A NOVEL ARCHITECTURE FOR HANDLING PERSISTENT OBJECTS

S H Lavington

Internal Report CSM - 170, May 1992

*(This is an Invited paper, which has been accepted for publication in a Forthcoming Special Issue of "**Microprocessors and Microsystems**" devoted to Persistent System Architectures.)*

Department of Computer Science
University of Essex
Colchester
CO4 3SQ
UK

Tel: 0206 872 677
e-mail: lavington@uk.ac.essex

A NOVEL ARCHITECTURE FOR HANDLING PERSISTENT OBJECTS.

S.H. Lavington

Dept. of Computer Science, University of Essex, UK.

Abstract.

The storage and processing of persistent objects presents opportunities for add-on hardware support. Particularly, there are aspects of memory management and whole-structure processing which are seen to benefit from a novel kind of SIMD-parallel architecture. This paper discusses the top-down requirements for handling persistent objects and then describes an active memory unit called the IFS/2 which meets these requirements. Performance figures for prototype hardware are given, and its application to an object-oriented data model is briefly described.

1. Introduction.

The inter-related areas of persistent storage, object-oriented programming languages, and object-oriented databases are very much top-down responses to the perceived inadequacies of conventional computing environments of the 1970s. Central to all three areas is the notion of an object. However, the precise meaning attached to the word 'object' differs subtly according to the context in which it is being used. More particularly, the applications programmer's notion of an object may not always reflect the necessary physical activity at the lowest level of a computing system. This is because the actual hardware architecture of most computing platforms is exactly the same today as it was in the1970s.

Although there is some economic pressure against the introduction of radical computer architectures, the seemingly unstoppable increase in software complexity compels researchers to experiment with novel hardware structures. The aim of such novel hardware is (or should be) to offer cost-effective solutions to the twin problems of slow and complex software.

In this paper we examine issues in persistent object management which cause conventional software to become slow and complex. We deal with both storage and processing topics. Persistent objects highlight the need for a mechanism whereby the type system of a programming language may become enforceable on all relevant data structures, irrespective of where these data structures are physically held. This, in turn, benefits from a memory management strategy based on a uniform naming scheme. Since most memory-management problems arise because of the differences between logical and physical storage, there is some advantage in devising architectures that have no dependency on physical addresses. Associative (i.e. content-addressable) memory has this property. Returning to the enforcement of type systems, there is a component of an object's method that is applications-independent. This section of code is

concerned with primitive operations on the object viewed as an abstract data type. An architecture which incorporated whole structure operations on stored objects could reflect at the hardware level some of the higher-level notions of ADTs.

The novel architecture described in the paper embodies the two concepts of associative memory and whole-structure processing. In Section 2 we analyse the requirements of object management, focusing on generic functionality that seems capable of being supported at the hardware level. Sections 3 & 4 present a tuple-based formalism for the low-level representation, management, and associative processing of objects. We introduce the notion of *active memory* and, in Section 5, describe a hardware realisation based on SIMD-parallel search engines controlled by transputers. The active memory unit, known as the IFS/2, has a command interface which is outlined in Section 6. Finally, in Section 7, we give an overview of a sample application which uses the IFS/2 for managing persistent objects.

2. Requirements.

Languages and database systems which support persistent objects require some form of run-time object-manager that unifies RAM and disc. This manager should allow users to access objects without any a priori knowledge of their physical location. In addition, a persistent object manager should provide:

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

From a programming viewpoint, persistence is made easier if the names used for objects at the applications programming level are used to access objects at the storage level, independent of memory technology. This implies some form of universal identifier, or uid. Such a truly global naming convention necessarily decouples the programmer and run-time code from all forms of physical location information. The usual decoupling, achieved via a virtual-to-real address mapping, can of course be used with very large addresses to support persistence[1]. Alternatively, the concept of a name could be taken through to the memory units themselves, which then become associative (i.e. content addressable). We examine this notion further in Section 4.

In order to be cost-effective, any underlying hardware support for persistent object management should not become locked into too narrow a view of objects. For example, relations are a powerful semantic construct which is often missing from object-oriented languages. However, it is felt by many that relations should ideally be included in an object-oriented data model [2]. The inclusion of relations improves the design and partitioning of large information systems and the representation of complex objects. Furthermore in several applications, for example MIS, CAD, and geographical systems where complex objects are encountered, object equality is just as useful as object identity. The notions of identity and value are to be seen, for example, in FAD[3] and OPAL[4].

In the above respects, the requirements for an object manager merge with the requirements for a database or knowledge-base server. There is therefore some advantage in biassing hardware

support towards generically-useful data structures such as relations, sets and graphs, which underpin most information systems. Direct hardware support for well-understood primitive operations on these structures, for example pattern-directed search and transitive closure, could remove considerable organisational and processing burdens from software. The correct choice of primitives could contribute usefully to the simplification and speeding-up of a wide range of information systems. Furthermore, there is much scope for the automatic exploitation of data-parallelism - a topic we return to in Section 4.2. The challenge is to present these technological features in a manner which gives direct benefit to end-users. This means designing to acceptable hardware and software interfacing standards and providing applications programmers with easy migration paths.

In the next two Sections we present a general (i.e. applications-independent) scheme for the storage and processing of objects of varying granularity. The scheme is designed to be semantics-free. The application to a particular object-oriented data model is discussed in Section 7.

3. A formalism for representing and selecting objects.

3.1 Objects as tuples

In a compromise between generality, efficiency, and ease of use, we assume that objects can conveniently be viewed at three levels: atomic, primitive and composed. Borrowing from relational terminology, we call the primitive object a *tuple* and the composed object a *tuple-set*. Thus, tuples may be used to represent instances of a class and tuple-sets may be thought of as classes. The *make-tuple* constructor is introduced for building tuples out of atomic objects, and the *make-tuple-set* constructor builds composed objects as sets of tuples. Within a computing system atomic objects are, as their name implies, indivisible. The partitioning and mapping of real-world concepts and entities into atomic objects is a matter for the information-modeller. Atomic objects are characterised as either *constant* (i.e. ground) or variable. In order to avoid programmers making unwarranted assumptions, we choose the more neutral phrase *wild card* in place of 'variable'. Wild card atoms are further sub-divided into *named wild cards* and *un-named wild cards.*

More formally, let there 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).

The bit-patterns of constant atoms can either represent actual values (e.g. bit, byte, signed integer, floating-point, etc.) or they can be tokens standing for other objects. For example, a constant atom might be used as a lexical token or internal identifier, standing for an external character string. A constant atom could, indeed, be used as a token standing for a whole file of pixel data. A concept (or abstract entity) might also be given a token, as could the short-hand 'label' of a tuple. In object-oriented systems constant atoms would be used to represent attributes, object IDs, or instance-references. Finally, a constant atom might be used to represent some higher-level <type> information.

Using the constructor *make-tuple,* we can compose primitive objects as collections of atomic objects. Tuples may be of any length, and may consist of any choice of component atoms. The ith tuple thus has the general format:

$$Ti = <Ai_1, Ai_2, Ai_3, ..., Ai_m>,$$

where $Ai_1, Ai_2, ..., Ai_m \in D$. 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 (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.

3.2 Variables as Constraints

The presence of wild cards in the fields of a tuple focuses attention on the precise interpretation of equality during operations such as the retrieval of tuples. This is clearly important in deductive databases and knowledge-based systems. Unlike simple relational selection, the action of matching an interrogand tuple (the 'query') against a stored tuple is capable of several interpretations or modes. Each mode represents a particular case of pattern-directed searching

There are two senses in which the interrogand tuple and a stored tuple may be said to be equal: they may be identical or they may be unifiable. 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.

We 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 can tabulate five of the more obvious modes as in Table 1.

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 the table, identity matching is a meta-level mode used, e.g., for systems housekeeping. Simple matching is the basic 'look-up' search implemented by conventional Content-Addressable Memory (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.

4. Low-level storage and manipulation.

4.1 Memory management.

From the requirements presented in Section 2, it is evident that the tuples of Section 3 should be held in an associative, i.e. content-addressable, memory. Several problems have to be solved. Firstly and naturally, a mechanism has to exist for the insertion, deletion, pattern-directed searching and retrieval of tuples - according to the semantics of search presented previously. Secondly, sets of tuples created by one user have to be distinguished during searching from similar, but unrelated, tuple-sets created by other users; in short, the associative memory has to be segmented according to logical (not physical) sets of objects. Because there may be several concurrent users, the associative memory has to have some form of protection or locking mechanism. Finally, very large numbers of tuples have to be accommodated; this inevitably implies a hierarchy of associative memory technologies, with appropriate caching and 'paging'

mechanisms.

Solutions to the above problems are to some extent driven by technology. We briefly review the possibilities, and then present an overall memory management strategy. Details relevant to the higher-level user are postponed to Sections 6 and 7; applications programmers may skip to Section 6 if they feel nervous about hardware!

Semiconductor CAM chips, such as the AM99C10 which holds 256 entries of 48 bits each with a 100nsec. access-time, are too expensive for all but the smallest object-caches. We have developed a technique based on hardware hashing and multiple SIMD search engines[5,6], whereby modules of conventional DRAM can be used as pseudo-associative memory. This permits us to build several tens of Mbytes of semiconductor CAM at a price per bit which is little more than that of RAM. The down-side is that this SIMD CAM has search times about 1000 times slower than those of special-purpose chips such as the AM99C10. We return to the speed issue in Section 4.2. The SIMD CAM acts as a cache to discs, which are searched associatively 'on-the-fly' in a manner similar to that pioneered by the ICL CAFS[7]. Further details of our most recent hardware implementation, known as the IFS/2, are given in Section 5.

The natural unit of access for an associative tuple store is a sub-set of one of the many possible sets of tuples which exist in memory. For example, if tuple-set Si contains the 'lives.in' relation, then the interrogand:

<Si> < ? ><lives.in><Ipswich>

describes a logical set of desired responders, namely, all those individuals within the <Si> segment of memory who satisfy the condition stated in the rest of the query. Thus, interrogand-tuples may also be treated as logical set descriptors.

Similarly, the natural unit of information for 'paging' and protection in an associative memory is also the logical set - (thus by-passing the phantom problems inherent in physical sets). Each logical set has a descriptor, similar to the example above. Descriptors can contain varying numbers of un-named wild cards. Thus sets can be of varying granularity ranging from the single tuple (eg '<Joe lives.in Ipswich>') to a whole set of tuples, or indeed to the whole contents of the associative tuple store. In[5] we give further details of a paging scheme based on set descriptors, known as semantic caching. Briefly, the associative cache holds normal tuples and also the set-descriptors of logical sets which are currently resident in cache. Upon receiving a search-query, relatively simple tests on set descriptors will determine whether the desired set of responders is wholly, partly, or not at all resident in the cache. We also propose to use set descriptors as lock specifiers for concurrent-user control, with a mixture of pre-conflict locking and post-conflict ('optimistic') validation[5].

4.2 In-situ processing: the active memory principle.

Amongst the objectives of all parallel architectures are: (a) minimising unnecessary movement of information; (b) minimising the programmer-effort involved in exploiting inherent parallelism. In data- intensive applications such as OODB, DBMS and KBMS, these two objectives become the more difficult to achieve because of the mis-match between the fixed units of access of conventional storage devices (eg disc, RAM) and the rich variety of objects to be handled by data-processing software. In contrast, associative memory of the form described in the previous

Section seeks to smooth the handling of variable-sized objects by providing access by name and by content. We may still, however, be left with data-movement and programmer-effort problems.

To combat unnecessary data-movement and programmer-effort, we introduce the notion of *active memory* whereby frequently-used data structures are both stored and manipulated within the same physical unit. This approach becomes practical if we can identify such frequently-used structures and also provide appropriate primitive operations for them. A study of many information systems, spanning the DBMS-to-AI spectrum, suggests that structures such as sets, relations and graphs are generic to the great majority of non-numeric applications. There are well-defined primitive operations on these structures, for example: selection, intersection, join, transitive closure and shortest path. A fuller discussion is given in Section 6. All such primitive operations have in common the need to carry out simple value-comparisons on data-elements - often in 'any' order. This suggests a natural form of inherent parallelism which could be exploited automatically by the same SIMD-associative techniques that we employ for the IFS/2 memory systems alluded to in Section 4.1.

In summary, then, we propose a one-level active memory unit which carries out whole-structure processing on a useful range of data types. These data types are represented according to a general tuple convention as described in Section 3.1. Tuples and tuple-sets may be used to represent objects of varying granularity, which are managed within an associative memory according to the 'paging' and protection scheme discussed in Section 4.1. Tuples and tuple-sets will certainly not be appropriate to represent all the data structures necessary for a particular application, nor does the active memory aspire to carry out general computation. Therefore, the active memory is seen as an add-on extra to a conventional computing system - in much the same way as optional co-processors were introduced some years ago to handle vector or graphics activity. Part of the active memory's practical design objectives are thus to achieve smooth hardware and software integration with the standard hardware and software interfaces of conventional computing systems. We now describe a prototype hardware realisation.

5. Parallel hardware implementation.

The SIMD techniques introduced briefly in Section 4.1 and described more fully in[6] were used to produce a network-accessible Knowledge-base server known as the Intelligent File Store (IFS). Three IFS/1 production units were built, the first being installed at the Artificial Intelligence Applications Institute (Edinburgh University) in December 1987. This contained 6 Mbytes of semiconductor associative storage. Based on this experience, we have recently built a prototype active memory unit called, for historic reasons, the IFS/2. The IFS/1 and its successor, the IFS/2, are shown in the photograph of Figure 1.

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, acting as associative cache to an associatively-accessed SCSI disc. The arrangement is shown in Figure 2. The present implementation has nine search engines and 9 Mbytes of cache and 700 Mbytes of disc per node, and three nodes. The 27 Mbytes 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 three relational algebraic buffers used in set and graph operations within the active memory unit. The

Figure 1: 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.

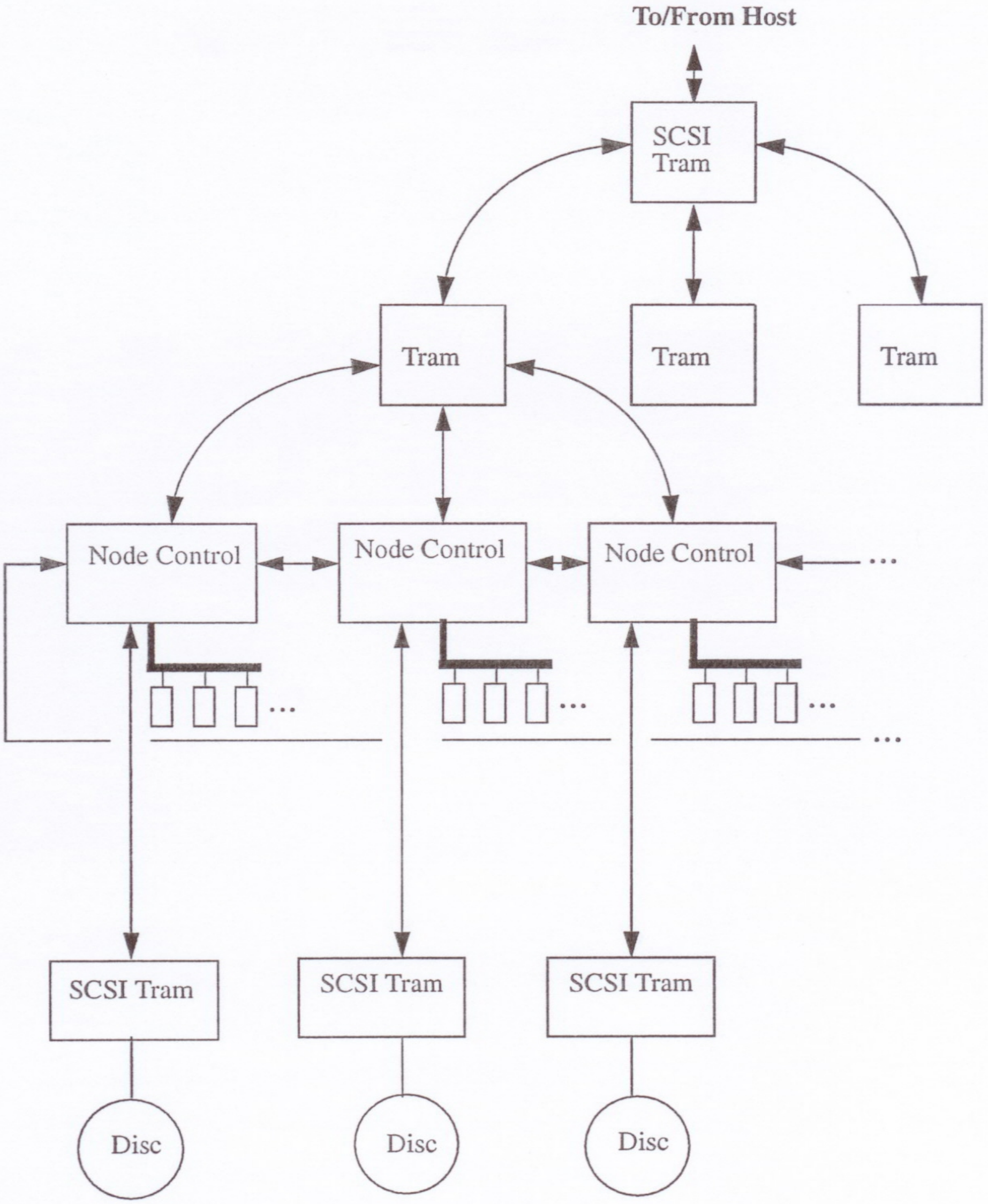

Figure 2: THE PROTOTYPE IFS/2

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. It can be seen from Figure 2 that the link to the host computer is via a standard SCSI channel.

The two prototyping cabinets of the IFS/2 shown in Figure 1 contain a total of 13 4-layer double-Euro pcbs: a) a general-purpose motherboard containing four standard SCSI Transputer modules (TRAMS), three to control the discs and one for the host connection; b) three node-controllers, each based on a T425 plus 2Mbytes RAM and dedicated logic plus two standard T801 TRAM daughter-board positions; c) nine SIMD search modules, each pcb containing three banks of 1Mbyte 80 nsec. DRAM and three search engines. The transputers run at 25MHz.

The implementation of the IFS/2 active memory unit is described more fully in [8]. Briefly, typical operating times for the prototype are as follows, when storing 276,480 3-field tuples within the 27 Mbytes of semiconductor associative memory.

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

Join of two 1000-tuple relations:

$R_{(1,2,3)} \bowtie_{=} S_{(1,2,3)} = R \cap S$: up to 90 millisecs.

The potential of the IFS/2 hardware is 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 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 relation's cardinality after the join was about 10 percent of the input relations' cardinality.

Test 1: $R \bowtie_{3=1} S$ Test 2: $R \bowtie_{3=2} S$

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

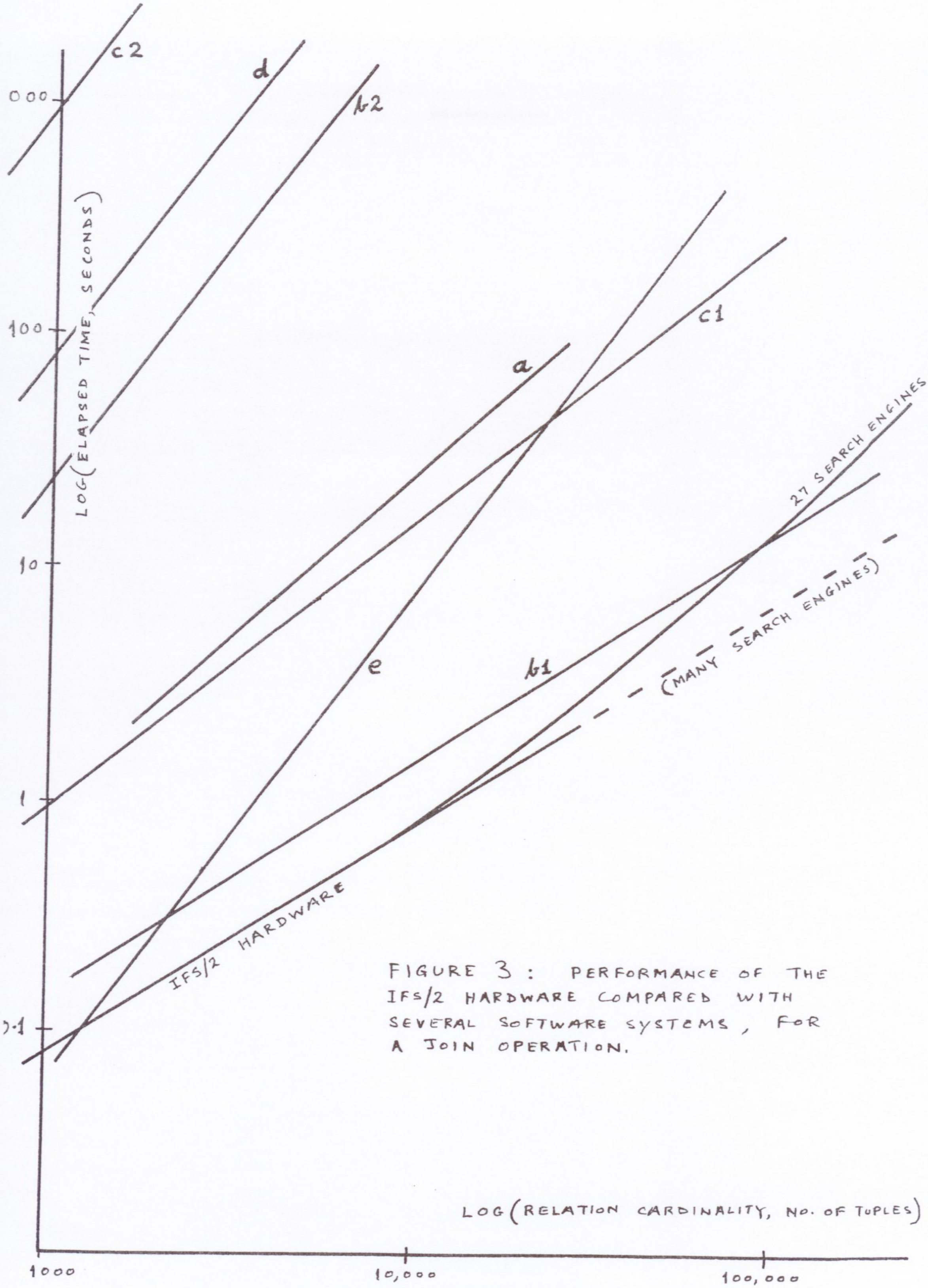

FIGURE 3: PERFORMANCE OF THE IFS/2 HARDWARE COMPARED WITH SEVERAL SOFTWARE SYSTEMS, FOR A JOIN OPERATION.

a) the INGRES relational DBMS
b_1, b_2) Quintus PROLOG
c_1, c_2) Kyoto Common LISP.
d) The IFS/2's software simulator, written in C
e) a hand-coded C program written to optimise the join command

The results are plotted in Figure 3 as elapsed time versus relation cardinality, on a log-log scale. The two choices of join-field in Tests 1 and 2 produced markedly different results for PROLOG and LISP, indicating indexing on first argument only, but substantially the same sets of times when the two tests were performed with INGRES and the C programs. Note that the LISP and PROLOG programs were written so as to maximise performance, even though the strategy used (namely arrays rather than lists) appeared unusual for the language. For example the PROLOG program started by reading the test data from a file and then asserting the relations, known as seta and setb. It then used the following rule for Test1:

```
join(_) :- seta(A,B,C), setb(C,D,E), assert (int(A,B,C,D,E)), fail.
```

The Quintus run-time was measured for execution of this rule.

It is seen from Figure 3 that the IFS/2 may be expected to perform whole-structure operations between 2 and 10,000 times faster than conventional software. Note that the times given in Figure 3 for the IFS/2 hardware are known to be capable of improvement; the prototype design is currently being refined.

The following general observations may be made on Figure 3:

(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, Figure3 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.

(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 of cardinalities under about 3000 tuples. Above that size, the IFS/2 performance curve can be kept linear by adding more search modules.

6. High-level and low-level primitives.

6.1. The high-level view.

The IFS/2's tuples and tuple-sets may be manipulated by the standard operations associated with the super-type *relation*. The full list of commands is the subject of current research; at present it includes:

(a) insert, delete, pattern-directed search;
(b) the diadic operations of relational algebra such as union, intersection, difference, extended

product (includes join), and division;

(c) aggregate operations (sum, average, minimum and maximum over a given column of a given relation);

(d) graph operations for supporting recursive queries, namely: transitive closure, reachable node set/subgraph, nth wave of nodes/tuples, return paths between two nodes, composition of relations, is a graph/reachable subgraph cyclic? return a set of waves.

The repertoire of graph operations, in particular, is the subject of planned review and probable extension.

The IFS/2 actually supports two different kinds of tuple-set: persistent and transient.They differ principally in their behaviour when passed as arguments to the relational operations in group (b) above. After being used as arguments to group (b) functions, transient tuple-sets are automatically removed from the IFS/2's associative memory, whereas persistent sets are not removed. Transient sets are, as their name implies, intended to hold temporary results which arise during the course of evaluating a complex query. When one of the group (b) operations produces an output set, this is automatically declared to be transient. In contrast persistent sets, which are the more general case, are declared explicitly by the IFS/2 user. IFS/2 procedures exist for making a transient set persistent, and vice versa.

6.2 low-level command interface.

The above higher-level primitives are mechanised via a low-level interface which consists (at present) of 42 C library procedure calls. The library procedures, running on a host computer, communicate with the actual IFS/2 hardware via a SCSI device driver. The 42 IFS commands are grouped as follows:

5 for housekeeping (e.g. opening and closing the IFS/2);
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;
3 for inserting and deleting tuples, and pattern-directed search over sets of tuples;
5 for handling character strings or similar lexical tokens; these are converted into fixed-length internal identifiers - (see [9]);
5 for label management; these commands help in particular with fast label dereferencing - (see [9]);
7 for the relational algebraic operations of groups (b) and (c) of Section 6.1;
14 for the graph operations of group (d) above.

As examples, we give the C procedure call for the IFS/2's general pattern-directed search command and then the call for a relational algebraic command that is the generalisation of join. The first command has the 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 3.1) 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 (excluding derived 'working' sets) as persistent.

When manipulating structures, e.g. via relational algebraic operations, 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. Chris will be a transient structure. 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**.

7. Application to an Object Oriented data model.

We have used the commands of Section 6.2 to support a specimen Semantic Object-Oriented Model (SOOM) written in C++. Our model supports relations between classes, so as to offer a richer information-representation capability than is possible with systems taking a more classical view of data encapsulation. Briefly, persistent data objects in SOOM are stored as two types of tuple[10], having typical formats:

(a) <object id><instance reference><relation><attribute>
(b) <object id><instance ref. 1><relation><object id><instance ref. 2>

The inclusion of semantic relations between data objects combines the object oriented paradigm with a flavour of the entity relational model. The SOOM system allows the identification of objects either by identity or by value. Promoting relations to an equivalent level of abstraction as classes permits a symmetric and compact representation of highly interconnected systems and complex objects. Using the IFS/2 active memory to support SOOM results in a very flexible and powerful object server.

SOOM is written in an extension of C++, which incorporates six new functions for interfacing with the IFS/2 active memory commands of Section 6.2. The six new functions support the

following features for static objects[10]: persistence, relations between classes, object equality and instance inheritance. The function-names are: *relation, create, get, find, card*, and *destroy*. These are built on top of the IFS/2 C host library procedures, which in turn act as the interface to the attached IFS/2 hardware unit. For example, the *find* function invokes the full associative matching power of the IFS/2's pattern-directed search, the command format of which is given in Section 6.2.

8. Conclusions.

The tuple and tuple-set constructors of the IFS/2 constitute a flexible formalism for the low-level representation of complex objects of varying granularity. The IFS/2's scheme of semantic caching and protection provides memory management for these tuple-sets in a manner that decouples objects and object-references from the constraints of physical storage technology. Particularly, objects are referred to by name and value since the IFS/2's entire one-level memory is associative (i.e. content-addressable).

The IFS/2 is presented as an add-on *active memory* unit, connected to a host computer by a standard SCSI interface. Its low-level commands are available to host programmers via C library procedure calls. These activate a repertoire of whole-structure primitive operations in the attached IFS/2. Operations such as pattern-directed search and transitive closure are available on well-understood structures such as relations, sets and graphs so that, if objects can be modelled in terms of these structures, then the IFS/2 gives hardware support to a useful range of ADTs. Thus, that portion of an object's method (i.e. behavioural schema) which is applications-independent and linked to a <type> system could be given direct support at the data-storage level.

The above features have an inherent benefit in simplifying and reducing the quantity of object-management software. Additionally, performance measurements have shown that the IFS/2 is between two and 10,000 times faster than conventional software when performing whole-structure operations such as *join*. This speed-up arises from the use of a novel form of SIMD parallelism both to store and process data in situ.

9. Acknowledgements.

It is a pleasure to acknowledge the contribution of other members of the IFS team at Essex. Particularly, Jenny Emby and Andy Marsh contributed to the IFS/2 hardware design; Bob Davies implemented the SOOM software; Martin Waite, Jerome Robinson and Neil Dewhurst contributed to the formal aspects. The work described in this paper has been supported by SERC grants GR/F/06319, GR/F/61028 and GR/G/30867.

10. Reference

1. Rosenberg J and Keedy J L "Software Management of a Large Virtual Memory". Proceedings of the 4th Australian Computer Science Conference, Brisbane, February 1981, pages 171 - 183.
2. Rumbaugh J "Relations as Semantic Constructs in an Object-Oriented Language"

Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, October 1987 pages 466 - 481. (Published by acm press, Special Issue of SIGPLAN Notices, Vol.22, No.12)

3. Bancilhon F, Khoshafian S and Valduriez P "FAD, a Simple and Powerful Database Language" Proceedings of the 13th International Conference on VLDB, Brighton, England September 1987.

4. Maierand D, Stein J "Indexing in an Object Oriented DBMS" Proceeding of the 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, California, 1986.

5. Lavington S H, Standring M, Jiang Y J, Wang C J, and Waite M E, "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)*

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

7. Maller, V.A.J. "Information retrieval using the Content Addressable File Store". Proc. IFIP - 80 Congress, pages 187 -192. Published as Information Processing 80, ed. Lavington, North Holland, 1980, pages 187 - 192.

8. Lavington S H , Emby J M, Marsh A J, James E E and Lear M J, "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.*

9. Lavington S H, Waite M E, Robinson J and Dewhurst N E J, "Exploiting Parallelism in Primitive Operations on Bulk Data Types." *Internal Report CSM-166, December 1991. Paper accepted for PARLE-92, Paris, June 1992.*

10. Lavington S H and Davies R A J, "Active Memory for Managing Persistent Objects". Presented at the International Workshop on Computer Architectures to Support Security and Persistence, Bremen, May 1990. Published in: Security and Persistence, Ed Rosenberg & Keedy, Springer-Verlag 1990, pages 137 - 154. Further practical details in Internal Report CSM-155, August 1990.