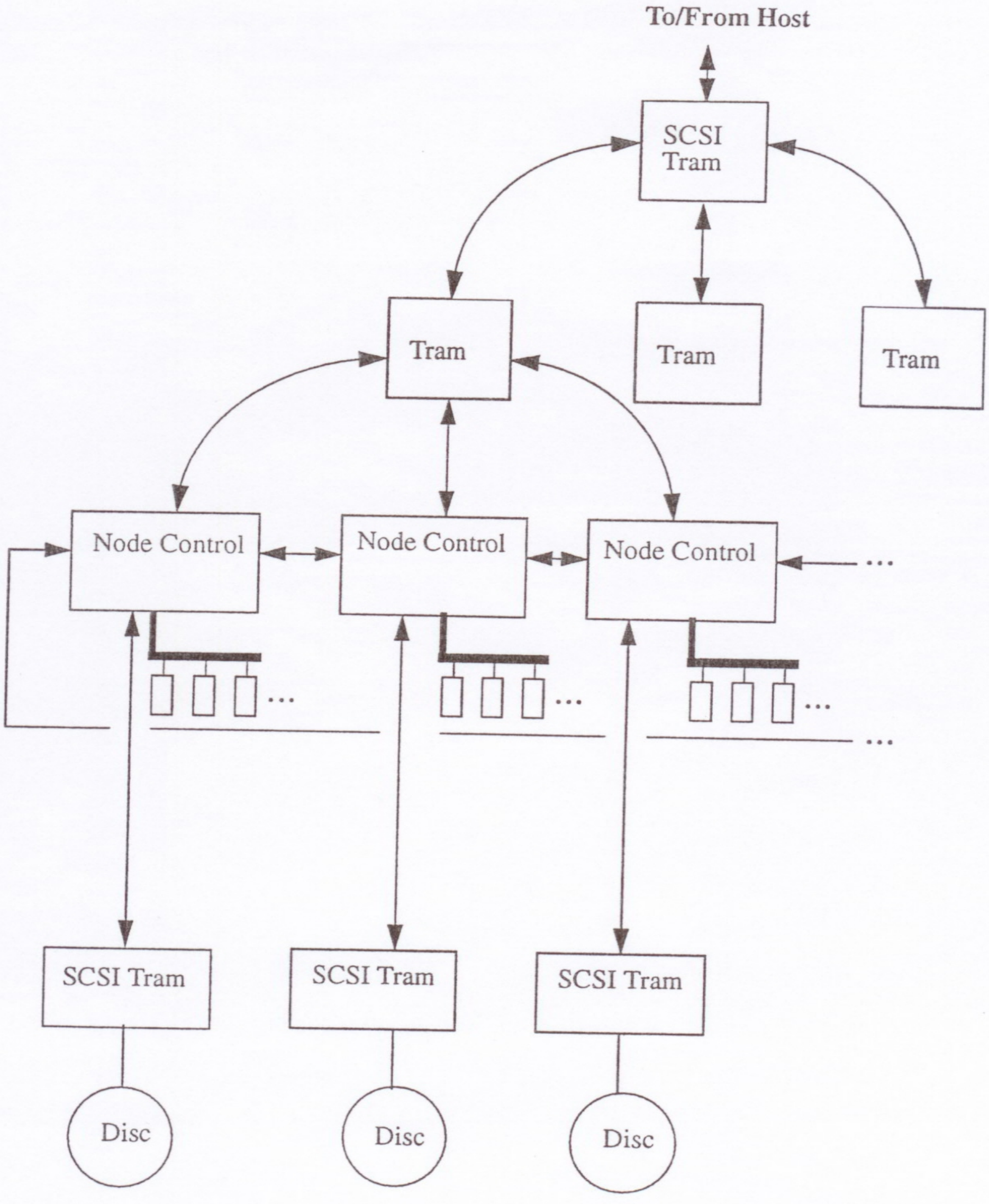


Figure 1 : THE PROTOTYPE IFS/2

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. (Interestingly, an experimental FPGA implementation of this same hasher design is about four times slower and four times as costly).

The IFS/2 employs a fixed number of logical hashing bins - (usually 512, but variable for experimental purposes). Each search module in Figure 1 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 Figure 1. (Graph operations may vary this distribution strategy - see below). Since tuples may be of variable length, this means that corresponding tuple-slots on all search engines have to be of the same length. The actual division of a search module DRAM into 'tuple slots' varies dynamically as data is inserted or marked as deleted.

Choice of the optimum number of search engines per node in Figure 1 is an interesting compromise. As mentioned above, the basic hardware data- comparison time for 32 bits is 160 nsec., plus time taken to read a match-status line, and is independent of the number of search engines (n) at a given node since matching is done in SIMD parallel across all n engines. For reference, the node-controlling transputer could do the same single word comparison in about half the time if it simply had 80nsec. local DRAM and no search engines to control. Therefore, it is not cost-effective to have less than two search engines per node. The overall data-comparison rate naturally increases linearly with n. However, all tuples which match the interrogand will require certain fields to be returned along the common bus; the more successful matches, the more potential bus contention and hence possible degradation in search rate. Apart from cost considerations, sharing a given number of search engines between more nodes would reduce local bus contention and therefore potentially speed up the simple searching operations. However, the diadic relational algebraic operations such as *join* necessitate inter- node communication - an argument for keeping the number of nodes low. Graph operation such as *reachable_node_set* introduce yet another set of considerations. For very bushy graphs, it is advantageous to distribute the graph amongst several nodes. In contrast, the data-comparisons for thin chain-like graphs are best confined to a single node. We are currently investigating all these effects in the three-node, 27 search- engine IFS/2 prototype.

One other point should be mentioned in connection with graph operations which are non-deterministic in nature. The task of detecting the end of computation is difficult, because each node has an unpredictable amount of work to do, involving unpredictable amounts of communication with other nodes. We have devised a simple inter-node hardware 'busy' line which greatly simplifies the controlling OCCAM firmware. This feature will be more fully reported elsewhere.

The IFS/2's total associative memory may store tuples from many tuple-sets, having various formats and belonging to various users. Each declared tuple-set is given a unique *class_number*;

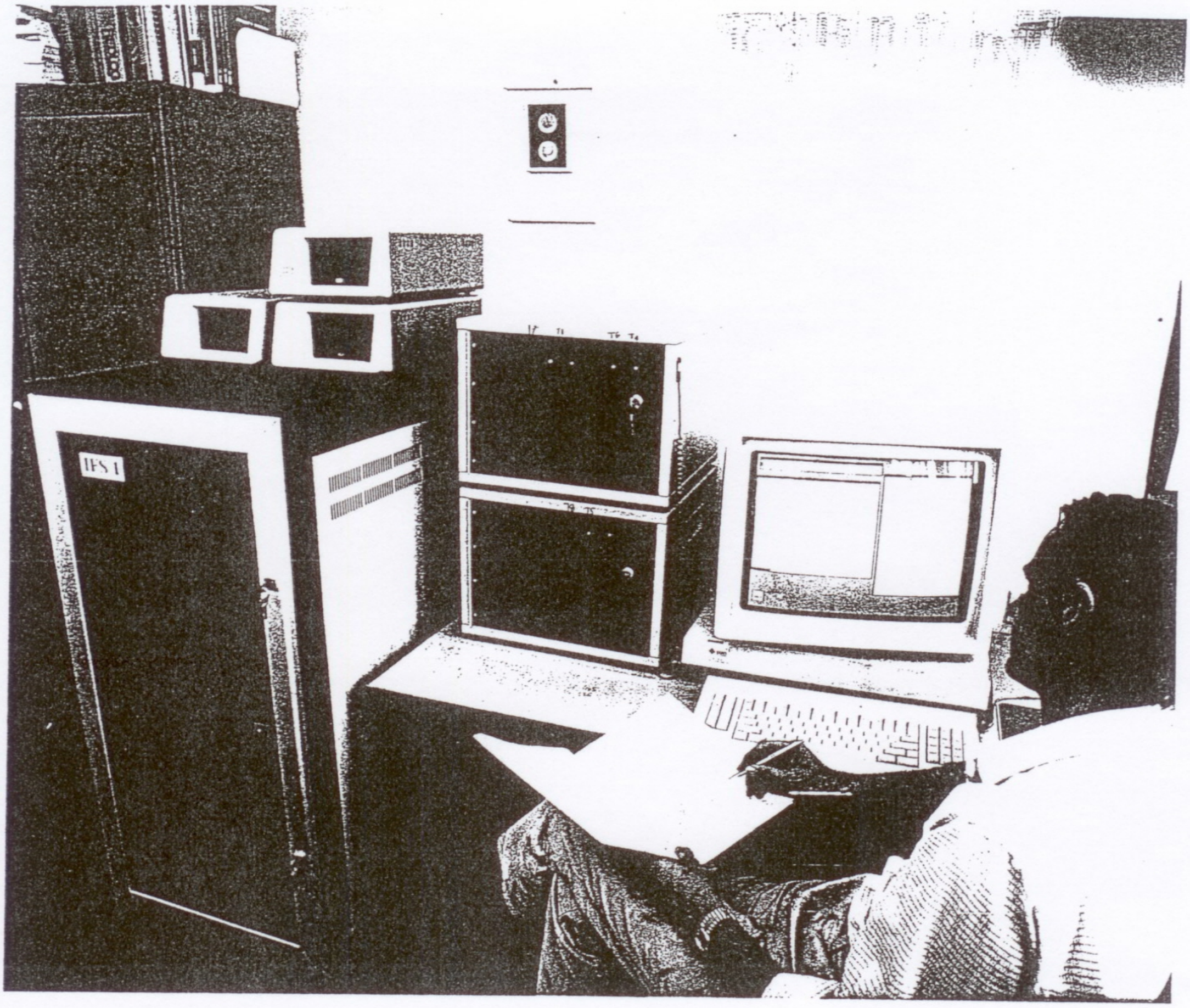

Figure 2: The IFS/1 and prototype IFS/2.

Left: an IFS/1 production unit (completed December 1987).

Centre: the two cabinets of a three-node IFS/2 prototype. The cabinets contain a total of 27 Mbytes of semi-conductor associative (ie content-addressable) memory which acts both as a cache and as associative buffers during the execution of whole-structure operations such as *join* and *transitive closure*. The memory is actually implemented in terms of 27 SIMD-parallel search engines, groups of nine being controlled by transputer-based hardware. Each IFS/2 node includes a 720 Mbyte SCSI disc; (the discs can be seen sitting on top of the old IFS/1 in the photograph). Connection between the IFS/2 and a host (eg Sun Workstation) is via a standard SCSI channel.

all stored tuples in that set are preceeded by a system control word which contains the class-number, a byte-count giving the length of the tuple, and a free/occupied marker. Predictions of search times are complicated, because they are data-dependent. A detailed analysis is given in ref. 15 where it is shown, for example, that the time for a simple match (see Section 5) for class c depends upon:

average number of hashing bins to be searched;
av. no. of tuples in class c per search engine per bin per node;
av. no. of expected matching tuples per search engine per bin per node;
av. no. of tuples *not* in class c per search engine per bin per node;
av. no. of responders (fields) per bin per node;
communication time taken to return each responder.

Actual times for sample data are given in Section 10, together with IFS/2 performance figures for relational and graph operations.

The following points of implementation detail should be mentioned about the prototype IFS/2.

(a) At present, up to $2^{16}$ distinct tuple-sets (or 'classes') can be accommodated. Each tuple-set format can specify up to 128 fields. Each field is represented by 32 bits, so that longer or shorter fields have to be modelled accordingly - (see also (b) below). Remember that a mask can be specified at search time - (See Section 5).

(b) Variable-length lexemes, such as ASCII character strings, are mapped into fixed-length 32-bit Internal Identifiers. In the old IFS/1, associative memory was provided to do this conversion by hardware (ref. 21). At present, we do the conversion in host software. In due course, we will use some of the IFS/2's SIMD search engines to give hardware-assistance to lexical token conversion.

A photograph of the prototype IFS/2 is shown in Figure 2. It is difficult to attach an accurate cost to the IFS/2 as an add-on unit, but it is estimated to be about three times the cost of an average Workstation.

## 8. The high-level language view of the IFS

The IFS/2's tuples and tuple-sets may be manipulated by the standard operations associated with the super-type *relation*.

These would be expected to include:

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.

4 **Graph operations**:
Transitive closure, find component partitions, find reachable nodes/edges, find shortest path, test for sub-graph matching, test for connectedness, test for cycles.

Ideally, all these kinds of primitives would already be embedded in a well-used persistent programming language. This 'ideal' language would efficiently support both database and AI paradigms. Since no such utopian language yet exists, we have to work through the more traditional software conventions. For example, there are at least three products which combine logic programming and relational database technology. Megalog (formerly Kb-Prolog, ref. 17) is a good example.

As mentioned previously, few programming languages based on higher-level set primitives exist. Perhaps SETL (ref. 8) is the best example. SETL has been used with good effect for software prototyping, but runs inefficiently because its higher-level primitives normally have to be mechanised by inappropriate (ie von Neumann) architectural support. Nevertheless, SETL demonstrates that set and relational primitives find acceptance amongst software implementers.

Of course, SQL may be regarded as the best way of presenting relational operations as far as databases are concerned. We are currently implementing an SQL front-end to the IFS/2. This will have a tailor-made interface to the IFS/2, which is based on a relational algebraic tree representation of each SQL statement. The package of database actions (selection, join, etc.) implied by each SQL statement is transmitted as a tree structure across the SCSI interface from the host - (see Figure 1). As this stage, only algebraic query-optimisation has been performed. Once the relational algebraic tree has entered the IFS/2, OCCAM firmware carries out IFS-specific optimisations and the SIMD search engines are employed to perform the set of low-level actions implied by the original SQL command. Note that intermediate (i.e. temporary) results are held in associative buffers within the IFS/2. Note also that the standard SQL commands *create_index* and *drop_index* are irrelevant to the IFS/2, and are not supported.

IFS/2 front-ends to non-relational languages will use a general-purpose C procedural interface. For example, we are developing an IFS version of an OPS5-like production-rule system called CLIPS (ref. 7). We also plan a Prolog front-end. The general-purpose C procedural interface upon which all such front-ends are built is now described.

## 9. IFS/2 low-level procedural interface

The IFS/2 procedural interface consists (at present) of 42 C library procedure calls. These procedure calls, available on a host computer, communicate with the actual IFS/2 hardware via the SCSI channel as shown in Figure 1. The 42 commands are grouped as follows:

a) 5 for housekeeping (e.g. opening and closing the IFS/2);

b) 3 for tuple descriptor management, allowing structures to be declared and destroyed; these cause entries to be inserted in or deleted from a Tuple Descriptor Table administered by IFS firmware;
c) 3 for inserting and deleting tuples, and pattern-directed search over sets of tuples;
d) 5 for handling character strings or similar lexical tokens; these are converted into fixed-length internal identifiers;
e) 5 for label management; these commands help in particular with fast label dereferencing - (see ref. 16);
f) 7 for relational algebraic operations;
g) 14 for graph operations which support deductive databases.

By way of example, the following is the procedure call for the IFS/2's main pattern directed search command - (see group (c) above):

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

where:

*matching_algorithm* specifies one of the search modes of Section 5;

*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 4) 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 as persistent.

When manipulating persistent structures, eg via the active memory's relational algebraic operations of group (f) above, the 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);

        }
        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**.

It is important to note that all **ifs** commands in the above program fragments are sent down a communications-link to the IFS/2 active memory unit, where they cause hardware actions that procede independently of the host CPU. The result is a reasonably direct route between a

relational primitive and its corresponding hardware support.

The full repertoire of relational algebraic commands in group (f) are:

**ifs_filter**
**ifs_filter_union**
**ifs_filter_diff**
**ifs_filter_prod**
**ifs_filter_div**
**ifs_calc**

The last command implements the usual aggregate operations (summing, averaging, and calculating maximum and minimum values) over a given column of a given relation.

As far as graph operations are concerned, the repertoire of possible commands is quite extensive - (see ref. 16). For the present, we have concentrated on those primitives which are useful for supporting deductive databases (including certain recursive queries). In addition to three procedures concerned with declaring graphs, the following commands are supported:

**transitive closure**
**reachable node set**
**reachable edge set**
**nth wave nodes**
**nth wave edges**
**check path**
**relation composition**
**check cycles**
**check cycles in a subgraph**
**waveset edges 1**
**waveset edges 2**

More details will be found in ref. 18. In general, graphs are materialised from base relations, an entry in a Graph Descriptor Table (GDT) giving the recipe for constructing the graph from fields selected from one or more base relations. For example, the set of nodes reachable from a given node may be obtained via the procedure call:

**ifs_reachable_nodes** *(gn, start_node, & result)*

where

*gn* is a graph number identifying an entry in the GDT which gives the recipe for constructing the graph from base relations(s);

*start_node* is the value representing the start node;

*& result* gives the class-number of the resultant set (held within the IFS/2).

## 10. 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 AMT DAP510 (ref. 19). This Distributed Array Processor 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. 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 relational join, when compared with several commonly-used software systems. All the software systems were normalised to a technology roughly comparable with that of the IFS/2's hardware, the nearest easy equivalent being a Sun Sparc Workstation running 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, which is being used as part of an analysis of several novel architectures, is more fully described in ref. 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 - see ref 17).
3. Quintus PROLOG.
4. Kyoto Common LISP.
5. The INGRES relational DBMS.
6. Another well-known relational DBMS, labelled X in Figure 3 for reasons of commercial confidentiality.

The source code for each of these programs is discussed in ref. 20. The C, MEGALOG, INGRES and X systems each gave approximately equal run-times for join tests (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 | 219.284 |
| 8,000 | 0.767 | 1,231.317 |
| 15,625 | 1,483 | too long for comfort |
| 27,000 | 2.533 | too long for comfort |
| 42,875 | 3.867 | too long for comfort |
| 64,000 | 6.100 | too long for comfort |

Ferranti International has recently announced a memory-resident Prolog database management system written in C, known as the Ferranti Prolog Database Engine (ref. 22). This greatly enhances the database performance of Quintus Prolog, producing *join* results similar to those of the IFS/2 in Figure 3.

In Figure 3 we have plotted the average of the times (a) and (b), in the case of Quintus PROLOG and Kyoto Common LISP.

Figure 3 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 machine of Figure 1 and an IFS/2 containing an infinite number of search modules.

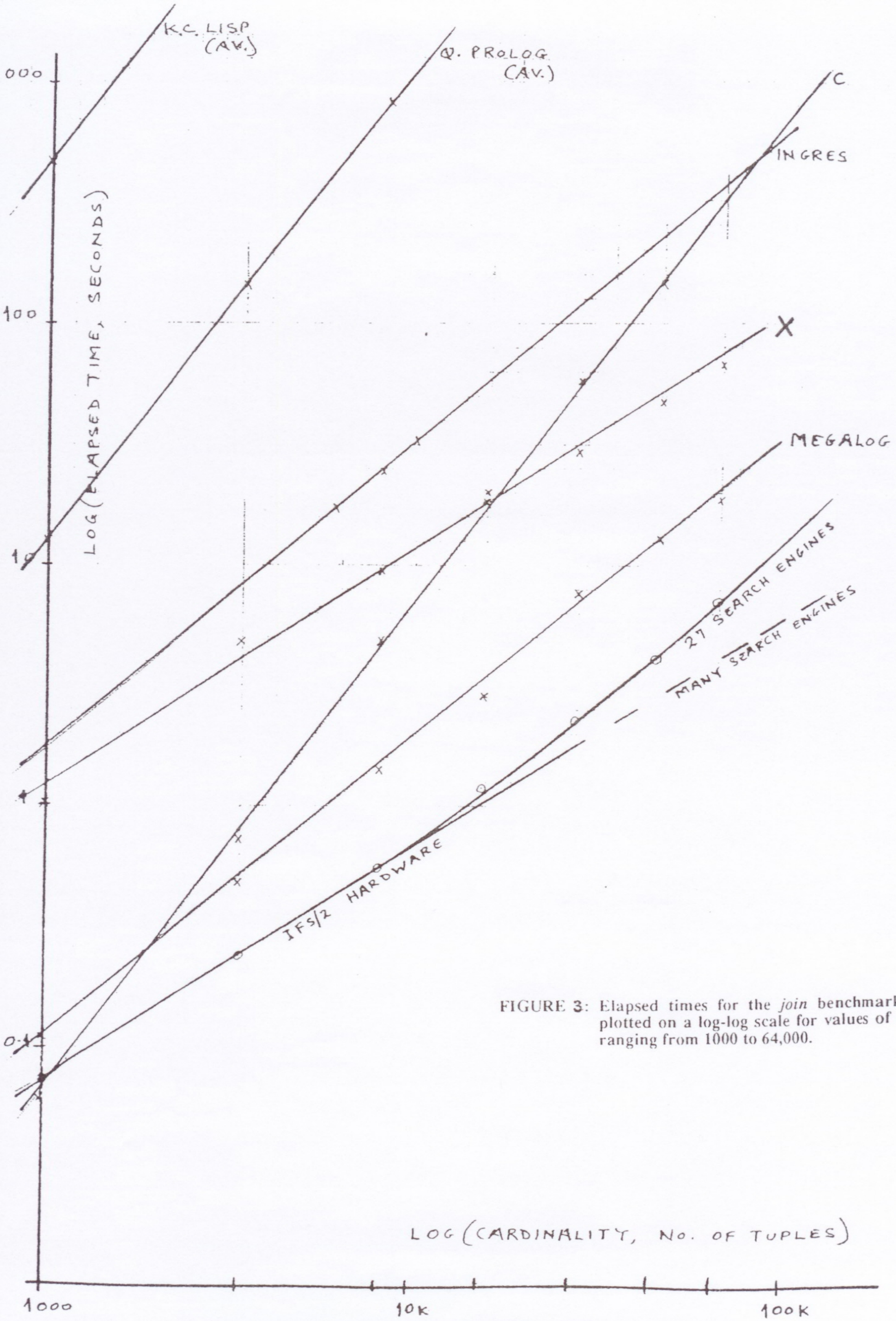


FIGURE 3: Elapsed times for the *join* benchmark, plotted on a log-log scale for values of n ranging from 1000 to 64,000.

As far as conventional software is concerned, Figure 3 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 3 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 3 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 software systems of Figure 3 lie below 100 tuples. These conclusions are in line with our measurements of the IFS/2 when supporting the CLIPS Production Rule System (Ref. 7). In Ref. 7 it is shown that the IFS/2 may speed up production systems by three orders of magnitude if the number of initial facts is greater than about 10,000. Conversely, use of the IFS/2 actually slows down performance for production systems having fewer than about 100 initial facts.

(b) For the particular *join* tests of Figure 3, there is no advantage in having more than 27 search modules in the IFS/2 hardware for cardinalities under about 3000 tuples. Above that size, the IFS/2 performance curve can be kept linear by adding more search modules.

## 10. Conclusions

To be cost-effective, hardware support for knowledge-based systems must not only offer raw performance improvements. It must also reduce overall software complexity and be capable of easy integration with conventional hardware and software standards. In this paper, we have identified frequently-used generic tasks within knowledge-based systems, and then moved responsibility for these tasks to add-on hardware. The super-type *relation* is shown to underpin the important generic tasks. The IFS/2 is a performance accelerator for the super-type *relation*.

In order to preserve applications- and language- independence, we have developed a scheme for the low-level representation of information based on the well-understood notions of *tuple* and *set*. These, together with a formalisation of pattern-directed search, form the basis for the IFS/2's repertoire of whole-structure operations on sets, relations and graphs. The operations are presented to applications software through C library procedures and a standard SCSI interface.

The IFS/2 hardware uses SIMD-parallel techniques to achieve associative (ie content-addressable) storage and processing of structured data. This hardware strategy enables the parallelism inherent in operations such as pattern-directed search to be exploited automatically. The design is modularly-extensible, and makes much use of off-the-shelf components such as transputers. The cost-per-bit of the IFS/2's associative storage is not much more - (perhaps five

times more) - than conventionally-addressed linear memory. The IFS/2's architecture conforms to the principle of *active memory*, by means of which logical objects of varying granularity are managed and manipulated in a manner that requires no knowledge of physical addressing on the part of host software. Performance figures for the IFS/2 prototype indicate that speed-ups in the range 10 to 100 times over conventional software are readily achievable. Work is in hand to develop the prototype in a number of directions, particularly: to produce useable higher-level language interfaces, extend the repertoire of graph operations, allow more flexibility in choice of field-formats, implement hardware support for lexical token conversion, understand better the optimal distribution of SIMD search engines amongst nodes, refine the low-level OCCAM firmware, and conduct realistic performance analysis and benchmarking against commercially-acceptable workloads.

## 11. Acknowledgements.

It is a pleasure to acknowledge the contribution of all other members of the IFS team at Essex. Particular mention should be made of Neil Dewhurst, Jenny Emby, Andy Marsh, Jerome Robinson, Martin Waite and Chang Wang. The work described in this paper has been supported by the UK Science & Engineering Research Council, under grants GR/F/06319, GR/F/61028 and GR/G/30867.

## 12. References.

1. V A J Maller, *Information retrieval using the Content Addressable File Store*. Proc. IFIP-80 Congress, pub. by North-Holland, 1980, pages 187 - 192.

2. J Page, *High performance database for client/server systems.* Parallel Processing and Data Management, ed. Valduriez, Chapman & Hall, 1992, pages 33 - 51.

3. P M Kogge, *The architecture of symbolic computers.* McGraw-Hill, 1991.

4. S J Stalfo and D P Miranker, *DADO: A Parallel Processor for Expert Systems,* Proc. IEEE Conf. on Parallel Processing, 1984, pages 92-100.

5. D Moldovan, W Lee and C Lin, *A marker passing parallel processor* for AI. Proc. Workshop on Parallel Processing for AI, held in conjunction with IJCAI-91, Australia, page 128.

6. W D Hillis, *The Connection Machine.* The MIT Press, 1987..

7. S H Lavington, C J Wang, N Kasabov and S Lin, *Hardware support for data parallelism in production systems*. To be presented at the Third International Workshop on VLSI for Neural Networks and Artificial Intelligence, Oxford, Sept. 1992.

8. R B K Dewar, E Schonberg, J T Schwartz, *High-level programming - an introduction to the programming language SETL.* Courant Institute of Math. Sciences, New York, 1983.

9. C J Skelton, C Hammer, M Lopez, M J Reeve, P Townsend and K F Wong, *EDS: a parallel computer system for advanced information processing.* Proc. PARLE-92, the Conference on Parallel Architectures and Languages Europe, Paris, June 1992. Published as LNCS 605, eds. Etiemble & Syre, Springer-Verlag, 1992, pages 3 - 18.

10. Arvind, R S Nikhil, K K Pingali, *I-Structures: Data Structures for Parallel Computing.* MIT LCS (CSG Memo 269), Cambridge, Mass. Feb 1987.

11. Persistent Programming Research Group *PS_algol Reference Manual,* Fourth Edition, Persistent Programming Research Report No 12. Department of Computing Science, University of Glasgow and Department of Computational Science, University of St Andrews 1987.

12. S H Lavington, M Standring, Y J Jiang, C J Wang and M E Waite, *Hardware Memory Management for Large Knowledge Bases.* Proceedings of Parle, the Conference on Parallel Architectures and Languages Europe, Eindhoven, June 1987, pages 226-241. (Published by Springer-Verlag as Lecture Notes in Computer Science, Nos 258 & 259).

13. I Robinson, *A Prolog Processor Based on a Pattern Matching Memory Device.* Proceedings Third Int. Conf. On Logic Programming, London 1986, pages 172-179. (Springer-Verlag, LNCS 225).

14. S H Lavington, *Technical Overview of the Intelligent File Store.* Knowledge-Based Systems, Vol. 1, No. 3, June 1988, pages 166-172.

15. S H Lavington, J M Emby, A J Marsh, E E James and M J Lear, *A Modularly Extensible Scheme for Exploiting Data Parallelism.* Presented at the Third International Conference On Transputer Applications, Glasgow, 1991. Published in Applications of Transputers 3, IOS Press, 1991, pages 620-625.

16. S H Lavington, M E Waite, J Robinson and N E J Dewhurst, *Exploiting parallelism in primitive operations on bulk data types.* Proceedings of PARLE-92, the Conference on Parallel Architectures and Languages Europe, Paris, June 1992. Published by Springer-Verlag as LNCS 605, pages 893-908.

17. J Bocca, M Dahmen, G Macartney, P Pearson and A Berthier, *KB-Prolog user guide, January 1990.* Available from the European Computer Industry Research Centre (ECRC), Arabellastr. 17, D-8000, Munchen 81, Germany.

18. N E J Dewhurst, S H Lavington and J Robinson, *DDB graph operations for the IFS/2.* Department of Computer Science, University of Essex, Internal Report CSM-169, May 1992.

19. S Reddaway, *High performance text retrieval with highly parallel hardware.* Proc. UNICOM Seminar on Commercial Parallel Processing, London, February 1992, pages 61 - 66.

20. A J Marsh and S H Lavington, *A synthetic join benchmark for evaluating DBMS/KBS hardware and software.* Department of Computer Science, University of Essex, Internal Report CSM-173, July 1992.

21. C J Wang and S H Lavington, *SIMD Parallelism for Symbol Mapping.* Proceedings of the International Workshop on VLSI for Artificial Intelligence and Neural Networks, Oxford, September 1990, pages C38-C51. Published in: VLSI for Artificial Intelligence and Neural Networks, ed. Delgado-Frias & Moore, Plenum Press, 1991, pages 67-78.

22. B L Rosser, J M Bedford and C R Dobbs, *The Ferranti Prolog Database Engine.* Ferranti International Report No. ASP/AI/REP/68, Sept. 1992.

23. S H Lavington, J Robinson and K Y Mok, *A High Performance Relational Algebraic Processor for Large Knowledge Bases.* Presented at the International Workshop on VLSI for Artificial Intelligence, Oxford, July 1988. Published in: VLSI for Artificial Intelligence, eds. Delgado-Frias and Moore, Kluwer Academic Press, 1989, pages 133-143.

(Note: references 2, 4, 5, 9, and 19 describe projects for which Chapters have been invited. Cross-reference to pages within this book should therefore be obtained eventually).