PAM: Massive Parallelism in Support of Run-Time Intelligence

Ian N. Robinson
Hewlett Packard Laboratories

Introduction

The 'intelligence' of an application is often judged through its interaction with its environment. This could involve, for instance, the control of an automated factory floor or a dialogue with a user. Such interaction requires the ability to readily store, access and modify information acquired *during execution*. The data-structures that encode this information are typically expressed declaratively (as for example, facts, constraints, rules or frame-based objects), and constitute the application's 'knowledge-base'. They are essentially interpreted at run-time; their activation being based on pattern matching against some other structure encoding a current event, query or goal in the application. This access mechanism is central to the performance of the system.

Software mechanisms to support this access (commonly based on hashing) are complicated by these data structures often having an arbitrarily complex structure, as opposed to the sets of uniformly formatted data-types found in databases. Also, when dealing with knowledge-bases, very general access is typically required, which in turn requires that indexes be maintained on all fields of the stored data. Such indexing is complicated by the use of *wild-cards*, or *variables,* which allow generalizations or partial information to be stored.

Various compile-time techniques have been developed to handle such complexities (e.g. discrimination nets[1], RETE networks for OPS5[2] and WAM code for Prolog[3]), but they do not adapt well to run-time. The overhead of maintaining these complex, and highly interdependent, indexing structures in a dynamic system places a considerable burden on the application's performance. Also, for it to be interactive, there is often a real-time constraint on the applicability of the data being accessed. Information on how best to avoid colliding with another moving object is of little use to the system after impact.

Associative hardware[4], on the other hand, excels in applications where rapid access is required to data that is dynamic, unordered and unpredictable; witness the widespread use of associative hardware in cache memories and translation look-aside buffers (TLB's). This article describes an associative memory system specialized to support the syntax and associated pattern matching rules common to declarative expressions, hence the name *pattern-addressable memory*, or PAM. A co-processor board based on this hardware - a custom VLSI chip combining both logic and memory - is described. It is shown how this hardware supports a number of popular intelligent system architectures.

Expressions, Symbols and Pattern Matching.

Three declarative expressions are given below as they would be stored or input to the PAM. They are taken from a hypothetical application overseeing a robotized factory floor. The floor is divided into an X-Y grid of locations and the robots, each with a unique i.d., have the task of ferrying various parts about the factory. Expressions are made up of *symbols*, which typically comprise of constants, variables

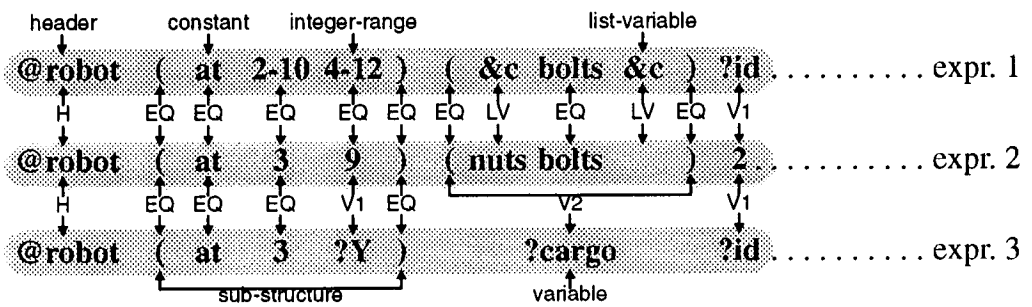

Figure 1. Three expressions illustrating symbol types and pattern matching rules

and the parentheses delimiting sub-structure. Headers are constants that indicate the start of a new expression. The PAM hardware also supports integer ranges and inequalities (that also allow a kind of fuzzy matching), and list-variables which allow arbitrary numbers of list elements to be matched (as in the cargo list of this example).

Thus expression 1 describes some robot within 4 squares of (6, 8) carrying, amongst possibly other things, bolts. Expression 2 describes a robot at location (3, 9) carrying nuts and bolts, and with an i.d. of 2. Expression 3 describes some robot on the x=3 axis.

The arrows between expressions illustrate the pattern matching rules (H, EQ, V1, V2 and LV) that lead to the three expressions pattern matching one-another. Headers only match headers with the same name (H). Constants and parentheses match their equivalents (EQ). A variable matches any other variable or constant (V1) or sub-structure (V2). A list-variable matches any number of constants, variables and/or parentheses, including none (LV).

The PAM stores symbols as 32-bit words, the first four bits of which denote the type.

Pattern-Addressable Memory

A number of associative memories designed to support matching on symbolic data exist[4, 5, 6]. Such hardware has typically been based on traditional content-addressable memory, with any additional logic being replicated for every word of storage. Pattern matching using the full syntax of declarative expressions is not well suited to this organization as so much of the process cannot be supported by comparators alone. Adding to the complexity of the additional logic, with the frequency of its replication, quickly gives rise to unworkably poor memory densities.

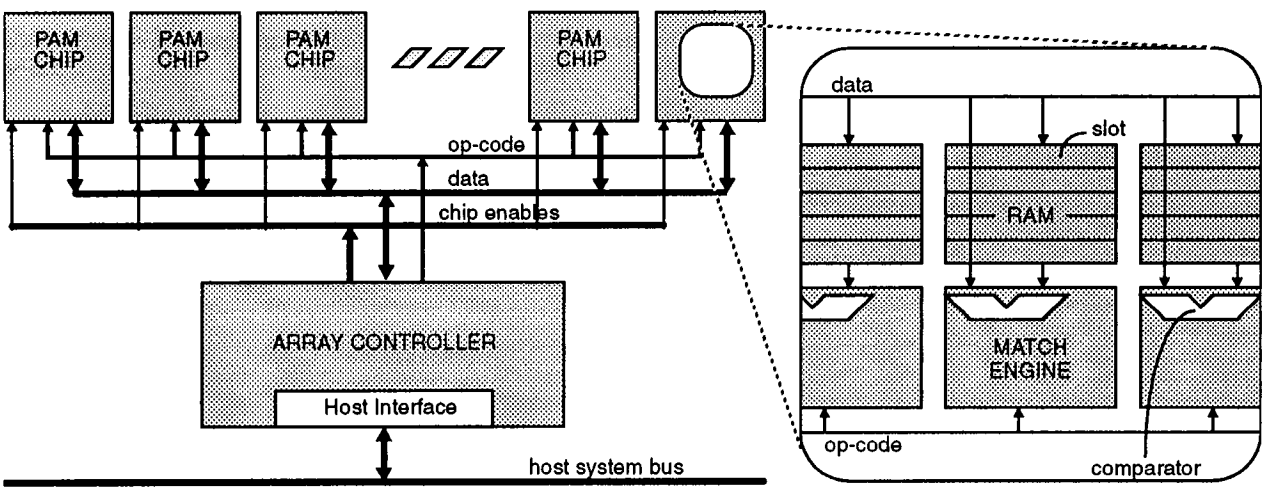

Figure 2. PAM Architecture

The PAM chip uses an organization in which the comparators and the match logic are multiplexed over small blocks of conventional RAM. This *match engine* with its attached memory forms the basic building block for the PAM chip (see Figure 2). The co-processor board, in turn, contains an array of these chips and an array controller. Operations, except for individual reads and writes, are performed in parallel over all the match engines in all the chips in the array. The board is designed so that a number of them, also, can be used together, attached to the same host.

Expressions are input to, stored, matched upon and output by the PAM as strings of symbols in an 'as written' order. The memory within each PAM chip is managed collectively as a single stack. Expressions written to the stack have their symbols stored in consecutive words, or *slots*.

In computing the pattern match of two expressions, it is the sequence of their individual symbol matches that is important. The progress of this match sequence is indicated by match 'flag's - a bit of state associated with each slot. The match input expression - or 'query' - is entered header first, and broadcast to all match engines. Thus the match sequence starts at the header and, in the case of match-

ing expressions, lasts through to the final symbol matched by the query. As each symbol is entered, every match engine acts in parallel to compare its slot's contents against the input symbol, and update its match flag accordingly. Expressions that mismatch will do so because somewhere through the sequence of symbol matches one fails.

To illustrate this Figure 3 shows an example based on a query over a set of expressions representing a simple parts-stock database (with tuples <part name>, <part number>, <quantity in stock>). The

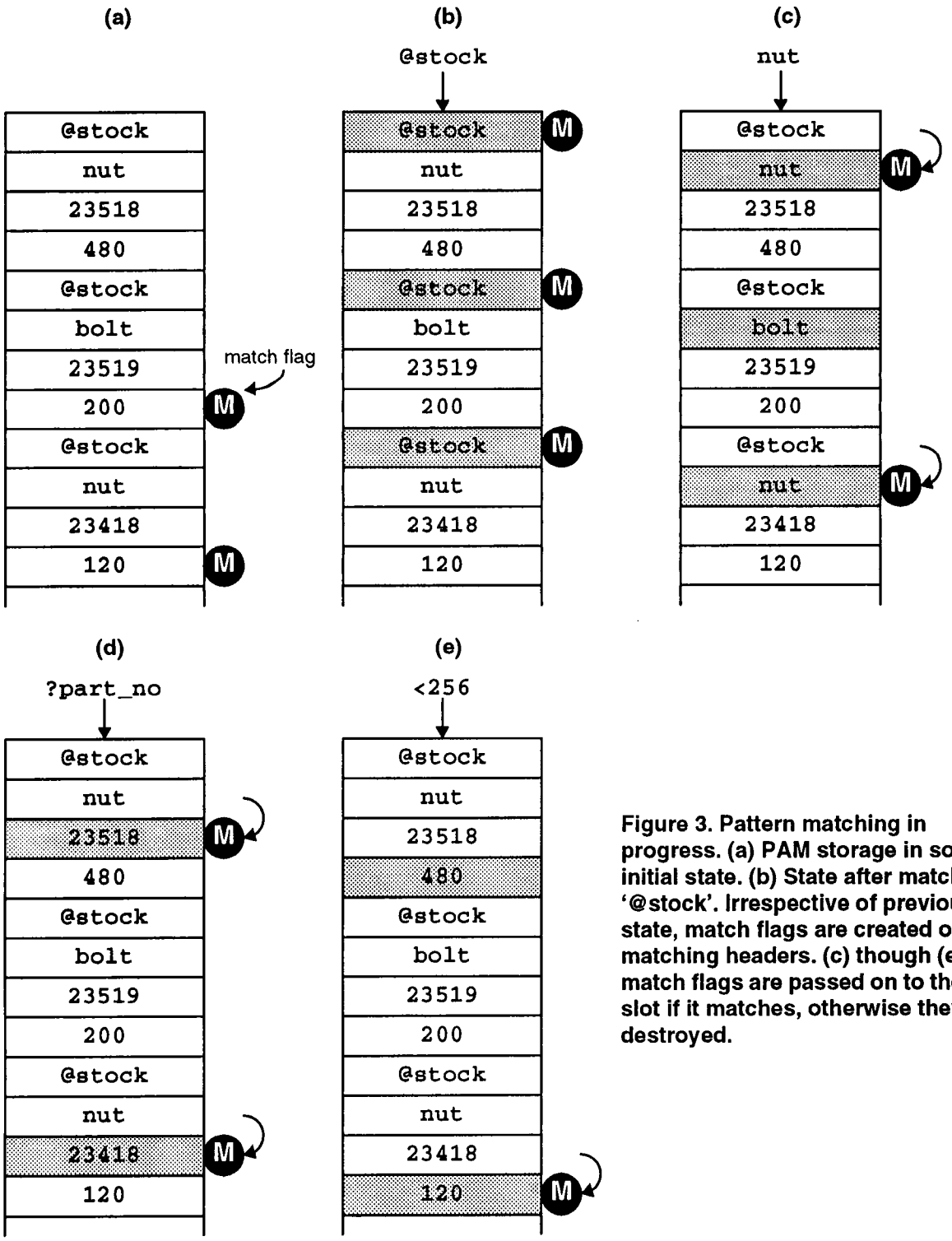

Figure 3. Pattern matching in progress. (a) PAM storage in some initial state. (b) State after match on '@stock'. Irrespective of previous state, match flags are created on matching headers. (c) though (e), match flags are passed on to the next slot if it matches, otherwise they are destroyed.

expressions are shown occupying the single stack format logically maintained by the PAM. Note how the match flags mark the progress of the match through the stored expressions. The result is that expressions that successfully pattern match with the match input - the *responders* - are indicated by match flags on the last symbol matched. This flag is responsible for selecting responders for subsequent operations. Apart from input and pattern matching, the hardware also supports the output of responders, their modification, their deletion, and the garbage collection of freed up slots. A 'match' signal - essentially a logical 'OR' of all the match flags - serves as an indication to the controller that responders exist.

The creation and movement of match flags gets rather more complicated when sub-structure and variables are matched - as in the 'robot' expressions of Figure 1 - but the overall concept is the same. For example, given a stored sub-structure and a variable as a match input, a match flag at the beginning of the sub-structure will be moved to its final parenthesis. Special circuitry takes care of performing such matches in one cycle.

Due to the multiplexing scheme what actually happens within the PAM chip is a little more complex. For each query symbol entered, the match engines scan their attached memories for matches, updating each symbol's match flag before moving on to the next. Figure 4 shows how, by sequential scanning of the memory blocks, the stack is effectively divided into *pages* (only five match engines and four pages are shown for clarity). The page select is common to all chips in the array and is driven by the array controller. Match times can be reduced for subsequent query symbols by not re-scanning pages on which no match flags exist. In the best case the page sequence will rapidly be cut down to the one page containing a responder. Matching then proceeds as if there were no multiplexing. Match times can be further reduced if *a priori* knowledge, as to where groups of expressions are located, can be used to restrict the number of pages matched on from the outset. These page control schemes help to offset the performance penalties of higher multiplexing ratios, making higher memory densities feasible.

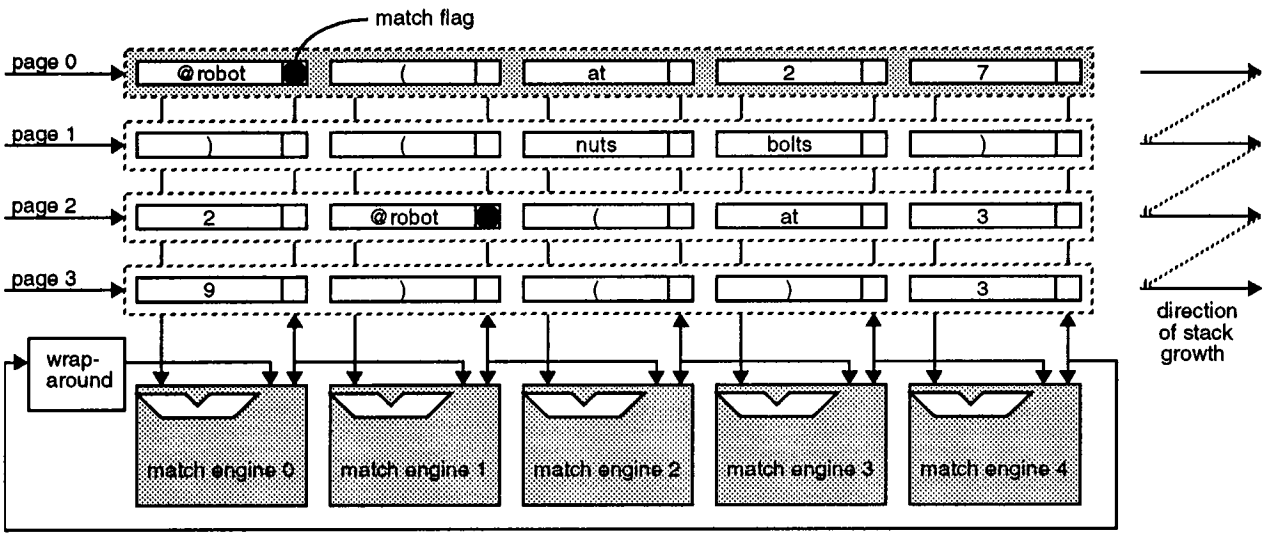

Figure 4. Pages and Match Engines (page 0 selected): flags are shown after a match on '@robot'. Pages 1 and 3 can be dropped as they contain no flags, pages are re-activated by the wrap-around logic that allows flags to move from one page to the next.

The wide world of pattern matching

By providing a flexible expression format and reasonably rich syntax a number of application styles can be supported. Figure 5 illustrates a number of these using a simplified, black-box view of the PAM.

An important feature that this figure illustrates is that, because the same syntax is available to both, the object of the pattern matching function can be either the stored or match input expressions. This is best illustrated in Figure 5, (a) and (b). In (a) - which corresponds to our previous parts-stock example - the matching expression(s) in the stored database are selected, the match signal indicating

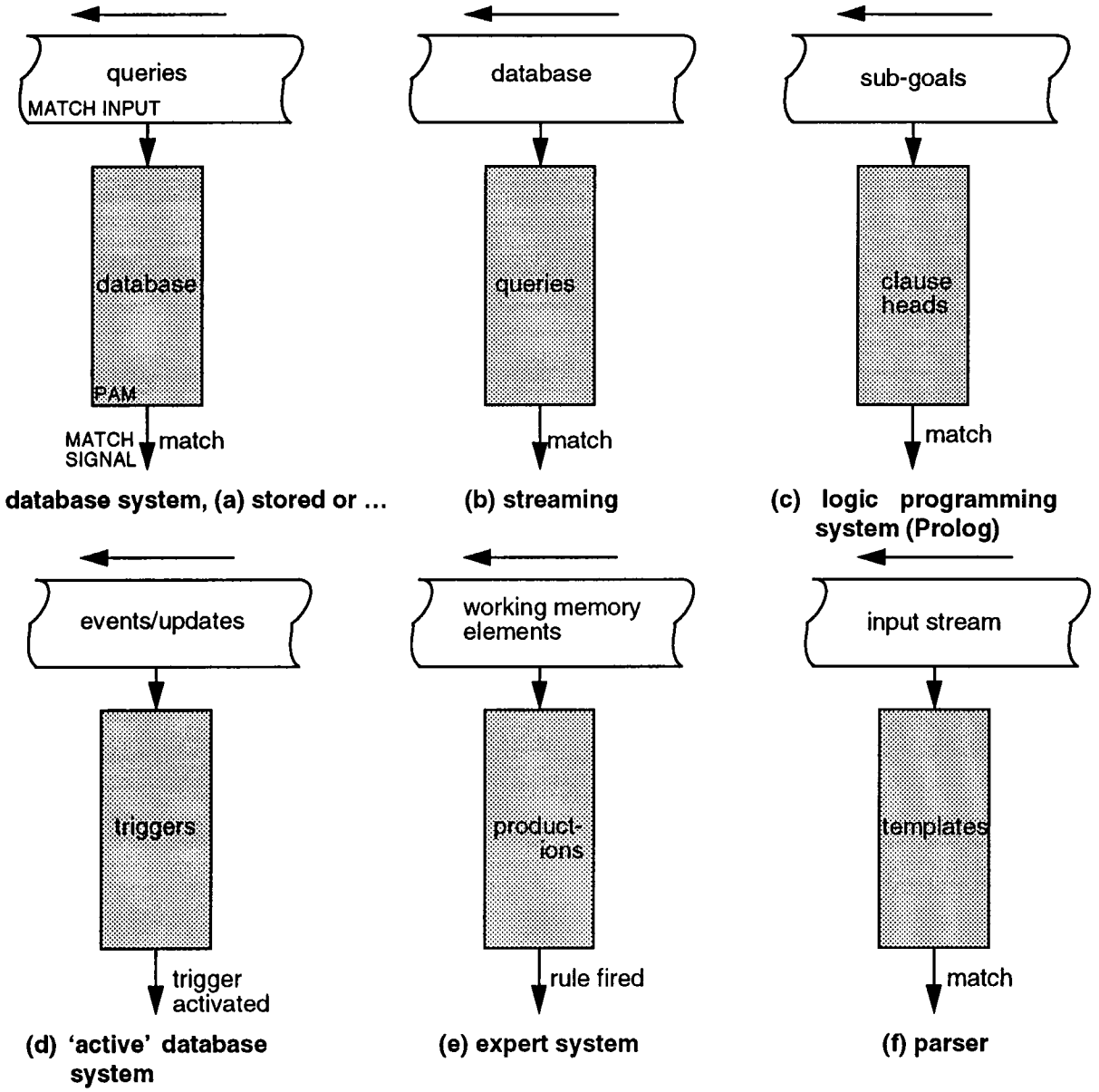

Figure 5. (a) - (e) Pattern matching applications

their presence. In (b) the database is 'streamed' by a stored batch of queries. When a match occurs the successful query is selected and the match signal allows the corresponding database expression to be captured.

The database style of operation naturally extends to backward-chaining systems such as Prolog (c), which was the inspiration for much of the syntax. In such systems the pattern matching between sub-goals and a knowledge-base of facts and rules constitutes a large part of the fundamental execution mechanism: unification.

In (d) the concept of the streaming database is extended so that the stored queries become 'triggers', activated by matching events input to the PAM. This can serve as the basis for an 'active' database as might be found, for example, in a blackboard architecture. Such a system will also support pattern-directed interrupts allowing for the kind of 'interrupt-driven behavior' popular in real-time control systems.

By including state variables in the stored triggers, updating them on successful matches and then

matching on combinations of these variables; triggers can in turn be extended to activate expressions based on particular *combinations* of events. This allows the PAM to support the firing of situation-action rules as found in production-system-style languages (e), the events triggered on becoming the 'working memory elements'. Similar rules, now in the form of templates, would allow the PAM to be configured as a parser (f). Particular constructions can be detected and flagged as the input token stream is scanned.

In addition there are fields such as memory-based reasoning and genetic algorithms 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 PAM.

In all these applications one of the big advantages of associative hardware is that it is fast. However complex the data-structures involved, pattern matching takes place on the fly as the match input is entered. Perhaps more important however is the fact that the computational complexity of the matching process is directly borne by the hardware, not through the use of software, e.g. complex indexing schemes. So, however dynamic the data-structures involved, with the PAM they are simply written to the memory and are thence immediately accessible via matching. They can be selected and deleted, and then they are gone; with no pointers to clear up or trees to balance. The PAM's main advantage over corresponding compile-time schemes is this ability to hold thousands of expressions and allow them to be continually created, modified and deleted, whilst still supporting rapid, and completely general, access.

The Current Prototype

The current version of the PAM chip is a small prototype - primarily orientated towards demonstrating functionality. Each chip contains 64 match engines and 32k bits of memory, all in an active area of 20mm^2 using a 1.2μm CMOS process. This represents a storage density more than 20 times that of the nearest comparable associative hardware[6]. The current version of the co-processor board is designed to plug into HP 9000 series 300 and 400 family workstations. It carries from one to sixteen PAM chips, giving a maximum per-board capacity of 512k bits. Up to sixteen boards can run concurrently off the same host. The software to interface to the board consists of a C library of PAM routines, like '`pam_match`' for example, and a header file describing the data formats used by these routines.

With regard to performance, consider the example of the robotized factory floor. In this system, the central database - consisting of expressions similar to expression 2 (Figure 1) - is continuously updated on each robots position and cargo. Meanwhile it is also queried with expressions such as 1 and 3 (e.g. 'find a robot on the x=3 axis'). Note that these queries use the more dynamic elements of the expressions.

With the various page control schemes involved, times for the match and read-out functions will depend on the number of pages searched, and the number and distribution of partial and full responders through the chips and pages of PAM storage. These results are tabulated below as averages of the best and worst case times for the various conditions. The system used has 16 PAM chips and runs with a cycle time of 200ns.

no. of responders	16 pages of '@robot's				4 pages of '@robot's			
	match expr. 1	match expr. 3	output	update	match expr. 1	match expr. 3	output	update
0	14.4	9.6	0		3.6	2.4	0	
1	15.1	10.2	0.6	16.5	4.3	3.0	0.6	5.7
2	15.3	10.5	1.2		4.5	3.3	1.2	
3	15.5	10.8	1.6		4.7	3.6	1.6	
4	15.7	11.1	2.0		4.9	3.9	2.0	

Table 1: Average match, output and update times (in microseconds)

Using four pages the status of 500 robots can be tracked. The figures above imply that such a system could support interleaved query and update rates of 85,000 expressions per second each. This

roughly corresponds to updating each robot's status every six milliseconds.

The architecture enjoys the bandwidth and scalability advantages inherent in its SIMD organization. The computational bandwidth on even the small prototype chip by itself is in excess of 1.3 gigabytes per second. The system can be scaled more or less arbitrarily: more memory to a chip (through larger die and tighter design rules), more chips to a board (through better packaging, e.g. SIMM's), and more boards to a system. Through these techniques a 1Mb capacity chip is quite feasible. Sixty-four such chips contained on sixteen SIMM's could populate one EISA board. Four such boards would give a PC or workstation a PAM capacity of 32 megabytes, and a match performance of 1.3 *terabytes* per second.

References

1. E. Charniak, C. K. Riesbeck, and D. V. McDermott, **Artificial Intelligence Programming**, Lawrence Erlbaum Assoc., 1980, pp. 121-176
2. C. L. Forgy, "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", **Artificial Intelligence**, vol. 19, 1982, pp. 17-37
3. Warren, D. H. D., "Implementing Prolog", tech. report 39, Edinburgh University, 1977.
4. S. S. Yau and H. S. Fung, "Associative Processor Architecture - A Survey", **Computing Surveys**, vol. 9, no. 1, March 1977, pp. 3-28.
5. P. Kogge, J. Oldfield, M. Brule and C. Stormon, "VLSI and Rule-Based Systems", **VLSI for Artificial Intelligence**, Kluwer Academic, 1989, pp. 95-108.
6. M. Hirata, H. Yamada, H. Nagai and K. Takahashi, "A Versatile Data String-Search VLSI", **Journal of Solid-State Circuits**, vol. SC-23, no. 2, April 1988, pp. 329-335.