In *Modeling the Innovation: Communications, Automation and Information Systems*, Carnavale, et.al. (eds), North-Holland, pp. 481-488, (*IFIP International Conference),* 1990.

Smart Memory: The Memory Processor Model

Tony R. Martinez

Computer Science Department, Brigham Young University
Provo, UT, 84602, USA

This paper overviews and proposes a class of smart memory devices called the *memory processor* model. Smart memory entails the tight coupling of memory and logic. The model seeks to alleviate the von Neumann bottleneck, take advantage of technology trends, improve overall system speed, and add encapsulation advantages. Speed is increased through locality of processing, communication savings, higher-level functionality, and parallelism. Parallelism is exploited at both the micro and macro levels. Data objects are accessed through descriptors, which give the memory a meta-knowledge concerning the objects, allowing for nontraditional access mechanisms. Both data types and operations are programmable. Innovative processing schemes, coupled with emerging technology densities, allow for substantial speed-up in traditional and novel memory operations. Three important paradigms introduced are *descriptor processing*, where operations are accomplished without access to the actual data, *associative descriptor processing*, supporting highly parallel access and processing, and the *single-program multiple-data* method, allowing parallelism by simultaneous processing of data objects distributed amongst multiple smart memories. Examples of specific operations are presented. This papers presents initial studies into the smart memory mechanism with the goal of describing its potential and stimulating further work.

1. Introduction

Recent advances and trends in technology allow new ways of thinking about the traditional boundaries between processing and memory. This paper proposes a model and presents initial ideas on techniques of coupling logic and memory with the goal of enhanced functionality and speedup in computing systems. Devices created by adding functionality to memory are referred to as *smart memories.* The architecture proposed in this paper is the *memory processor model* of smart memory. This model represents a large class of potential devices, and is referred to throughout this paper as simply smart memory (SM). The goal of this paper is to present and review initial studies on promising smart memory mechanisms, in order to stimulate thinking and further research in this potentially important area.

Although there are logic-enhanced memories currently coming on the market [4], as a rule these memories are very specialized and are streamlined to a single function. These include FIFOs, cache-tag RAMs, and video RAMs. In contrast, SM as proposed here, is general purpose in the sense that it can efficiently support a number of different data types and a variety of operations on those types. Both the types and the operations do not have to be defined when the SM is cast into hardware, but are programmable inside the SM to match the specific needs of a given system or application. Massively parallel SIMD processors have been considered by many as a type of smart memory. These include the MPP [2] and the connection machine [5]. These models are very large complete architectures, differing greatly from the architecturally small (chip-level) modularity grown mechanisms presented in this paper. Associative processors [10] support mechanisms similar to some presented in this paper, but again are large special purpose processing systems.

Throughout the paper, a storage item being manipulated is referred to as an *object,* and that which initiates the manipulation of the object as an *accessor*. When listing potential characteristics or functionalities of these smart memories, it is assumed that the example functionalities, although not exhaustive, are a superset of that which would actually be implemented in a given module. It must be decided which of the possible features return the highest overall performance in a given usage. Thus, the memories could be called *application specific* memories.

The basic notion of the smart memory (SM) scheme is the tight coupling of processing capability with storage capacity in the same locale. In terms of near term technology, this is seen as logic and memory combined on a single chip.

The proposed model of SM seeks to alleviate the von Neumann bottleneck, take advantage of technology trends, increase overall system performance, and provide encapsulation and modularity features. These four goals are briefly discussed below.

The distinct separation of processor and memory, together with the "fetch and execute" mechanism of von Neumann computing has proved inefficient for many types of computation. This *bottleneck* is exacerbated in a parallel processing environment. The SM model seeks to alleviate the bottleneck by allowing more processing to take place at the memory, and by creating an environment geared toward overcoming the psychological effect of the bottleneck [1] by creating new paradigms of processor/memory interaction.

Until recently, processing and memory devices were fabricated from two different technologies. With the onset of integrated circuit processing and memory elements, the two are now built from the same technology and are easier to integrate. The continually increasing density off on-chip circuitry forces us to examine our methods of utilization. It has been predicted [6] that there will be a 1000 fold increase in the density of circuitry on a chip in the next 15 years. To give a feel for this increased power, if half of a single chip were used for memory and the other half for logic, the chip could contain 1 Gigabit of memory and still have enough logic space available for the equivalent of 500 of today's microprocessors. Even more critical is the disparity between on-chip processing and off-chip communication. The above paper predicted that while on-chip density would increase by a factor of 1000 in the next 15 years, bandwidth off and on a chip would increase by only a factor of 64. Thus, this disparity will continue to worsen at a dramatic rate. The above observations lead to the conclusion that methods for taking advantage of on-chip circuitry, coupled with efforts to minimize off-chip communication are critical for future systems. SM is proposed as one method of attaining these goals.

A critical goal in any new architecture is increase of overall system performance and speed. The SM approach attempts to attain higher overall performance through four basic mechanisms. *Locality of processing* gains speed by doing as much computation as possible on-chip. The actual physical proximity of the memory and logic circuitry allow for higher speed processing than can be attained over larger distances. *Communications savings* occur by allowing, when favorable, operations to be sent to memory, rather than fetching extremely large amounts of data across a bus to a separate processor. The addition of both logic and meta-knowledge to a memory module, allows *higher level functions* to be carried out in response to memory requests rather than simply the traditional read/write capability. Hardware logic and mechanisms can be streamlined and optimized specifically for memory manipulating functions. Finally, *parallelism* can be used at the module and global levels to attain higher performance. The above mechanisms are overviewed in this paper.

Encapsulation and Modularity advantages enhance the usability of systems. Potential advantages of data abstraction, modularity, and object-orientedness are well documented and are not reproduced here [9]. Some of the advantages gained specifically for data objects include synchronization mechanisms, security and protection, typing information, automatic indexing, and others which are mentioned later.

SM is aimed at being a general purpose tool , which can be efficiently tuned for many computing environments. It is especially useful in environments requiring symbolic data which do not display high locality, stability, or contiguity. Originally these SM concepts where sought to overcome many of the memory processor interaction problems which arise in parallel systems [7].

The goal of this paper is to give an overview of the salient concepts of this smart memory model. More detailed discussion can be found elsewhere [7,8]. The paper first introduces the basic memory processor model, together with the notions of *descriptor* and *value* memories. An example architectural layout is then presented. Three techniques for enhanced computing are then overviewed. These are *descriptor processing, associative descriptor processing,* and the *single-program multiple-data* method. Finally, a sampling of the different potential functionalities is given.

2. A Smart Memory Proposal

2.1. The Memory Processor Model

The basic model is that of traditional memory technology coupled with a processor on the same chip as shown in figure 1.

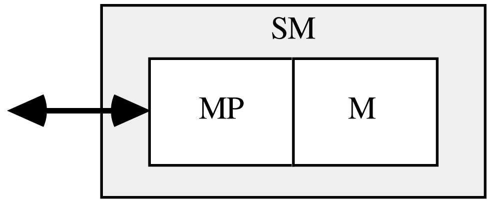


Figure 1 - Memory Processor Model

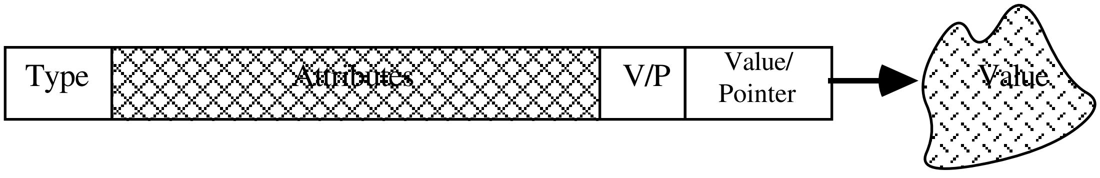
The processor in the SM model is known as a *memory processor (MP).* The MP differs from normal processors for two significant reasons. First, their goals are different. Consequently, the MP can be architected and streamlined specifically for memory manipulating operations. The second difference follows from having the MP and memory reside on the same chip. Because of this coupling, there are no constraints on the processor architecture arising from an external bus. These constraints include data path widths and speed. The MP can have much wider internal data paths functioning at higher speeds. These paths do not require the generality, or overhead, of a system bus (nor the associated pins). Registers can also be large enough to match the data paths, allowing a mechanism for fine grain parallelism.

The MP runs from a writable control store (WCS) or other programmable mechanism, allowing the SM the flexibility of changing functions and operations over time. If certain functions are found to be critical or very general, these could be placed directly in the hardware. This allows the SM to function as an abstract data-type machine. The way in which an operation on an object is implemented is invisible to the user.

The MP is connected to the outside world through a buffer, which receives and sends requests from/to accessors. Requests are assumed to be delivered and responded to in an asynchronous fashion, in the general case.

2.2. Descriptors

All data objects in the SM are accessed through a *descriptor.* A descriptor is a fixed length data word with a number of fields describing the data object. Minimally, a descriptor contains a type, a flag, and either the value or a pointer to the value as signified by the flag.



The descriptor also contains a number of fields, which may differ for different types, which are called *attributes*. These attributes could include size, statistics (max/min, etc.), synchronization locks, usage information, etc. The possible attributes and their uses are discussed in detail in the SM functionality section.

If a value is small enough to fit within the descriptor, then it is stored there. If not then it is stored in *value memory* and is pointed to from the descriptor. The descriptor gives the SM the meta-knowledge necessary to carry out higher-level functions. The fact that the basic descriptor is of fixed size does not rule out using a field(s) within the descriptor to point to an object containing more attribute information concerning the data object. Each data object has a unique name and is referenced through that name. To the SM it is irrelevant whether the accessor considers this as a name or an address.

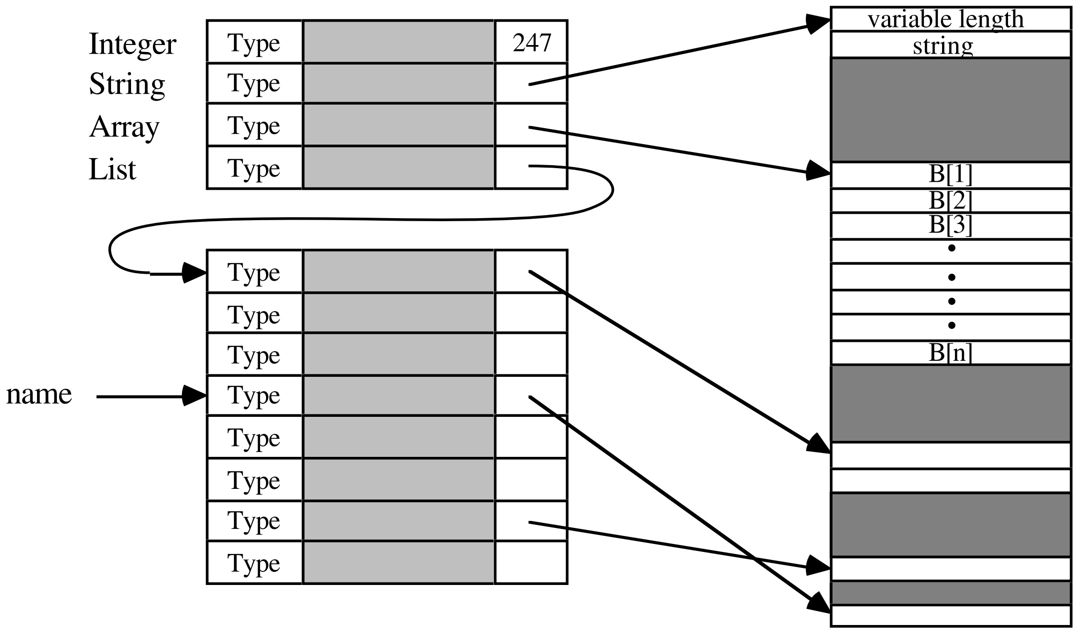


Figure 2 - Descriptor Pointer Examples

The pervasive effect of the fixed size descriptors is to bring the efficiency of regular structure to the now inefficient world of irregular and dynamic symbolic data. Typical fixed size data structures, such as an array of integers, are accessed efficiently through indexing since any cell of the array is a simple offset from the array's base. This follows from the fact that the array values are in contiguous memory locations and they are all the same size. For symbolic data, such as a list of variable length strings, this is not the case. One could not index into the list since each cell has a different size. Using descriptors, a cell in a symbolic list can be accessed by fast indexing into a descriptor vector and from there to the actual value through a pointer. Another advantage follows when the size of an object within the list changes. This typically means some reshuffling of memory, especially if the size of the new object is larger that before. However, the descriptor location and size need never change. Only the pointer (and possibly some attribute information) need change within the descriptor and accessing speed can remain high. Figure 2 shows a possible memory diagram for four different classes of data: fixed integer, variable-length string, array of fixed values, and a list of variable length values.

In the integer case, the value is contained within the descriptor. The string is simply pointed to and it resides in a variable amount of value memory. The array can have a single descriptor, and the values may reside in contiguous value memory, allowing efficient indexing. The list has multiple levels of descriptors depending on its nesting depth (in this case 2). The list could be accessed through the top level of the descriptor with an index, or a given descriptor within the list could be accessed directly with a name. A name could point into such a list at any level of the descriptor hierarchy. Descriptors of objects with dynamic structures could be connected by pointers, although the sub-objects are typically accessed directly with a name.

2.3. Value Memory

Values which are too large to fit within the descriptor are pointed to and stored in a *value memory*. Potential schemes for value memory are discussed elsewhere [7,8]. There are a small set of memory constructs in which a value can be stored. These include linked blocks of parameterized size for variable length data, and fixed arrays of parameterized size for storage of fixed-size and contiguous data. The value memory is completely abstracted from the user and can be implemented in an amorphous and efficient fashion.

2.4. Memory Processor Model Layout

This section presents a possible layout of SM for a chip. This is not meant to be a detailed implementation, but to give a flavor for the types of functional blocks which would be part of an implementation. For example, a displayed memory block does not exclude the possibility of a combined cache and slower memory as the physical incarnation of that block. Figure 3 shows an abstract block model of a memory processor model layout.

The model contains a buffer, possibly a separate I/O processing mechanism, which sends and receives access requests and replies to/from processing devices. These requests are read by the

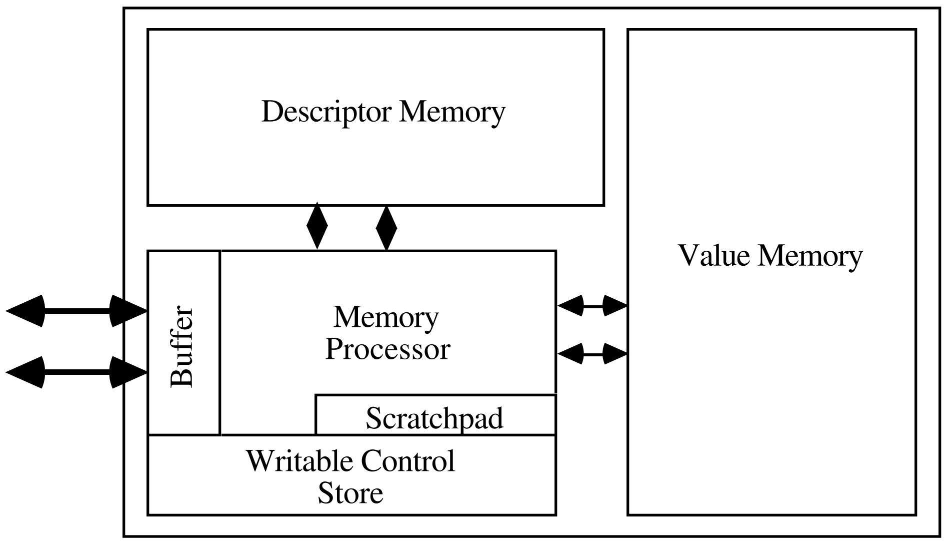


Figure 3 - Memory Processor Model Block Layout

processor as instructions to be executed. The request is decoded and executed by the memory processor (MP). The types of operations and their method of execution is programmable. The methods are stored in the writable control store section of the chip, which could be microprogrammable, using writable control store, (writable) programmable logic arrays, etc. The MP might also need some scratchpad RAM for operations and management functions (i.e. sorting, free lists, etc.). The value memory and descriptor memory are separate blocks within the SM chip, with separate data paths between them and the MP. Note that the paths between memories are not akin to the general purpose busses used between a traditional processor and any number of attached devices. These are uni-purpose *paths* with direct connections between MP registers and the memories, having potential for much higher widths and speeds than off-chip busses. The memory processor itself is composed of special purpose registers and a topology fashioned for efficient processing of descriptor and value memory objects.

3. Smart Memory Processing Mechanisms

This section discusses three processing techniques for memory operations which can be used as mechanisms for large classes of operations.

3.1 Descriptor Processing

Many operations on data can be accomplished without ever accessing the actual data values themselves. This occurs when the operation can use such information as data structure, access history, type, status, etc. These operations can be executed using only the descriptor of an object without actual access to the object. This method of processing is called *descriptor processing.*

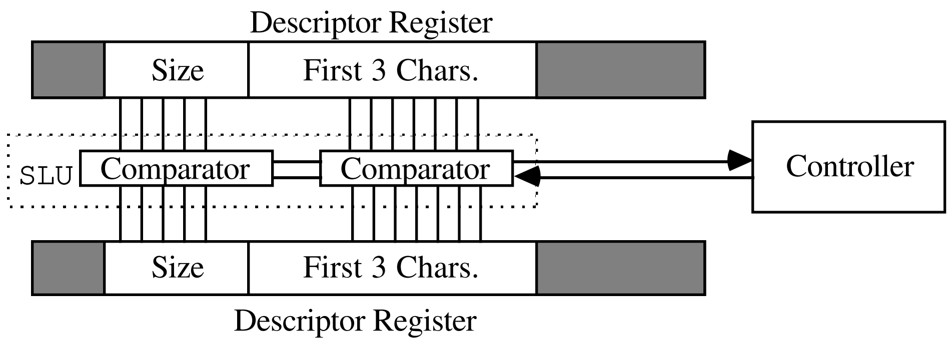


Figure 4 - Descriptor Processing

An example of this could be the *compare-string(a,b)* operation. A comparison of the size field is sufficient in most cases to test inequality. Also, it might be reasonable to keep the first three characters of the string in the descriptor. By comparing the size and these three characters with those in another descriptor, the large majority of all compare operations could be accomplished without reference to the actual values. In the case where the first characters of the string are equal, initial blocks containing each string would be fetched and compared. Figure 4 shows an example where two descriptors, loaded into descriptor registers, are directly compared.

Descriptor processing not only has the potential to speedup traditional operations by avoiding needless access, but due to the meta-knowledge existing in the descriptors, makes available data operations not considered with normal memory systems. For example, an operation might ask if a data value has been modified within a given period. More examples of descriptor processing are detailed in the SM functionality section.

3.2. Associative Descriptor Processing

The use of Content-Addressable Memory (CAM) has up to now been limited due to expense (it requires a larger cell size), coupled with the availability of alternate methods to perform associative lookup (hashing). When associative mechanisms are used solely for accessing an object by a name (or field), then the hashing mechanism is the most cost-effective method. If CAM can be used to its full potential as a parallel processing mechanism, then it can be well worth its implementation expense. CAM memory cells can currently be implemented with 5-10 transistors, compared to ~5 transistors for SRAM and 1 transistor for DRAM. CAM requires regular data structures, where data or fields can be aligned along a logical column. The MP model has this characteristic in the descriptor memory. A recurring theme in the descriptor mechanism has been to allow regularity (fixed size and field semantics) and its concomitant speed potential, to be used in an irregular and dynamic environment (symbolic data in the value memory).

The basic notion is to make all or part of the descriptor memory a CAM, and then use associative processing methods [10] for extended memory operations. The descriptor memory would still have potential for normal access when needed. By making the descriptor memory CAM-based, the fields of descriptors can be used to implement complex accesses in a parallel fashion. The memory has a dynamic tag vector which can set and mask its bits under direction of the controller. One can then request an access such as "*Retrieve* all *Rule-Objects* with three fields," where *Rule-Object* is some user-defined type. Figure 5 shows a possible model for this computation.

The type field is set to the bit pattern for *Rule-Object* and the *Num-Fields* field is set to 3. The other descriptor fields are set as *don't care (x)*. All descriptors fulfilling both requirements set their corresponding bit in the flag vector. The values can then be retrieved in sequence through the pointers to value memory in a direct fashion. Note that *retrieve* in the above command could be replaced by *delete*, *increment*, etc.

Many operations can benefit by using associative descriptor processing in a heuristic fashion in order to find the best possible candidates for retrieval, even though access and testing of the values will still be necessary. This type of associative descriptor processing is called *descriptor pruning*. For example, assume the accessor wants to retrieve one or more objects in the SM which *unify* with a specific object (operand object). Rather than begin a unification algorithm against each possible candidate in sequence, the descriptor CAM can be used to prune away impossible candidates. The descriptor could have a field containing the number of terms in the data object. The number of terms in the operand object would be put in the tag vector, after which the flag vector would contain only bits for those data objects having the same number of terms as the operand object. Other fields could simultaneously be tested (type, depth, etc.) such that a large percentage of nonunifiable objects are pruned in one parallel step.

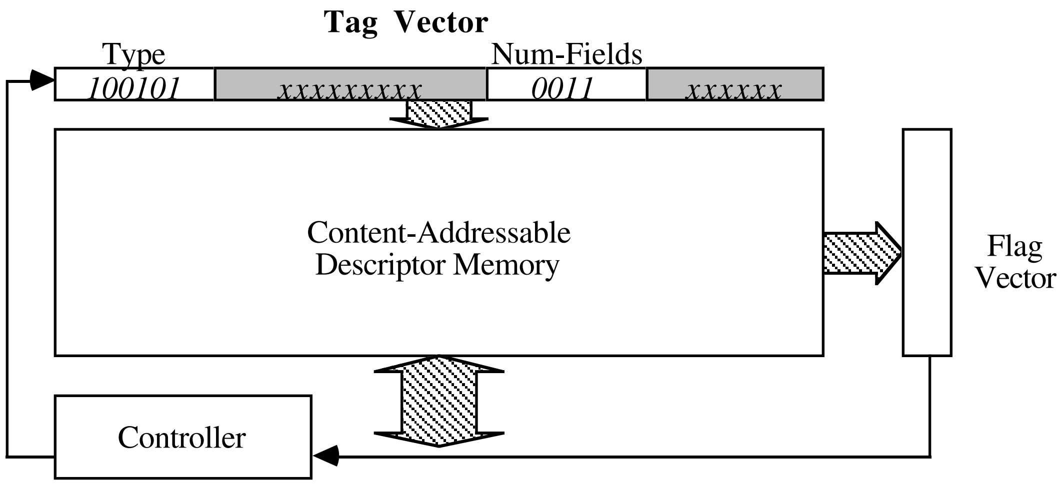


Figure 5 - Associative Descriptor Processing

ADP, and other SM mechanisms, can be used for other purposes than traditional and extended memory operations. For example, by considering descriptors as abstract channels and processors, ADP can be used to accomplish complex communication events in multiprocessor systems, as proposed in [7].

3.3 Single-Program Multiple Data Method

Another important mechanism is called *single program multiple data (SPMD)* processing, which extends traditional SIMD processing. *Program* is used instead of the traditional *instruction* since there is no notion of lock-step execution of a broadcast opcode. Rather, a set of SM's receive an operation which they apply to their data simultaneously, though asynchronously (the method used to execute the operation can also differ between modules since that aspect is encapsulated). Multiple data objects can be distributed across a set of SM's in a balanced (each SM having approximately an equal portion of the data structure) fashion such that processing can occur in an efficient parallel fashion. Each SM may contain portions of many distributed data objects. This balanced distribution is referred to as *symbolic interleaving*. This interleaving is parameterizable with one object datum per SM and a complete object on one SM as the two extremes. The most efficient distribution depends on the data and operations required, and can be set accordingly. Similar methods have been shown to be very efficient, as shown by applications on the *connection machine* [5]. For operations which require data to be gathered and possibly processed during gathering (i.e. *retrieve the 10 largest values in a distributed object)*, a hierarchical topology of SM would allow logarithmic time for gathering.

4. Smart Memory Functionality

In this section, the basic classes of SM functionality are overviewed, with more detailed discussion and other potential functionality described elsewhere [7,8]. The functionalities are broken out into four broad categories: extended traditional memory manipulation, memory-intensive operations, services, and higher level functionality. This section seeks to give a sampling of potential utilities in order to give a flavor for potential application. A single SM device would only support a small subset of the possible functionalities, those for which it is specifically programmed.

The functionalities discussed below follow from the features of the architecture of the memory processor model of SM as discussed in the previous section. These features include descriptors, streamlined processing, descriptor processing (DP), associative descriptor processing (ADP), and the single-program multiple-data method (SPMD).

4.1. Extended Traditional Memory Manipulation

Traditional symbolic data types which could require support from SM include sets, lists, stacks, arrays, structures, environments, text (strings), graphs, frames, semantic nets, and discriminant nets [3]. Common operations on objects, such as insert, car/cdr, etc., could be implemented directly as primitives of the SM.

Operations dealing with the structure of an object (such as *size-of*, *number-of-dimensions*, etc.) can be done strictly through descriptor processing (DP). The *compare* operation is one that is done through a combination of DP and value comparison.

Many operations can benefit from a combination of DP, ADP, and SPMD methodology. The *delete* operation need only involve the descriptor (of course, the value could be garbage collected later), and could be done through associative descriptor memory, and across a set of SM's. For example, *delete*(all *type1* and *type3* objects older than 3 days) could be multicast to a set of SM's, where each SM would do parallel deletion both internally and at the SM level. A large set of database style queries can efficiently use the combination of DP, ADP, and SPMD.

Other operations benefit from SPMD, while still necessitating access and manipulation of their values. Set operations (*intersect, union, member*, etc.) have been shown to fit this scheme by applications of the *connection machine* [5]. *Replace-text(string1, string2)* could do parallel replacement of all occurrences of one string in a text object with another, by having the text object symbolically interleaved amongst SM's.

An important class of operations in a distributed communication environment (such as rings, hypercubes, or banyans, where bandwidth can remain high, but latency is also higher) is object-copy and object-move. Rather than access a data object a word at a time, simply ask for the entire object or a subset thereof. This not only takes advantage of communication technology, but is especially useful for noncontiguous symbolic objects. Rather than access a distributed dynamic graph by iteratively accessing a node, and then accessing through its pointers, the graph can be requested as a single object, allowing the SM to locally handle the overhead required to assemble the graph. These operations could also be used by the system for load balancing, swapping, etc.

Operations dealing with entire data objects (reverse, transpose, etc.) can optionally be accomplished by making no actual change to the data, but by setting a *structure-tag* within the descriptor. A reversed list would have a tag set in the descriptor such that an access request would cause the list to be read out starting from the end. This notion could also allow multiple-views of the same data from different accessors by having different structure-tags set for given sets of accessors.

4.2. Memory Intensive Operations

There are a class of operations which require one or a few operands to be manipulated against a large set of data. An example of this is the pattern-match of an object against all objects in a large set. The pattern is compared to each object, and depending on the semantics of the operation, some value is returned. This type of operation is very cache-unfriendly since a comparison happens only once with each piece of data. If this were done at a processor, all of the data would have to be fetched across the communication network, only to be discarded (or worse yet, all written back) after one comparison of the operand with each datum. This is a case where the program should be taken to the data, and not the data to the program. The memory-intensive operations use a combination of DP, ADP, and SPMD, for highest parallelism and efficiency.

Pattern-Matching is a basic operation in symbolic and AI applications. An important class of this operation is best-fit pattern matching where there are no exact copies of the operand, but one wants to locate the closest or most similar object or objects. In current AI applications, these objects could be frames, rules, images, etc. The method of operation for this and other memory-intensive operations starts by having the set of objects to be compared against (knowledge base, dictionary, etc.) symbolically interleaved across a set of SM's. The pattern is then multicast to that set of SM's. Descriptor pruning is used to heuristically find the best possible candidates for matching. This parallel operation uses the associative descriptor memory and for this case would compare such operand features as size, number of fields, type, etc. with the descriptors. After one parallel ADP operation only the best candidates would remain for actual accessing and comparison of the values with the values in the operand. The best fit(s) from the different SM's would then, depending on the semantics of the operation, either be sent to the accessor, deleted, modified, etc.

Unification is another common operation able to benefit from the above methodology of SPMD and descriptor pruning, as mentioned in section 4.2 Number and types of fields are critical comparisons to test potential of unification during descriptor-pruning. Other memory intensive operations suitable for SM include statistical analysis of data, truth maintenance, and logical deduction.

4.3. Services

In this section, services provided by the SM, which are not purely memory access mechanisms, are discussed. These services aid in the overall memory functionality, efficiency, and usability.

The *type* of each object is kept in a field in its descriptor. This type not only aids in many accessing mechanisms, but also allows such functionalities. as run-time type-checking and operand coercion.

Descriptors allow a natural protection and capability mechanism. Data can be given ownership and access rights which protect it from overt and accidental compromise. The advantages of a capability-based system can be attained through these mechanisms, including the ability to pass rights to others.

Debugging tools can use a field in the descriptor to aid in the process of debugging programs. Such items as breakpoints, usage counts, time-stamping, etc., can be optionally part of the processing mechanisms.

The SM can also be an aid to systems communication. SM can support migration mechanisms by keeping forwarding and other routing addresses for dynamic objects within the descriptor. A potentially useful function is the ability of the SM to initiate a message (interrupt) to a processor, when it has finished a large-grain operation, as an alarm mechanism, or as a multicast mechanism to communicate modification of a shared object.

4.4. Higher Level Functionality

With increasing availability of logic capacity, higher level functions could eventually be supported through similar mechanisms. These operations include non-standard retrieval and filtering, result caches (where recent results of process intensive operations are saved for direct reference), mapped on-conditions, virtual data [3] (where a stored object is actually a computational process), and low level textual search primitives (such as wildcard, etc.).

5. Conclusion

This paper has overviewed the memory processor model of smart memory. The model seeks to alleviate the von Neumann bottleneck, take advantage of technology trends, improve overall system speed, and add encapsulation advantages. Speed is increased through locality of processing, communication savings, higher-level functionality, and parallelism. The techniques of descriptor processing, associative descriptor processing, and single-program multiple data mechanisms allow substantial small-grain parallelism for traditional and innovative memory operations. Current research goals of the memory processor model include implementation studies and VLSI prototypes, fault tolerance, and support of further novel mechanisms of memory/processor interaction.

6. Bibliography

[1] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, pp. 613-641, (August 1978).

[2] Batcher, K. E., "Design of a Massively Parallel Processor," *IEEE Transactions of Computers,* Vol. C-29, 9, pp.836-840, (1980).

[3] Charniak, E., C.K. Riesbeck, and D.V. McDermott, Artificial Intelligence Programming, Lawrence Erlbaum Associates, New Jersey (1980).

[4] Cole, B.C., "Smart Memories are Eating into the Jelly-Bean Market," Electronics, pp. 65-69, (Feb. 5, 1987).

[5] Hillis, W.D., The Connection Machine, MIT Press, Cambridge, Mass., (1985).

[6] Lundstrom, S. F., and R. L. Larsen, "Computer and Information Technology in the year 2000 - a Projection," Computer, 18(9), pp. 68-79, (Sept. 1985).

[7] Martinez, T. R., "Smart Memories: Potential Functionality and a Proposed Architecture," *MCC Technical Report PP-178-87*, (June 1987).

[8] Martinez, T. R., "Smart Memories Architecture and Methods" Submitted.

[9] Myers, G. J., Composite/Structured Design, New York: Van Norstrand Reinhold, (1978).

[10] Yau, S.S, and H.S. Fung, "Associative Processor Architecture - A Survey," *Computing Surveys*, Vol. 9, No. 1, (1977).