A Modularly Extensible Scheme for Exploiting Data Parallelism.

S. H. Lavington[a,1], J. M. Emby[a], A. J. Marsh[a], E. E. James[a] and M. J. Lear[a]

[a]*Dept of Computer Science, University of Essex, Colchester, CO4 3SQ, UK.*

Abstract. Non-numeric (e.g. symbolic) information systems contain much natural parallelism. This parallelism becomes evident in frequently-occurring tasks such as pattern-directed search and set intersection. We describe an add-on unit, based on transputers, which exploits data parallelism by performing set and graph operations in situ within an associative memory frame-

Keywords. Transputers, associative, parallelism

1. The Problem

Applications such as real-time command and control or the handling of financial data feeds to on-line share dealing systems demand the rapid manipulation of complex data. The dynamic and heterogeneous nature of the data, combined with the variety of access criteria, present significant technical challenges to the database designer. Furthermore, information systems are increasingly required to exhibit 'intelligence'. It is observed that the smarter the information system the slower and more complex becomes the software. Speed improvements of two orders of magnitude are commonly called for.

When generic functionality is analysed it is clear that smart information systems contain much natural data parallelism. For example, operations such as pattern-directed search or set intersection imply the comparison, in 'any' order, of many data items. The challenge is therefore to devise an architecture which takes advantage of this natural parallelism to provide cost-effective speed-up, whilst simplifying software. 'Cost-effectiveness' is not only a matter of appropriate technology; it also implies an architecture which is applications- independent, provides easy software and hardware migration paths (i.e. easy integration into present platforms), and is modularly extensible.

In this paper, we describe a novel prototype hardware system which addresses the above problems of slow and complex software. Specifically, we describe an add-on structure store or active memory unit called the IFS/2, which supports smart information systems such as knowledge bases. The IFS/2 uses transputers for distributed control and distributed processing. In Section 2 we summarise the design requirements and present a scheme for information representation. In Section 3 we

---

[1] Corresponding Author: Simon Lavington; E-mail: lavis@essex.ac.uk.

discuss a SIMD architecture incorporating transputers. Finally, we describe the implementation and performance of the transputer-based IFS/2.

2. Developing a suitable architecture.

2.1. Requirements.

Data-intensive information systems are often required to support concurrent users, so that the concept of shared memory with a single access-control point is a suitable starting point. Sometimes the shared memory may be partitioned logically into several independent datasets or 'segments'; if so, it may be desirable to control access to each logical segment. The shared memory contains data structures (or objects) of varying granularity; these can very often be described by the super-type relation (which clearly covers sets and graphs). The operations on this super-type include the normal relational algebraic commands such as selection and join, plus operations such as transitive closure and other graph manipulations to be used in handling linear recursive queries, etc. Since the data will in general contain variables, 'selection' may take the form of one-way or two-way pattern-matching - ideally including structure-matching (via automatic label dereference), so as to support unification. Finally, there is a need in some information systems for nearness-matching, using Euclidean or Hamming distances as the metric.

The above requirements are discussed more fully in [1], where it is shown that associative (i.e. content-addressable) access is central to the desirable memory-based functionality. Furthermore it is shown that if the memory unit could also carry out certain well-used operations on structures in situ we should be able to greatly reduce the bandwidth requirements between a shared memory and the normal locus of computational control (i.e. the CPU). This is the active memory principle. In short, 'operations should be moved to the data' rather than vice versa. This should form a suitable architectural framework within which to exploit the natural data parallelism exhibited by many information systems.

2.2. Low-level information representation.

Devising an appropriate representation for non-numeric (e.g. symbolic) data in associative memories is still very much a research issue - (see for example [2] and [3]). Based on our experience with a knowledge-base server known as the IFS/1 ([4]), we start with a flat domain of primitive entities and the constructors tuple-of and set-of. In our prototype active memory, known as the IFS/2, entities are 32-bit fields which can be used by software to represent symbols, object-IDs, integers, short floating-point numbers, lexical tokens (see later), labels, variables, type-descriptors, abstract nodes, etc. The fields are concatenated to form tuples, with the following format:

<system control word><home-label><field 1><field 2>. . . <field n>.

The <system control word> contains a valid/invalid marker, a count of the total number of 32-bit words in this tuple and the <class-number> for this tuple. <class-number> refers to an entry in a Tuple Descriptor Table (TDT). Class numbers, which are effectively used to segment the IFS/2 associative memory, are assigned by IFS/2

firmware when a tuple-type is first declared. On first declaring a tuple-type, the user defines the categorisation of each field as being either ground, named wild card (i.e. named variable), un-named wild card, (i.e. anonymous variable), or label. This information is kept in the TDT. Furthermore, tuple-types are defined in the TDT to be either labelled or un-labelled, so that the <home-label> field above is optional.

In the prototype IFS/2, all entities have to be modelled in terms of fixed-length 32-bit fields. In the prototype there can be up to 2^{16} classes (i.e. logically-distinct segments), and a tuple can have up to 2^7 fields. For character strings, the IFS/2 has a separate associative dictionary called the Lexical Token Converter ([4]) which maps variable-length ASCII character strings into fixed-length 32-bit unique lexical tokens and vice versa.

1FS/2 <home-labels> start by being strict, somewhat like Gödel numbers. That is to say, a <home-label> is a unique system-allocated token which stands for a unique tuple. Some information models, e.g. belief systems, will wish to preserve this strictness. Other models will wish the <home- label> to remain fixed even though one or more modifications (updates) have been made to a field in its tuple; this is a congruence interpretation. The IFS/2 performs automatic label-dereferencing in either case, though the congruence case takes about twice as long because of the necessary redirection.

Tuples are held in the IFS/2's associative memory. When performing a search, an interrogand takes the form of a tuple with 0, 1 or many fields wild - (but with the <class- number> always defined). This gives the mechanism for constructing sets of varying granularity. Responder-sets form the operands for diadic relational operations such as intersection; for these diadic commands the resultant set is given a <class-number> and automatically stored in the IFS/2's associative memory, where by default it becomes persistent. Memory management (i.e. 'paging' and concurrent-user access control) in the IFS/2's hierarchy of associative storage devices is carried out in terms of logical set descriptors. The 'paging' scheme, known as semantic caching, is described in [5].

Before moving on to describe the physical architecture, it is necessary to consider the choice of implementation technology and, in particular, the potential advantages of employing transputers.

3. Implementation framework.

In order to provide large quantities (e.g. Mbytes) of semiconductor associative cache at a reasonable cost per bit, a SIMD scheme of search engines and hardware hashing has successfully been used [4]. We have also used similar SIMD techniques in the construction of a relational algebraic processor [6]. In both cases, each search engine was effectively a simple comparator assigned to a module of DRAM. The many search engines operated in lock step under centralised control, the controller being implemented as several finite-state machines using TIL PAL/GAL technology. Hardware hashing, typically using 9-bit keys (512 hashing bins), was used to limit the area of SIMD search. The scheme has the advantage of modular extensibility: each increment of DRAM has its own comparator, so that the time for an associative search is independent of total memory capacity. In the IFS/1, ([1]), first delivered in 1987, 6

Mbytes of associative cache were backed by a 300 Mbyte associatively-accessed disc; the two units formed a one-level store using semantic caching for the 'paging' strategy.

For the new IFS/2, the strategy for control is quite different because of two factors: (a) the active memory architecture requires more processing activity, e.g. during two-way matching and unification; (b) the diadic operations such as set intersection require significantly more buffering and broadcasting activity for responders. The transputer's ease of interconnection and ability to 'overlap' processing and communication make it a very attractive component for distributed control and distributed processing. The transputer also offers a flexible prototyping environment compared with the labour-intensive (though higher-performance) implementation strategy for controllers based on PAL/GAL technology.

In Section 4 we describe the IFS/2's implementation in terms of *nodes,* where a node contains 3 transputers and a number of search engines. The nodes are connected together, via transputer links, through a common interface to a host computer. The totality of OCCAM firmware running on the node transputers supports the IFS/2's procedural interface. As far as user programs running on the host are concerned, this interface consists of calls to a collection of C library procedures. There are procedures for declaring, inserting, deleting, and searching for (sets of) tuples; performing the relational algebraic operations intersection, difference, union, join; and implementing recursive query primitives such as transitive closure, reachable node set, composition of relations, etc. For example, a call to the procedure:

ifs_search(matching_algorithm, code, c_n, query, &result)

allows one of a variety of pattern-directed searches to be carried out in parallel on all tuples in class c_n, as a single indivisible command.

4. Node Structure and performance.

Each IFS/2 node (Figure 1) consists of several parts: a T425 transputer which acts as the search-engine controller, (SEC) sharing its processing load with a T801, the unification processor, (UP) and using as backing store a 770 Mb hard-disc driven by a SCSI TRAM.
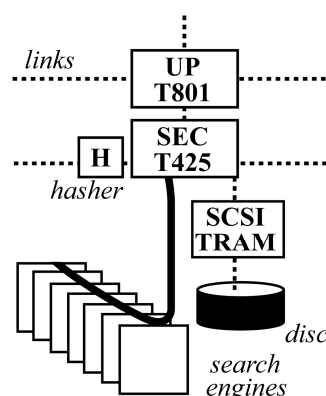

Figure 1: **A node**

The UP implements arbitrarily complex functions on a tuple that do not require much access to the search engine memory - e.g. the computationally intensive phases of unification. The UP links horizontally with other nodes and provides the path to host (see also figure 2).

The SEC resident firmware directly manipulates the search engines and can address them in parallel. Firmware also implements those higher level functions that require inter-node communication as well as further access to the search-engines e.g. *join*.

The SEC transputer currently runs at 25 MHz on EMI configuration 6, giving an external access time of

200 ns. Data transfer takes place in blocks of 300 bytes at a peak rate of 1.8 Mbytes^{-1}.

The T425 and T801 have 1 and 4Mbytes of DRAM local memory respectively. The SCSI TRAM effectively simulates on-the-fly associative searching of a given disc track.

Each search-engine is (at present) 1 Mbyte of 80 ns 256K x 4 DRAM plus a small amount of programmable array logic for implementing the various match options including masking. The engines slot into a common 32 bit wide databus and are all memory mapped to the same address block in the SEC's memory. As is described later, they are software partitioned (horizontally) into zones for different associative tasks and further sub-divided into hash bins (currently 2^9) to increase search rate by reducing the search space. Since the calculation of hash values is relatively slow by software, a 16 bit hash value is provided by a special hardware hasher (H) installed on the same pcb as the SEC. The hasher consists of 7 GALs and 2 PROM tables, and produces the initial hash value in the time it takes to write the tuple to memory (200 ns per 32 bit word) plus 200 ns per field.

The optimum number of search engines per node is an interesting compromise. The absolute hardware data-comparison time for 32 bits is 200 ns and is independent of the number of search-engines (n) at a given node since comparisons are done in SIMD parallel across all n engines. The data-comparison rate increases linearly with n, giving a corresponding potential increase in search rate. However all tuples successfully matched will require certain fields to be returned along a common bus, leading to a possible degradation in search rate. The optimal balance between a high degree of parallelism enabling tuples to be quickly identified and the limitations on retrieval posed by bus bandwidth are still being explored. Having many SECs with fewer engines at each would also keep the data- comparison rate high but would introduce extra communication latency problems. The general arrangement of nodes is shown in Figure 2. Three nodes have been implemented in the 1FS/2 prototype, with up to 9 search engines per node. The ability of the transputer to have CPU activity concurrent with data transfer on all four links is exploited. The modularity of design can be clearly seen.
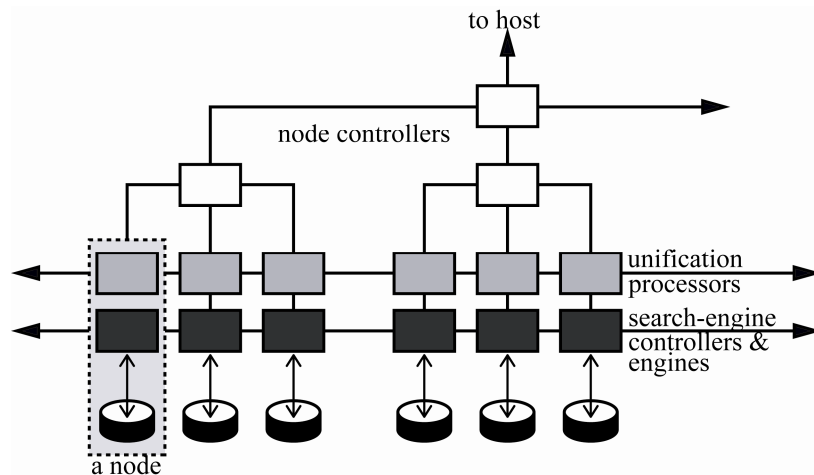

Figure 2: **System architecture**

Each search engine provides 1 Mbyte of content addressable memory that can be partitioned for use both as a cache for disc and as buffers for set and graph operations. The division between cache and buffers is flexible, but no more than half the available area would normally be used as cache (including overflow hash-bins). The buffer portion is used to hold intermediate working sets which are required to take part in element-by-element comparisons typical of operations on the super-type *relation* - see [6].

A typical operation proceeds as follows. We consider a simple search. The interrogand tuple, a mixture of ground and wild fields, is written to the hardware hasher and the set of hash bin values where responder tuples may be found is returned. For each hash bin value the search process is the same. The start address of the bin is calculated. A mask is set up to isolate the class number, c_n, within the system control word. Comparisons are done simultaneously on all the search engines at a node: a masked comparison is performed on each system control word in the bin: the ground fields of stored tuples of the correct class are then compared in detail to see if a complete match exists. The corresponding wild card fields of successfully matched tuples are then returned.

For an interrogand of class c_n, the time taken for specimen IFS/2 operations are (in µs):

Simple search (all fields known):	$(6np + 4(nc + 1)nfc)/2$
Simple search (with wild cards):	$nbs(9np + 14nc + 6s + 2r) + c$
Insertion:	$6(np + nc + 1) + 4nfc$
Deletion:	$nbs(9np + 14nc + 6s + 2e)$

Where
nc - av. no. of tuples in class c_n per search engine per bin per node
np - av. no. of tuples not in class c_n per search engine per bin per node
nbs - av. no. of bins to be searched
r - av. no. of responders (fields) per bin per node
e - av. no. of expected matching tuples per bin per node
s - av. no. of expected matching tuples per search engine per bin per node
c - communication time taken to return responders per node.

The number of bins searched (nbs) depends on the number of wildcards in the interrogand and, to a minor extent, on the position of the wildcard within the interrogand.

The following times were obtained for a 3-node IFS/2 with 9 search-engines per node and half the available associative memory reserved for relational algebraic buffers, when storing 3-field tuples and assuming no hash-bin overflow:

member	:-	16 µs (best case)
member	:-	73 µs (average case)
member	:-	130 µs (worst case)
search with one wild card	:-	1536 µs plus 22 is per responder.
search with two wild cards	:-	10176 µs plus 38 ts per responder.
deletion of a single tuple	:-	134 µs (worst case)
insertion	:-	260 µs (worst case)
cardinality of whole database	:-	0.5 s

The basic search times form the foundation of the more complicated diadic operations such as join. The implementation of these is currently in hand.

5. Acknowledgements.

References.

[1] S. H. Lavington & R. A. Davies, "Active memory for managing persistent objects". Presented at the Int. Workshop on Computer Architectures to Support Security and Persistence, Bremen, 1990. Published in: Security and Persistence, eds. Rosenberg & Keedy, Springer-Verlag, 1990, pages 137-154.

[2] P. Kogge, J. Oldfield, M. Brule & C. Stormon, "VLSI and rule-based systems". VLSI for Artificial Intelligence, eds. Delgado-Frias & Moore, Kluwer Academic Press, 1989, pages 95-108.

[3] I. Robinson, "The pattern addressable memory: hardware associative processing". VLSI for Artificial Intelligence, ibid., pages 119-129.

[4] S. H. Lavington, "Technical overview of the Intelligent File Store". Knowledge-based Systems, Vol. 1, No. 3, June 1988, pages 166-172.

[5] S. H. Lavington, M. Standring, Y. *3*. Jiang, C. J. Wang & M. E. Waite, "Hardware memory management for large knowledge bases". Proc. of PARLE, the Conference on Parallel Architectures & Languages Europe, Eindhoven, June 1987. Pub, by Springer-Verlag as LNCS Nos. 258 & 259, pages 226-241.

[6] S. H. Lavington, J. Robinson & K. Y. Mok, "A high performance relational algebraic processor for large knowledge bases". VLSI for Artificial Intelligence, ibid., pages 133-143.