On Supporting Associative Access and Processing over Dynamic Knowledge Bases

Ian N. Robinson
Hewlett Packard Laboratories,
1501 Page Mill Road, Building 3L,
Palo Alto, California, 94304,
U.S.A.

Abstract

Dynamic knowledge bases are a fact of life in many artificial intelligence applications. Using current techniques, however, it is not always possible to provide the desired level of associative access to them whilst meeting real-time, or even near-real-time, performance criteria. This paper argues the case for a hardware associative storage system that uses symbolic pattern matching as its access mechanism. A working prototype of such a system, designed as a co-processor for a workstation host, is then described. The co-processor is based on an array of custom designed VLSI 'smart memory' chips. These combine storage and search/processing logic on the same die. Parallelism is exploited both on chip and between chips to yield a high system performance. The paper concludes with some examples of how this hardware can be used to support real applications.

1 Associative Pattern Matching

Associative operations based on pattern matching are fundamental to many artificial intelligence applications. This paper concentrates on the broad category of knowledge-base systems. In such systems rules are activated based on pattern matching as the ordering information - found in procedural (or compiled) systems - is not present. When, and how, a particular fact or rule is to be applied is typically not known until run-time.

A major contributor to this uncertainty lies in the application's interaction with the everyday world. There is also the interaction between rules and, in more complex architectures, between component reasoning systems. In fact it is seen as an advantage to maintain associative access within such complex systems, rather than 'hard-wired' connections, since it facilitates modularization of the component parts [Reece and Shafer, 1990].

In general, however, this lack of predictability imposes a limit on how many of these associative operations can be optimized away at compile-time. Thus the run-time component will still contain significant use of associative access to the data structures encoding the knowledge base. Unfortunately such associative operations are computationally expensive. To mitigate this, various compile-time techniques have been developed to translate the knowledge-base data structures into code sequences. These use indexing operations, commonly based on hashing, to handle the associative lookup, e.g. Rete for OPS5 [Forgy, 1982J, WAM code for Prolog [Warren, 1977], HiPER for embedded knowledge-base systems [Highland and Iwaskiw, 1989].

As applications and their rules and queries become more complex, so too can the indexing schemes necessary to support the associative access. With simple data structures, finding one or two keywords to index on is fairly straightforward. As the structures and queries become more complex - possibly including the use of variables ('don't cares'), and with any of their elements being potential keywords - then correspondingly complex indexing structures are required (e.g. discrimination nets [Charniak *et al.,* 1980]). Otherwise a performance hit is taken every time a query does not conform to the chosen indexing scheme. Since the alternative is usually exhaustive linear search, such penalties can be costly.

These indexing schemes have acceptable performance when the database is reasonably static. However the more dynamic the knowledge base used by the application the greater the frequency of index table updates. Furthermore, the more complex the indexing scheme the greater the overhead involved in performing each individual update. Even relatively simple indexing schemes can be rendered impractical when the update rate of the entry indexed upon is too high. In fStormon, 1989] the example is given of applying complex queries to the value entries of a stocks and commodities database. With this value being updated hundreds of times a second it is not practical to index on it, and so answering such queries requires a sequential scan of the database records.

With the intricate indexing schemes necessary to efficiently access complex knowledge bases, even less

dynamism can be tolerated. Examples of such applications include intelligent control or monitoring systems such as in computer integrated manufacturing (CIM), medical and process monitoring, and autonomous vehicle and robot control systems. Typically such applications are required to reason about complex time varying systems - including themselves (e.g. dynamic resource allocation) - and face real-time constraints on the utility of their actions or decisions. The overhead involved in maintaining associative access in such dynamic applications severely impedes the ability of the system to meet its performance goals.

Even in systems where there is little pressure from outside events the internal dynamics can be considerable. Consider a system that reasons via hypothesis generation and test, or via constructing possible world scenarios. This involves adding new rules to the knowledge base at run-time, and employing them to test their efficacy. These rules may then be modified, or deleted and new ones added in their place. Maintaining an indexing scheme over such transitory data can entail considerable overhead.

In summary, these characteristics of unpredictability and dynamism move the onus of associative computation squarely back into the run-time system, where the associated overhead severely impacts system performance. The problem then is to support associative knowledge retrieval when the data structures encoding that knowledge can be quite complex and dynamic, with minimal overhead devoted to casting the structures into an efficiently accessible form.

2 Hardware: The Cache Analogy

Run-time being the domain of special purpose hardware, a solution is proposed based on associative memory techniques [Yau and Fung, 1977]. Software indexing schemes and their upkeep are by-passed altogether, and associative hardware is used to implement a rapid exhaustive search. The advantage of this approach is that the data structures are dealt with directly, with minimal to no encoding. Such use of associative hardware is not without precedent: consider the ubiquitous *cache memory.* The same characteristics of unpredictability and dynamism, though in this case of data and its location, gives rise to the requirement for run-time hardware to associatively match on addresses and return the associated data (or a signal that the data is not present - a 'miss').

This paper describes a cache-like system designed to operate with data structured in the form of symbolic *expressions.* Whilst a conventional cache supports rapid access based simply on memory addresses; this system has to support complex, non-predetermined, access to expressions of various lengths and structural complexity. By allowing control over access and modification of cached expressions some measure of associative processing can be supported. These additional capabilities lead the system to be configured as a co-processor rather than as a part of the host's memory system. Thus associative operations are off-loaded from the host CPU, just as graphics and floating-point co-processors off-load their particular tasks.

A significant body of work exists relating to hardware associative memories for such applications. Such hardware has typically been based on the traditional content-addressable memory (CAM) [Kogge *et al.,* 1989; Stormon, 1989; Wade and Sodini, 1989]. By comparison these suffer from restrictions on expression format, in some cases forcing entire stored expressions to be encoded into fixed length fields. Such approaches also tend to rely on comparators alone for the matching function, which presents problems when trying to capture the full syntax of symbolic expressions - both stored and used as queries. Lastly the storage organization used in the co-processor described here permits a significantly increased capacity over CAM-based architectures.

3 The Chameleon Board

Figure 1 shows a photograph of a wire-wrap prototype of this associative co-processor. Called the 'Chameleon board' (named after that creature's ability to pattern match with its environment) it is designed to be compatible with commercially available workstations, plugging in to the system's backplane.

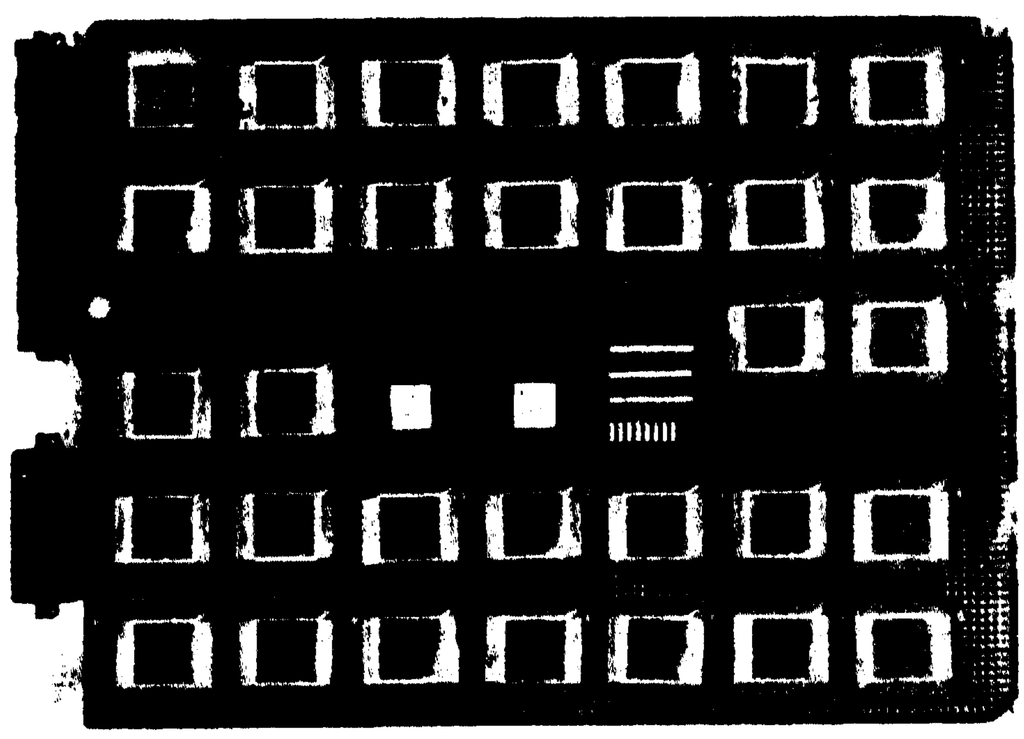

Figure 1. Chameleon Board

The computational heart of the system is the array of custom 'smart memory' chips called *pattern addressable memories* (PAM's). These combine memory for symbol storage with parallel and distributed processing logic. This logic is in the form of processing elements (PE's) replicated through the on-chip storage. The array of PAM chips acts as one combined storage system. All the PE's operate in a SIMD fashion on their respective areas of memory. The op-code and any data for the operation, e.g. match query, are broadcast by a central array controller (located in the center of the board in the photograph).

In the last three sections a hardware system has been presented that meets the stated requirements. The system supports inserting, deleting, modifying and associative matching on and with complex symbolic expressions, using parallelism to achieve high system performance. The current prototype PAM chip contains 1024 32-bit-symbol-plus-2-bit-status slots and 64 PE's, dividing the storage into 16 pages. The chip has been designed and fabricated using a 1.2u scalable design rule CMOS process, and has a die size of just 5.4mm x 4.9mm (approximately 140mils x 120mils). Using commercial design rules and die sizes, a PAM with a capacity of 1Mb and 128 PE's is quite practical.

The current PE cycle time is 200ns, this encompasses query symbol input, reading the current page, evaluation of the comparators and the jump wire, and the subsequent write back of the new match token state. The prototype Chameleon board contains an array of up to 32 PAM chips. Provisions are made for extending the SIMD operation of the array over multiple boards connected to the same host.

6 An Example: Scheduling in a CIM environment

By way of an example consider part of a simple factory floor scheduling application controlling mobile palettes. Entries in the database for each palette denote its location/destination, its current state and its contents. Thus the expressions loaded into Chameleon have the following form:

```
@palette (<#>  <dest.>  <state>  (<list_of_contents>))
@palette ( 1    Bay_14  moving  (bifrucated_congle_pins
                                 lug_nuts))
```

Such a database can then be queried with the following expressions:

```
@palette ( ?#   ?where  idle     ())
```

...find an empty palette.

```
@palette ( ?#   Bay_17  ?state   ( & lug_nuts & ))
```

...any lug nuts coming to, or currently at, Bay 17?

Times for the match and read out functions will depend on the number and distribution of partial and full responders through the chips and pages of PAM storage (the palette database is assumed to completely fill the Chameleon board, corresponding to roughly three to four thousand palettes). Table 1 lists the match and read-out times as a function of the number of responders for these two queries.

Although this is a fairly trivial example it should be noted that the ©palette's and the slot values within them can be subject to constant insertion, deletion and modification via simple read/write operations on the PAM's storage. Also the complexity of the query, in terms of variables used, has little affect on the match performance. Using the results above such a scheduling system would be able to sustain interleaved update and query rates of 40,000 expressions per second each.

#respndrs.	match time: query 1			match time: query 2			output time		
	min	avg	max	min	avg	max	min	avg	max
0	6.4	12.8	9.6	9.6	9.6	9.6	0	0	0
1	7.4	13.4	10.4	10.0	10.0	10.0	0.4	0.4	0.4
2	7.4	14.0	10.7	10.0	10.4	10.2	0.6	0.8	0.7
3	7.4	14.6	11.0	10.0	10.8	10.4	0.8	1.2	1.0
4	7.4	15.2	11.3	10.0	11.2	10.6	1.0	1.6	1.3
5	7.4	15.8	11.6	10.0	11.6	10.8	1.2	2.0	1.6

Table 1. Match and Output Times
(in microseconds)

7 Knowledge Bases and other Application Areas

The above example considered some application querying a dynamic database of facts or events. By adding rules to create a true knowledge base, the foundations are established for a hardware *blackboard system.* Blackboard systems fHayes-Roth, 1985] are popular mechanisms for supporting the intelligent control systems described in Section 1. The blackboard provides a central knowledge base shared, transparently, by a number of knowledge sources. It serves to establish the context for knowledge processing actions, provide a repository for hypotheses and control the problem solving process. Knowledge sources are scheduled based on events posted to the blackboard. All of these processes are associative in nature and commonly involve dynamic data.

Typically invocation of knowledge sources is an event-driven process, leading to the concept of associative *triggering.* Other examples can be found in production systems, and interrupt-driven behavior in robot control systems [Reece and Shafer, 1990J. Both require actions to be triggered on complex combinations of events. From the pattern matching system's point of view these triggers form the database against which external events are matched. The match is signalled by the absence of the 'miss' signal. So, for example, "©temperature >400" supports range checking on a temperature value or, referring back to the simple palette example, a trigger could be entered to await a palette becoming empty at a certain location. It is also possible to look for conjunctions of events posted in any order. The mechanism to support this uses the tag words described earlier. Every sub-clause of the conjunction ends in a tag denoting its state - 'satisfied' or 'not_ satisfied'. This symbol is modified appropriately after each event expression is broadcast. A short sequence of instructions can then be run to test for conditions that do not contain any 'not_satisfied' symbols. By implementing these triggers in the PAM new conditions can be entered at run-time, and existing ones modified or

deleted, all with no associated re-compilation. Full blown Rete-style systems can be implemented by replacing the 'satisfied' tag with an associative pointer to the corresponding beta node, where the join operation is also supported in the PAM (a scheme similar to Kogge *et al. [*1989]).

Other Chameleon applications include direct support for declarative languages such as Prolog, where pattern matching constitutes a large part of the fundamental execution mechanism [Kogge *et al,* 1989; Robinson, 1986J. Although compilers exist for such languages, the Chameleon system again provides the capability to handle dynamically created clauses.

Lastly there are fields such as memory-based reasoning LStanfill and Waltz, 1986] and genetic algorithms [Davis, 1987] in which systems attempt to reason or adapt themselves in the absence of rules. Such applications rely almost entirely on pattern matching and appear to be well suited to the capabilities of the Chameleon system.

It should be borne in mind when considering the capacity of Chameleon that, like a cache, it need only be of sufficient size to manage the transitory data in an application, not the entire database. Thus it could be viewed as filling the role of short term memory or as a support for a focus of attention. The end result is a system that allows complex applications to be run faster; or, as a corollary, real-time applications to be more complex.

8 Conclusions

In genera], hardware has been perceived as an expensive and unpromising direction - much hoped for functionality being usurped by new compilation techniques. However, as long as there are dynamic and unpredictable environments then associative hardware, such as the Chameleon system, will have a role in meeting system performance needs. The PAM architecture is also poised to take advantage of the trend towards application-specific memories. Yesterday's off-the-shelf memories are beginning to be replaced by libraries of off-the-shelf RAM blocks for semi-custom ASIC's.

This paper has described, and demonstrated the applications of, an associative co-processor which in prototype form contains 1Mb of storage. The subsystem plugs into the backplanes of Hewlett Packard 9000/300 or 400 series workstations and acts as a hardware accelerator for a variety of symbolic pattern matching functions. Currently a PCB version of the board is being constructed so that these prototypes can be evaluated in a number of applications. As an alternative, a simulator is being completed which will maintain a log file describing execution times given the actual hardware.

Lastly, it is interesting to consider the potential of such a system using the 1Mb chips proposed earlier. Coupled with denser (SIMM-style) packaging the same 8" x 11" Chameleon board could have a capacity of 16M bytes and contain 16k PE's. Such a system would have an aggregate processor bandwidth of 5 x 10^{12} bits per second (per board).

References

[Charniak *et al,* 1980] E. Chamiak, C. K. Riesbeck. and D. V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, 1980.

[Davis, 1987] L. Davis, editor. *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann, 1987.

[Forgy, 1982] C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence,* 19:17-37, 1982

[Hayes-Roth, 1985] B. Hayes-Roth. A Blackboard Architecture for Control. *Journal of Artificial Intelligence,* 26: 251-321, 1985.

[Highland and Iwaskiw, 1989] F. D. Highland and C. T. Iwaskiw. Knowledge Base Compilation. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence,* pages 227-232, August 1989.

[Kogge *et al.,* 1989] P. Kogge, J. Oldfield, M. Brule and C. Stormon. VLSI and Rule-Based Systems. In *VLSI for Artificial Intelligence,* pages 95-108, 1989. Kluwer Academic

[Reece and Shafer, 1990] D. A. Reece and S. Shafer. The Impact of Domain Dynamics on Intelligent Robot Design. Computer Science report CMU-CS-90-130, Carnegie Mellon University, May 1990.

[Robinson, 1986] I. Robinson. A Prolog Processor Based on a Pattern Matching Memory Device. In *Proceedings of the Third International. Conference on Logic Programming,* E. Shapiro editor, pages 172-179, 1986. Springer-Verlag.

[Stanfill and Waltz, 1986] C. Stanfill and D. Waltz. Toward Memory-Based Reasoning. *Communications of the ACM,* 29(12): 1213-1228, December 1986.

[Stormon, 1989] C. Stormon. The Coherent Processor - A Content Addressable Memory for AI and Databases. In *WesconW: Conference Record,* pages 240-244, November 1989.

[Wade and Sodini, 1989] J. P. Wade and C. G. Sodini. A Ternary Content Addressable Search Engine. *IEEF Journal of Solid-State Circuits,* 24(4):1003-1013, August 1989.

[Warren, 1977] D. H. D. Warren. Implementing Prolog. Technical Report 39, Edinburgh University, 1977.

[Yau and Fung, 1977] S. S. Yau and H. S. Fung. Associative Processor Architecture - A Survey. *Computing Surveys,* 9(I):3-28, March 1977.