In *Future Generation Computing Systems*, vol. **6**, No. 1, pp.26-58, 1990.

Smart Memory Architecture and Methods

Tony R. Martinez

Computer Science Department, Brigham Young University

Abstract

This paper discusses potential functionalities of smart memories. Smart memory entails the tight coupling of memory and logic. A specific architecture called the *memory processor* model is proposed. 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. 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, and the model is streamlined for memory operations and services. Innovative processing schemes, coupled with emerging technology densities, allow for substantial fine-grain parallelism 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 paper 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 presents some initial ideas concerning methods of coupling logic and memory with the goal of enhanced functionality and speedup in computing systems. These models, created by adding functionality to memory, are referred to as *Smart Memories*. 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.

A brief synopsis is first given of what is meant by a smart memory in this paper. This is followed by a section discussing the pros and cons of such a model. Section 4 describes a more detailed model of a potential smart memory device, the memory processor model. The following section presents the processing techniques which support a large class of memory operations in a highly efficient manner. The smart memory functionality section lists a number of possible functionalities and how they could be implemented on the memory processor model.

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 smart memories, it is assumed that these functionalities, though not exhaustive, are a superset of that which would actually be implemented in a specific device. In some cases, support of one operation may preculde efficient implementation of others. It must be decided which of the possible features return the highest overall performance for a given application.

2. Smart Memory Overview

The most basic notion in the smart memory (SM) scheme is that of coupling 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.

A second important aspect deriving from the on-chip coupling of both processing and memory is that of encapsulation. Encapsulation allows the complete abstraction of any object accessed within the SM to any outside accessor. Any accessor of data need not know the actual way in which a given object is stored. Smart memory operations are also abstracted and the SM concept takes on the overall flavor of the object-oriented or abstract data type paradigms.

An SM allows higher-level functionality on its data objects due to a meta-knowledge of those objects. The SM can have a range of knowledge about both the structure and the semantic content of the stored objects. Current memories know nothing about what kinds of objects are stored within their array of bits, nor anything about their semantics, although some tagged memory architectures do begin to approach these notions. The model proposed in this paper implements the meta-knowledge by storing information about each object in a structure called a *descriptor*.

There are logic-enhanced memories which are currently coming on the market [4]. However, as a rule, these memories are very specialized and are streamlined to a single function. These include FIFO queues, 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. Neither the types and the operations need be defined when an SM is cast into hardware, but are programmable 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 [10]. These models are very large complete architectures, differing greatly from the architecturally small (chip-level) modularly grown mechanisms presented in this paper. Associative processors [19] support mechanisms similar to some presented in this paper, but again are large special purpose processing systems.

There are a number of possible system topologies into which SM modules could integrate. All memory could be stored in SM modules. Alternatively, each processor might also have its own traditional local memories to be used for high speed access not requiring further functionality (i.e. local read-only data and instructions). In short, since SM will by its nature require more logic than standard RAM, SM should only be used to store and operate on data where substantial system performance improvement can occur.

The SM access mechanism is more like asynchronous message passing than normal read-write access (i.e. retrieve the 10 largest values from object X, rather than fetch location 1000). Figure 1 displays some alternate structures where processor (P), traditional memory (M), and smart memory (SM) interact. The smart memory systems presented here were originally targeted for symbolic data in multiprocessor systems [13,14]. In those cases SM modules could be shared and insterspersed within a shared memory or message passing MIMD system. Parallel processing aids (i.e., montiors, synchronization, communication, etc.) can then also be a support option of the SM. In this paper, we discuss SM for general computing environments.

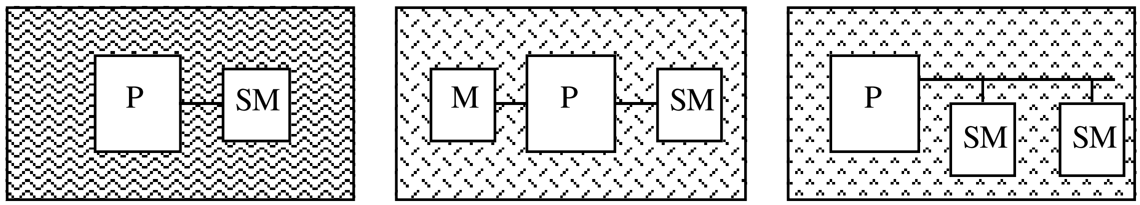


Figure 1 - Alternate Processing Nodes

3. Smart Memory Pros and Cons

3.1. Why a Smart Memory

The reasons for proposing smart memories are divided into four broad classes, each discussed in the following four subsections. Specific examples of these arguments are given in section 6 on SM functionality. These categories are the von Neumann bottleneck, technology trends, overall performance, and encapsulation/modularity.

3.1.1 The Von Neumann Bottleneck

The distinct separation between processing elements and memory has been a point of criticism, although debated by some, concerning the traditional von Neumann architecture. Instructions and data are passed between memory elements and a processor by means of a bus, which has been labeled the von Neumann bottleneck. This current state scheme leads to a sequential fetch and execute paradigm. In contrast to this scheme is the connectionist approach where memory and processing capability are tied into one and the same mechanism; i.e. processing occurs as data flows through nodes and node connections, and the network memory is defined by the current state of the nodes (functions) and connections (weights). The SM approach does not claim to be connectionist, however it is meant to guide processing/memory innovation in that direction by both mitigating somewhat the bottleneck, and creating an environment geared towards new paradigms of processor/memory interaction. In a multiprocessor environment, the bottleneck is exacerbated due to memory contention, distance between nodes, and synchronization.

Perhaps the most pervasive effect of this bottleneck has been psychological [1]. The memory-fetch and process paradigm has colored not only languages and architectures, but our concepts of computation and problem solving in general.

3.1.2 Technology Trends

One factor causing the separation of logic (processing) and memory is that in early computing systems they were built from two different technologies. With the onset of integrated circuit processing and memory elements, the two are now built from the same technology and it follows that their integration should be much easier.

The increasing density of on-chip circuitry allows freedom to place increased functionality on a single chip. It has been predicted [12] 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. Current sequential paradigms do not efficiently scale to this amount of density. Smart memory is one mechanism seeking to take advantage of the tremendous increase in logic density to obtain substantial speed-up in future systems.

Even more critical is the disparity between on-chip processing and off-chip communication. In today's VLSI technology the high cost of off-chip communication (in terms of pin-out, speed, and power) is already becoming a bottleneck. The article [12] also 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.

3.1.3 Overall Performance (Speed)

The main motivation in beginning this research effort, as in most architectural studies, is to improve the overall system performance. This typically centers around speed. The SM approach attempts to attain higher overall performance through four basic mechanisms: locality of processing, communication savings, higher-level functionality, and parallelism.

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. Rather than fetching large amounts of data across interconnect, processing on that data, and writing it back to memory, an operation can be sent to the memory with the processing taking place at the chip, and value(s) or acknowledgement being returned. This overlapping of latency with processing is similar to state-multiplexing as it takes place in the HEP computer [17] and in circulating data flow machines [6].

Since more work is done on data resident on the SM chips, much less communication need take place over bus or other interconnect topologies between processors and memories. This *communication* bandwidth *savings* both increases speed and makes available more of the expensive communication bandwidth for other purposes. Also, larger grain movement of data, such as the movement of a complete structure rather than a word, takes advantage of communication technologies ability to deliver better bandwidth than latency.

By adding both logic and some meta-knowledge to the memory module, *higher level functions* can be carried out in response to memory operations. A memory operation can be more powerful than the traditional read and write. For example, rather then simply fetch a 32 bit word, a low-level read operation could be *fetch-graph*, which could send an entire graph structure (which most probably would not reside in contiguous memory) to a requestor. Hardware logic can be streamlined and optimized specifically for memory manipulating functions. A number of examples of speed-up through functionality are presented in the SM functionality section.

Parallelism can be exploited in the SM scheme both on-chip and at higher levels. Compared to off-chip communication, much wider data paths can be used on-chip.

This leads to wide on-chip processing registers where separate logic can simultaneously process different slices of the data. Parallelism can take place within the processing mechanism of the smart memory, and through the simultaneous use of multiple smart memories on a single problem. These techniques are discussed below.

3.1.4 Encapsulation and Modularity

The advantages gained by the encapsulation and modularity which the SM model enforces upon its data objects are meant to aid in system usability. These types of improvements have previously been sought through data-abstraction, object-oriented programming, structured design, and other paradigms. The potential advantages of these schemes have been well documented and are not reproduced here [15]. Some of the advantages gained specifically for data objects include storage and operation abstraction, synchronization mechanisms, security and protection, typing information, etc. These and other benefits are detailed in the SM functionality section.

3.2. Why not a Smart Memory

Three basic items need to be kept in mind while devising SM innovations, since any one of them could mitigate the effectiveness of SM. They are cost, indirection, and utility.

By adding functionality where it was not before, more cost is inevitably accrued. This cost is predominantly incurred in additional circuitry. However, in this case, technology is on our side, as the cost in circuitry continues down concurrent with the aforementioned rise in density. If real improvement in system performance can be attained, this cost will be well justified.

A more serious problem is to what extent the speed gains through locality, minimized communication, functionality, and parallelism may be offset by indirection. SM makes it's improvements by adding logic between the physical storage and the processor. Hence, the access mechanism, though more functional, is also more complex (i.e. going through a descriptor rather than directly to a data word). A very simple stripped mechanism (read/write) is faster, at that simple level, than a more complex mechanism. SM seeks to simplify and minimize this indirection. Many recent architectures have failed from inefficiency due to excessive complexity and indirection. The SM model here proposed overcomes this issue by allowing dramatic speed increases for common (tuneable) high-level operations, while maintaining ability for comparable speed when using traditional access mechanisms.

A final concern is that of utility. No amount of innovative mechanism is worthwhile unless it is actually utilized sufficiently to warrant its cost. It is felt that many useful operations and methods, which provide improved system performance, can be supplied through SM. However, some will require new ways of thinking from systems users. As different types of functionality are considered out of the many possibilities which could be included in SM designs, the most pressing issue will be the extent to which the function would/could be used in current and future applications.

4. A Smart Memory Proposal

4.1. The Memory Processor Model

This section presents the *memory processor* model of smart memory. The discussion is not meant to be at an implementation level, rather that of the required functional building blocks. The basic model is that of traditional memory technology coupled with a processor on the same chip as shown in figure 2.

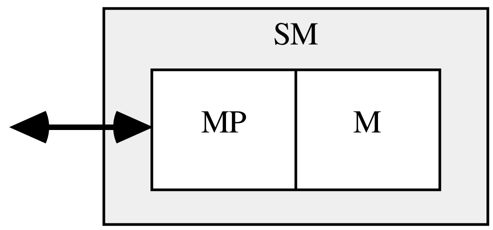


Figure 2 - Memory Processor Model

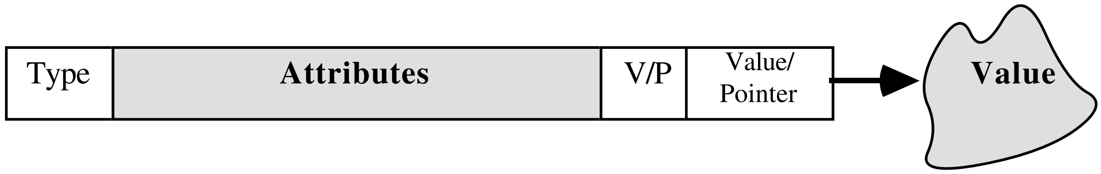
The processor in the SM model is known as a *Memory Processor* (MP). The MP differs from normal processors for two significant reasons. First, their goals are different. Consequently, the MP can be architected and streamlined specifically for memory manipulating operations. The second difference follows 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 low level 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. Thus, the method of operation and/or storage of an object can be changed, (or an SM replaced by a newer model), without affecting the user (except for possible changes in operation speed).

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.

4.2. Descriptors

All data objects in the SM are accessed through a *descriptor*. A descriptor is a fixed length data word with a number of attribute fields describing the data object. Minimally, a descriptor contains a type, a flag, and either the value (the actual data object) 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. Since the storage structure and descriptor contents are not observable from the outside, the terms descriptor and data object can be used interchangeably from the viewpoint of an accessor. The address of the descriptor is the only *hook* to the object. 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. Either associative or addressed access may be used.

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. (These are standards mechanism in such list processing languages as Lisp). Figure 3 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.

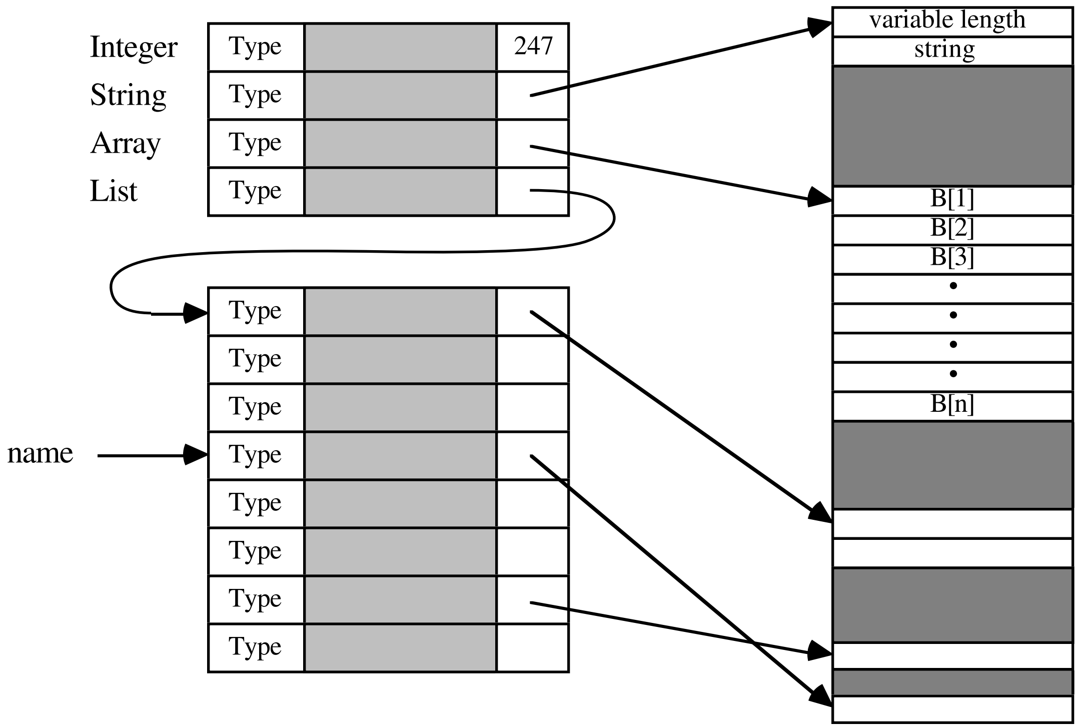
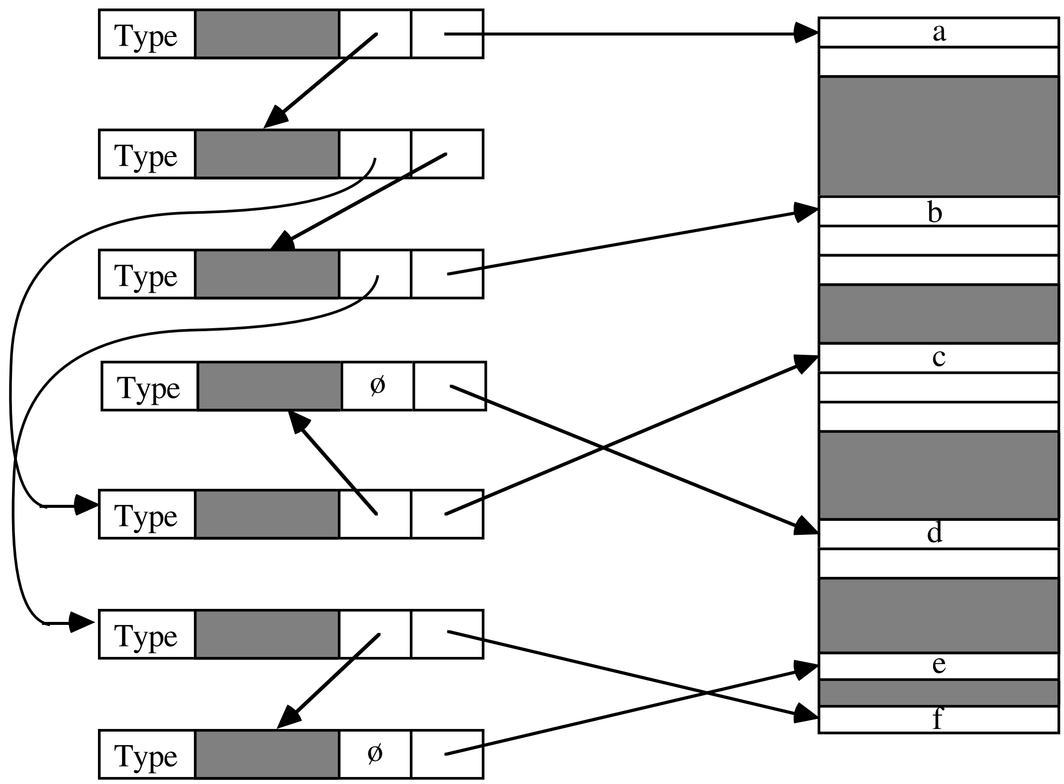


Figure 3 - Descriptor Pointer Examples

A different problem arises when the structure of the data object is also variable and dynamic. An example of this is a list where the number of elements and the depth of the elements change in time. The advantage of the unchanging location of a descriptor remains, but descriptors can no longer be contiguous and indexed. However, any subobject with a fixed structure within the dynamically structured object still possesses both advantages. Dynamically structured objects are connected through the use of pointers. Figure 4 shows a possible representation of a dynamically structured list. Each descriptor has two pointers, the first is to the next object in the list, and the second is to the value of the current descriptor. Note that no two descriptors in this figure are physically contiguous (although some could be).



(a (b f e) c d)

Figure 4 - Dynamically Structured List

However, it is typically unnatural to access a dynamically structured object by indices anyways, and an associative lookup would be more natural. Accessing can be done by following links (the slower way) or by directly accessing components of the object. For example, in a semantic network the number of nodes and their connections may change dramatically in time. However, a given node typically has a name, such as *elephant*. One can then directly access elephant, and from there begin to follow links to other related objects. Thus, associative names are used as handles to access an item or to enter into an object and begin accessing from that point.

4.3. Value Memory

The memory pointed into by the descriptors, and where actual values are stored, is called value memory. Values may be pointed to by multiple descriptors, allowing both sharing and distinct semantic views of data. Below are presented two potential value memory structures to give a flavor of the types of possible implementations.

In order to gain the advantages of memories with paging systems (contiguous data areas, minimized fragmentation, simple swapping and data allocation, and optimized access speeds), fixed size portions of value memory are allocated to store data values. However, rather than have one page size, different sizes and functionalities are made available to fit specific needs. The value memory should be amorphous in relation to data objects, and be able to efficiently hold the types stored. Small sized data (integers, characters, flags, short-strings, etc.) are stored within their descriptors and do not reside in value memory.

Since the mechanism of value memory is abstracted from other functionality, there are a number of ways in which it could be implemented, depending on technology improvement. A simple mechanism, similar to current methods for variable length string storage is to have an array of data words, each being m bits wide, where m is on the order of 64. Variable length data objects are stored in a structure called a *chunk* (Figure 5).

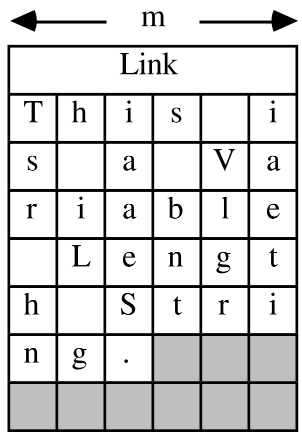


Figure 5 - Chunk

A chunk is made up of ~8 contiguous words. The last word of the chunk is a link to the another chunk if the data does not fit in the current one. The advantages in breaking the data into chunks rather than storing them in a single segment are similar to those supporting paged memory systems. In many cases, only part of the data need be accessed to accomplish a desired operation (i.e. *compare*). Also, it may be that only a subset of the data is to change (i.e. a *text replacement)* and only a subset of chunks need be affected.

The implementation of a chunk can be accomplished in a number of ways. If the exact size of the object is stored in the descriptor, and a chunk is forced to be full unless it is the last in the chain, then the exact number of chunks containing the data is known. This method makes for fast access of a data object since the data in a single chunk (except the last) is of a fixed size. The link is accessed first, and the next access can be initiated in a pipelined fashion with the access of the remaining words of the chunk. However, this is not the fastest method for dynamic update of data, since if the size of a portion of data is changed within a given chunk, new copies must be made for all subsequent chunks in the chain.

An alternative is to allow chunks in the chain to be only partially full. It would then be necessary to keep a count of the valid elements within a chunk as a field in the first word of the chunk, or to have an end of data marker at the end of the valid data. This method would make for somewhat slower access, but update would be easier. Only those chunks modified would have to be changed (modified, deleted, or added), and the other chunks would not have to be manipulated (with the exception of some link changes for chunks bordering new or deleted chunks). This method was used for storage of strings in the *symbol* computer [16].

Data objects having fixed size are stored in *hunks*. Hunks are larger fixed sized segments of the value memory. A hunk corresponds to more normal page sizes and data objects reside within word boundaries. An example is an array of integers as shown in Figure 6.

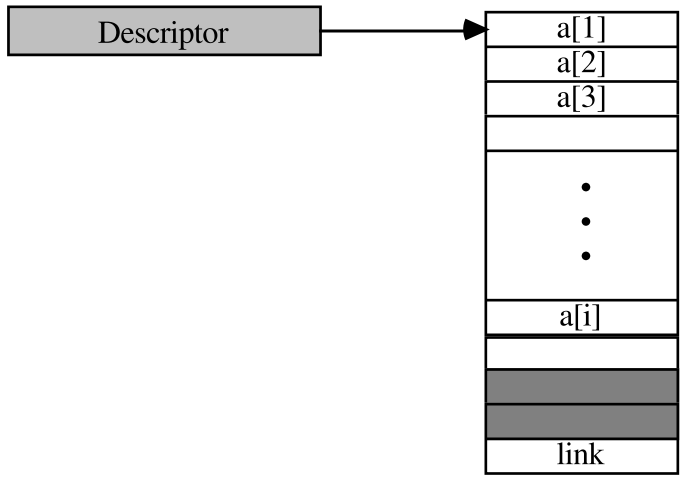


Figure 6 - Hunk-Based Array

There might be a small set of different sized hunks to choose from. (The same could apply with chunks). Each data value in a hunk could take up more than one word of memory. Conversely, there could be many values per word (i.e. a bit array). Optimization of storage and access, however, is invisible to the user and managed completely within the SM mechanism. Where each item is of the same size, any item can be indexed directly from the base address pointed to by the descriptor. If each item were of different fixed sized (i.e. like a structure in the C language) then offsets could be kept within the descriptor.

4.4. Smart Memory Model Layout

In this section, possible methods for laying out the SM on a single chip are presented. These are not meant to be detailed implementations, but are meant 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, or a single displayed bus or port does not preclude multiple instances of the same in an implementation. Figure 7 shows an abstract block model of a smart memory model layout (sizes do not correspond to actual area in a VLSI implementation).

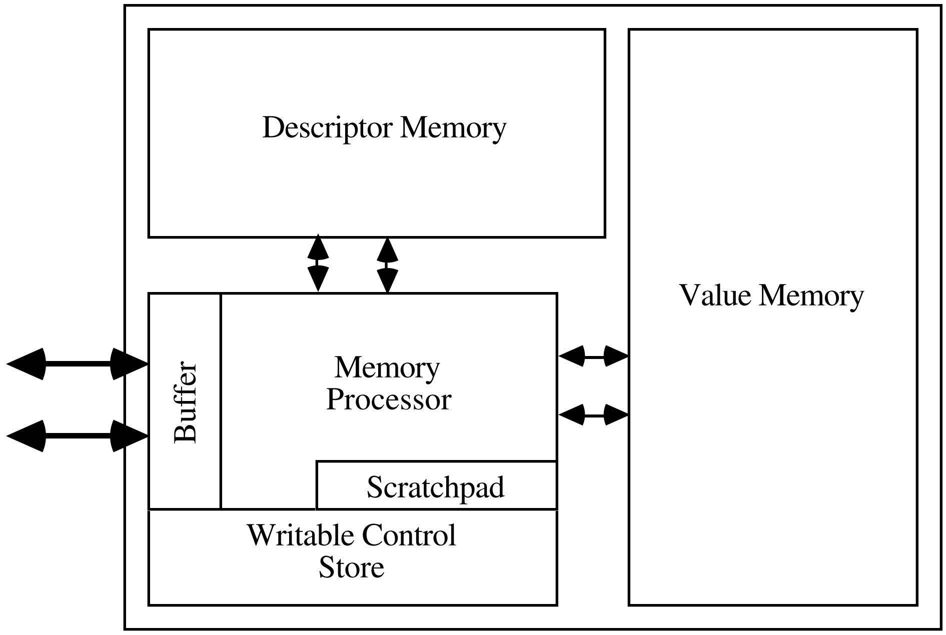


Figure 7 - Smart Memory Model Block Layout

The model contains a buffer, possibly a separate I/O processing mechanism, which sends and receives access requests and replies to/from the communication fabric. These requests are read by the processor as instructions to be executed. The buffer could be thought of as the instruction register, which is loaded by the accessors. The request is decoded and executed by the *memory processor (MP)*. The types of operations and their method of execution are 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.

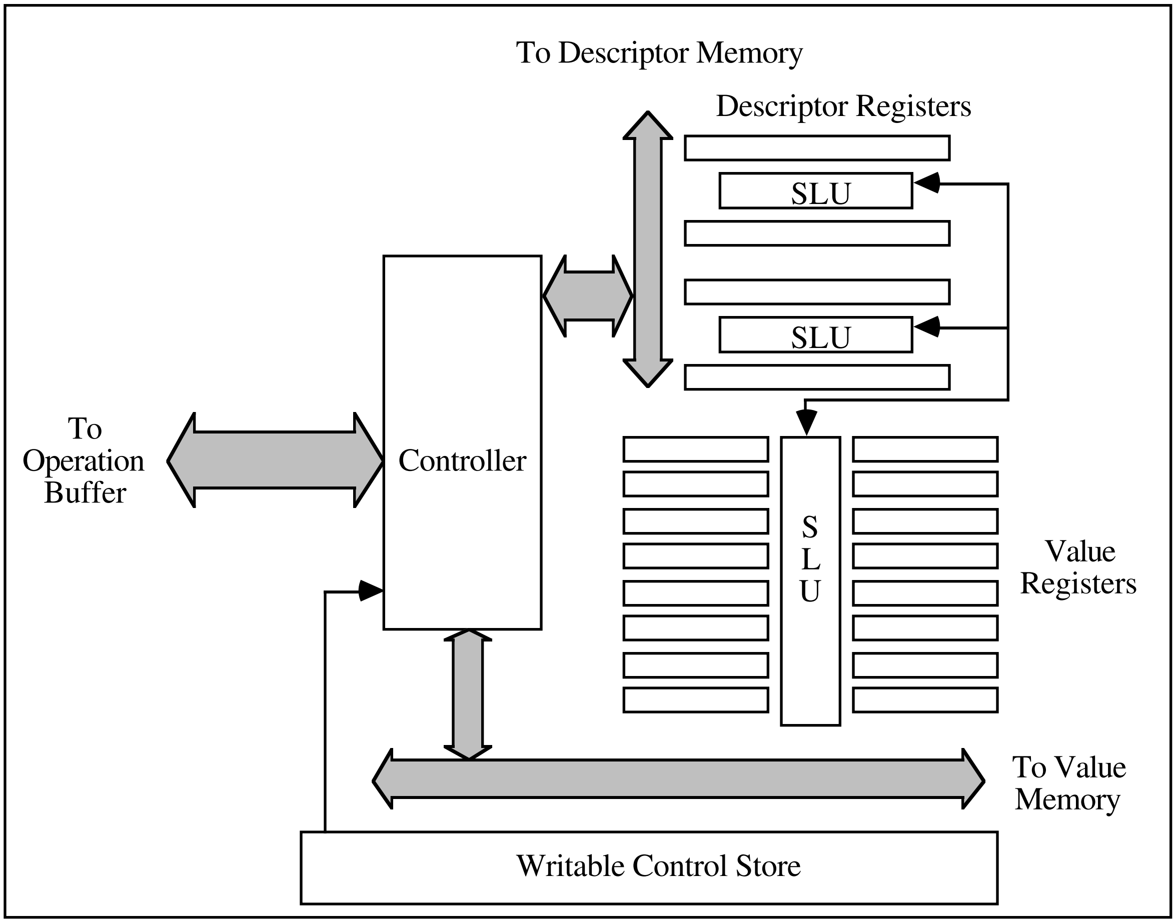


Figure 8 - Memory Processor Block Layout

A potential abstract layout of the memory processor is shown in Figure 8. Although using traditional technology, methods, and cells, the MP can be streamlined specifically for memory operations and the types of memory it supports. The controller receives requests from the buffer and the writable control store contains the method of execution. Two internal busses connect the MP to the descriptor and value memories and to their respective sets of registers. Two sets of two descriptor registers are shown. Between each pair of registers is a *Symbolic Logic Unit* (SLU), the counterpart to the traditional ALU. SLU and ALU functionality are similar in many respects, however, the SLU is streamlined specifically to symbolic functions (i.e. *compare*) and is programmable to specific functions. Internal data busses can be wider than typical external busses and can be made the same width as their corresponding registers. Two banks of value registers, each matching the word size of the value memory, are shown. SLU processing capability is available between these registers also. The number of registers in a bank is the same as a value memory chunk. Thus, two chunks can be compared or manipulated in parallel.

It might also be possible to load entire chunks in parallel from the value memory. If a copy of a large data object were requested, the MP could load chunks in a pipelined fashion, and the buffer could place them onto the communication fabric with no delays. Since the width of a chunk would probably be greater than the channel width of the communication medium (bus, link, etc.), the buffer would have to break up the chunk into a multiplexed sequence matching the size and peak speed of the communication medium. The MP would have a new chunk ready as soon as the last one is out, and thus no delay overhead need come from the SM. This direct passing of data from value memory to the communication fabric allows high speed throughput, while obviating any need for large memory areas within the SM for bundling up large objects.

Much obvious connectivity is not displayed in the figure for purpose of simplicity. (For example, a direct path from the pointer fields of the descriptor registers to the memory address buffer(s) of value memory would be logical). Also, no mention is made of possible disk I/O channels and processing capability, which could be necessary in a virtual memory scheme. The MP model is easily parameterizable to allow a class of heterogeneous chips. Size of both descriptor and value memory, writable control store, number of registers, data paths, ports, etc. are parameters which can be varied during design and fabrication.

5. Descriptor Processing

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

5.1 Descriptor Processing

Many operations on data can be accomplished without ever accessing the actual data values themselves. This occurs when the operation can use meta-knowledge about the data such as 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*.

An example of this is 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 data. In the case where the first characters of the string are equal, the first chunks containing each string would be fetched and compared. Figure 9 shows an example where two descriptors, loaded into descriptor registers, are directly compared. The shaded portions represent other attributes not used in the compare operation.

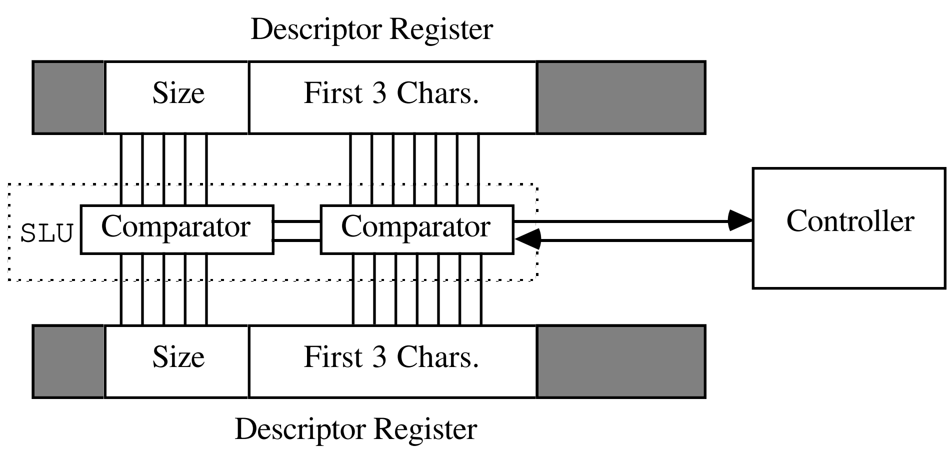


Figure 9 - Descriptor Processing

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.

5.2. 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. Most CAM devices 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 techniques are similar to those of associative processing [19], but with new operations specifically tailored for memory usage.

The basic notion is to make all or part of the descriptor memory a CAM. 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 10 shows a possible model for this computation.

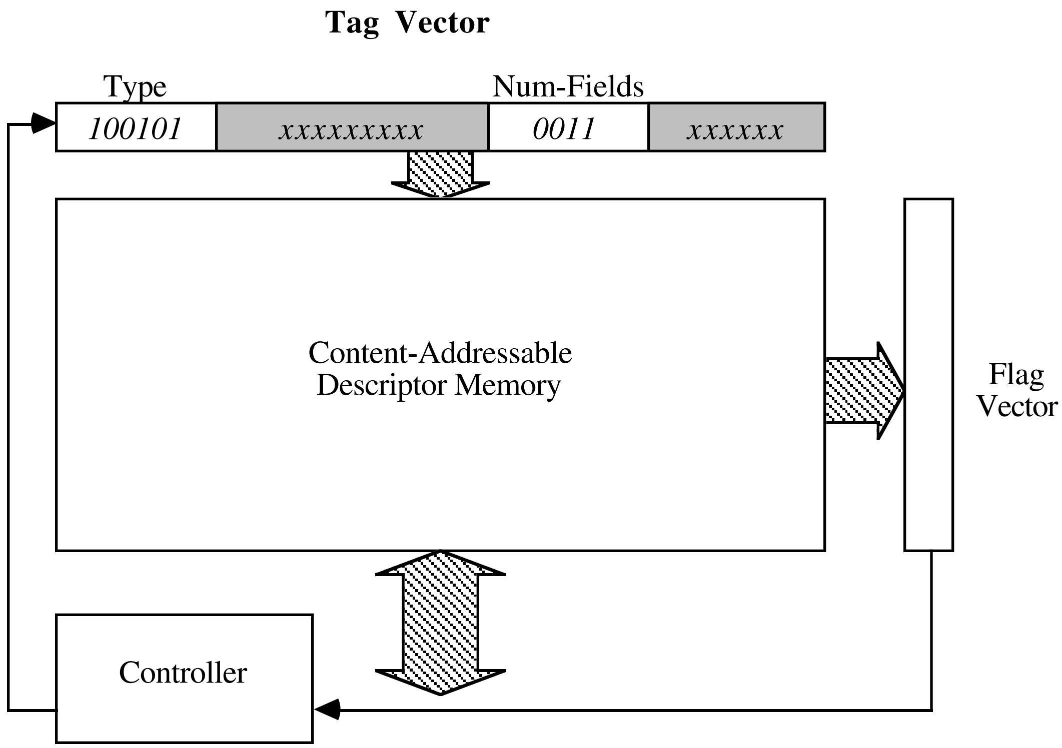


Figure 10 - Associative Descriptor Processing

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.

In keeping with the notion of heterogeneity, not all SM's need have this associative capability. It is felt that associative descriptor processing and descriptor pruning are operations which can attain high utility in symbolic memory processing, and thus, the expense of CAM is justified. More examples of these operations are mentioned in the SM functionality section.

Associative descriptor processing, 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 [13].

5.3. Single-Program Multiple-Data Method

Another important mechanism is called *single program multiple data (SPMD)* 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* [10]. 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.

5.4. Experimentation and Implementation

The MP model of SM uses current technology and extant hardware notions to derive an innovative architecture. By using traditional hardware and implementation techniques, it is felt that this model is well fitted for fast experimentation. Off the shelf processors and memories could emulate much of its functionality and the design is sufficiently modular that custom chips could be fabricated without undue difficulty. It has the flexibility, in terms of data type definition, operation definition, etc., to allow users to experiment and learn what types of functionality are really needed and cost-effective in an SM module. At the same time, it allows for creativity and innovation in how memory/processor interaction can take place.

6. Smart Memory Functionality

In this section, the types of functionality that the SM makes available are discussed. 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. Also, a single SM device would only support a 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).

6.1. 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). *Append* can also be done through descriptors if in-chunk size information is kept. In that case, two chunks could simply be linked together. Otherwise, a new copy of the last chunk of the first operand would be created. 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 would be later garbage collected), 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 data-base 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* [10]. *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.

6.2. 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, or deleted, or modified, or whatever.

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.

6.3. 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, as discussed in the previous section, but also allows other functionalities. Run-time type-checking of data can be used for encountering and signaling errors. The same information can be used to do *coercion* of operands with different types to a single type depending on the operation semantics.

Time-Stamping a field in the descriptor when an object is modified, created, etc., can be useful at both the algorithmic and system level. Algorithms can use time-stamping for synchronization, interpreting accuracy of values, identification, etc. The system can use it for swapping heuristics, load balancing, debugging, instrumentation, garbage collection, etc.

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.

When using distributed and parallel systems, synchronization of data becomes one the most important and difficult issues. The SM encapsulates the data such that the SM acts as a monitor [11], handling implicitly synchronization of the access of any object using internal locks and control. Explicit synchronization can also be supported through a visible lock bit in the descriptor and indivisible operations for setting it. Internal indices can be kept for array, list, or queue manipulation, such that *fetch next* type operations would be used rather than f*etch nth.* This allows a natural mechanism for multiple accessors who need to insert and retrieve values from a shared object. The SM can maintain multiple indices for different sets of processor(s) accessing simultaneously within the same data object. A descriptor tag can also be used for primitive support of *futures* [9], binding of logic variables, and automatic testing of constrained variables [5].

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.

System functions can be greatly enhanced through the SM mechanism. Usage and dependency statistics can aid load balancing and swapping methods. For example, the ADP (associative descriptor processing) operation *mark descriptors not touched in last 10 seconds*, could be used for swapping out data in a virtual system. Garbage collection methods can benefit from tags and other SM capabilities, while also being able to do garbage collection independently without tying up outside processing. Garbage collection of descriptors can be supported by reference counts in the descriptor which are maintained and updated through the memory processor. When a descriptor is deleted (reference count of 0), both it and the value memory to which it pointed can be attached to respective free lists by the memory processor. Memory allocation mechanisms and objects (i.e. free-list, cachability tags) would be encapsulated within the system. Information kept in tables for virtual and disk I/O can also be naturally maintained in descriptors.

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.

6.4. Higher Level Functionality

In this section, higher level and nontraditional accessing mechanisms are proposed. Many of these would require some additional specialized logic and/or software. These functionalities would require more logic capacity on the chip and appear to rival current processors, but with the predicted improvements in technology densities, our philosophy in silicon usage must change as we seek continued performance improvements.

The first is simply labeled *nonstandard retrieval*. This refers to access which causes each data value to be manipulated as it is retrieved. For example, *retrieve array A with all values squared*. The operation can optionally cause an actual change to the stored data. This operation can be used as a filter to allow only data with certain features or within a certain range to be retrieved.

Another notion is that of a *result cache*. When a process intensive operation has been executed on a data object (i.e. *sum-all-values, is-data-base-consistent*) the result could be stored at the SM (in the descriptor for small values). This could be the case whether the actual processing took place in the SM or at the accessor site. In a parallel environment, another accessor might have no knowledge of the accomplished computation, and ask for the same value to be recomputed. The SM could then give it a fast response using the cached value.

Another powerful mechanisms is that of *mapped on-conditions* tied to data structures [18]. These could be flagged in the descriptor with a corresponding pointer to the code to be executed. For example, a condition on a data object could be *when modified send value to accessor x,* or *if deleted delete array Z.*

Flexibility can be gained through the use of *virtual data* [8]. This refers to accessing of an object which does not really physically exist. A processing mechanism is called when the data is accessed, the value is computed and then returned. For example, an object A could be defined as the sum of objects B and C. However, B and C are constantly changing, so rather than store a value, B and C are summed each time A is accessed. The descriptor would contain a flag marking that it is a virtual field and information and/or pointer regarding its computation. Note that A could still be written into. If the written object is a value, then A would no longer be a virtual value. If the data written into a name is a program, then the object would become virtual. This mechanism is similar to the AI method of procedural attachment [3].

The *typing* mechanism allows for generic operations on the object. The operation *increment*, for example, could be simultaneously issued to different data objects, and each could execute it differently, according to their type.

A method which could become a primitive mechanism in pattern matching is that of allowing *magic* symbols in the operand pattern. These symbols (such as the wildcard "*") are commonly used in text editing to aid in searching for data. Other symbols include: match any character, match end-of-line, match anything except this character, etc. Other higher level search mechanisms could be implemented as primitive in the SM. A sampling of these comes from the Snobol programming language [7]. These include disjunctive pattern types (where the operand pattern is A or B or C ...), match everything up to a pattern, match everything between two delimiting patterns, etc.

A final notion is that of SM memory recruitment. Any SM has a limited amount of memory. Some objects operations can be executed more efficiently if the entire object resides within a single SM module. One way to support a larger object to be contained on one SM, is to allow the SM to dynamically recruit memory from an available pool. An SM could also return memory to that pool. One method of implementing this is to have multiple SM's together on the same substrate (WSI, board) and have extra memory blocks initially unallocated to a specific SM. These could then be dynamically reserved. Also, in a system having SM's with different memory sizes, chip-to-chip block transfers can allow the larger (growing) data object to be moved to an SM with larger memory.

7. Issues and Future Research

Other issues require decisions to be made by implementors depending on applications goals and system configurations. These include the memory name-space, whether to support virtualization in SM, and whether to use multiprogrammed or dedicated SM devices.

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.

8. Conclusion

This paper has discussed smart memory potential in general and proposed a specific smart memory architecture called the memory processor model. 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 parallelism and speedup for traditional and innovative memory operations.

9. 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] Danforth, S., "LNL, A Lazy Narrowing Language with Conditional Rules and Invertible Primitives," MCC Technical Report - PP-132-86, (Dec. 1986).

[6] Dennis, J., "Data Flow Supercomputers," *Computer*, Vol. 13, 11, pp. 48-56, (1980).

[7] Griswold, R.E, and J.F. Poage, I.P. Polonsky, The SNOBOL4 Programming Language, Prentice Hall, Inc., New Jersey (1971).

[8] Guzman, A., and M. Hermenegildo, "Constructs and Evaluation Strategies for Intelligent Speculative Parallelism - Armageddon Revisited," MCC Technical Report , 1987.

[9] Halstead, R. "Implementation of Multilisp: Lisp on a Multiprocessor," ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp. 9-17, (Aug. 1984).

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

[11] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Communications of the ACM, 17(10), pp. 549-557, (Oct. 1974).

[12] 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).

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

[14] Martinez, T. R., "Smart Memory: The Memory Processor Model," *IFIP International Conference*, Rome, Italy, March 1990.

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

[16] Rice, R., and W. R. Smith, "SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems," Spring Joint Computer Conference, pp. 575-616, (1971).

[17] Smith, B, "A pipelined, shared resource MIMD computer," *Proc. 1978 Intl. conference on Parallel Processing*, pp. 6-8 .

[18] Warren, R., MCC Parallel Processing Group, Private Conversation.

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