FUNCTIONAL CAPABILITIES OF ACTIVE MEMORY UNITS

by

Gabriel M. Silberman

Technical Report #199

February, 1981
FUNCTIONAL CAPABILITIES OF ACTIVE MEMORY UNITS

GABRIEL M. SILBERMAN.

Computer Science Department, Technion, Haifa, Israel.

0. ABSTRACT

The Active Memory Unit (AMU) concept and its functional capabilities are introduced. An AMU is a computer module which provides, independent of the CPU, a one-level memory space, subsuming in it all file I/O operations.

Conceptual designs for CPU, operating system, I/O, memory management and their interfaces, which make use of the AMU's capabilities, are presented. The potential impact of this approach on computer systems is reviewed, concentrating on those aspects related to Computer Architecture, Operating Systems and Programming Languages.

"Memory Directives", i.e. instructions to the memory unit to be executed in parallel with the unit's other functions, are proposed as a vehicle to efficiently utilize the processing capabilities present in an AMU environment.

KEYWORDS

Intelligent Memory Units.
Storage Machines.
Memory Hierarchies.
Parallel Processing.
1. INTRODUCTION

An Active (sometimes called "intelligent") Memory Unit (AMU) is defined to be computer module which provides, independent from the CPU, a one-level memory space [1], subsuming in it all file I/O operations. This is done by providing a very large (in the order of millions of Mega-bytes) addressing space coupled with processing capabilities which enable the memory unit to absorb all memory management functions, including virtual memory mapping and control, file management, etc. The processor requesting the contents of a memory location in the logical address space is completely relieved from any consideration of the residence of that information. This would be one step beyond todays virtual memory systems, where the programmer is not cognizant of physical memory residence, but the operating system, and naturally the CPU, are fully aware of the underlying mechanism which supports the logical address space.

An AMU would be set up as a hierarchy of storage units - the "passive" components - ranging from fast, expensive and small-capacity, to slower, cheaper and larger capacity units (see Figure 1). To make this unit perform as a self-contained active module within a computer system, a specialized control structure - the "active" components - are integrated into it; its tasks include interfacing to other parts of the computer system and managing the movement of information between the various storage levels as dictated by information requests. Figure 2 depicts the functional block diagram for the control structure in an AMU.
The control structure for the hierarchy is driven mainly by demand, but it will also attempt to foresee information needs and prepare for them by moving to faster storage levels those data and program modules which are likely to be needed in the near future, i.e., performing prefetching of information. Support for the prefetching functions include the usage of "Memory Directives" and the maintenance, within the memory unit, of "Program Profiles" based on resident history.

Memory Directives are instructions to the memory unit intermixed with the regular (CPU-oriented) instructions and data, whose production is delegated to compilers and other system utilities. This approach implies a major departure from a regular von Neumann architecture; this is due to the fact that the memory unit will be reacting to information requests by the CPU(s), and the processing invoked by these memory directives. The latter service is performed concurrently and asynchronously with the classic centralized computer control (in the CPU).

Program Profiles will try to anticipate the information needs of a certain program/user based on its past activity, moving information which is likely to be requested to faster storage levels. This would amount to a pre-execution set-up activity, much in the same way as file allocation in response to JCL commands.

If we look at the trends in computer systems, we notice that (1) processing capability is continuously becoming cheaper; (2) the memory hierarchy is taking a higher proportion of the total system cost; and, (3) memory management by a general-purpose CPU has become too time consuming.

Stein [2] notes that the above conditions guarantee a shift from striving to optimize CPU usage towards the efficient utilization of the memory hierarchy. This objective can be achieved by either; or both, multiprocessing or the unloading of memory-related work from the CPU(s) and
onto specialized processors (beyond today's I/O channels). Several designs in the latter field have been proposed lately, almost all implying a departure from von Neumann architecture towards configurations in which data becomes also a driving force in the execution of programs.

Denny et al. [3] present a specific example of logic-enhanced memories, tailored to a given task, in which simple processing elements are integrated with the memory cells. Bisiáni [4] also shows how to tailor architecture to a specific task, mainly by microprogramming a memory processor to access data according to its predefined structure.

A different, more flexible approach, is taken by Toursel [5]. His S-architecture is based upon simultaneous access of instructions and its associated data on several, concurrent control streams. This is supported by "data compilation", i.e. adjusting data organization to program structure using control primitives within the data stream.

We distinguish the above designs, which strive to facilitate memory accesses by the CPU, from the approach taken in Backend Networks [6]. These "networks" are oriented towards space and/or data sharing among several host (mainframe) processors, each operating independently, their motivation being mainly reliability, availability and resource sharing.

The Active Memory Unit concept requires extensive processing capabilities in the memory unit. By following this approach we primarily seek to alleviate the CPU, and the operating system running on it, of those chores associated with memory management and file I/O.

Furthermore, we aim to provide a more efficient overall architecture, based on a uniform, self-contained, one-level memory environment. This can be achieved by extending the applicability of those processing capabilities already present in the AMU, beyond the fields of memory management and file
I/O, to others such as operating system and programming language support. Also, the presence of Memory Directives simplifies multiprocessing, both with homogenous and significantly non-homogeneous processors.

Functions taken over by the AMU from the CPU and Operating System include I/O, virtual memory mapping, protection and control, data staging, resource allocation (possibly including the CPU), and inter- and intraprocessor communication and synchronization. Also, the environment of the AMU can provide for better reliability and growth potential.

In addition to opening-up ways to improve performance, the AMU provides better solutions to existing problems. For example, the encapsulation of all the above functions within the AMU results in improved reliability, security, easier sharing, and graceful degradation.

The most powerful, and inherently flexible, feature of AMU's is the presence of Memory Directives. They constitute the main vehicle by which the processing power in the AMU provides support in areas such as task scheduling, prefetching and both inter- and intra-CPU communication.

We shall present in Section 2 the main effects of the AMU approach on the fields of Computer Architecture, Operating Systems, and Programming Languages. Section 3 reviews the mechanisms associated with Memory Directives, and their applicability in various fields; an example is used to illustrate the suitability of this principle in the AMU design.

Improvements to the management of the memory unit which are made possible by the capabilities in an AMU are given in Section 4. Section 5 presents some thoughts on the design, testing and evaluation of a computer system based on the AMU approach, aiming at a framework for the development of such a system in all its aspects.
2. IMPACT ON COMPUTER SYSTEMS

Active Memory Units should have a major impact on computer systems once their varied advantageous features are evaluated. This section points out the main effect their introduction to commercial systems would have in the fields of Computer Architecture, Operating Systems and Programming Languages.

2.1. COMPUTER ARCHITECTURE: Besides the obvious effect on the architecture of the memory itself, an AMU greatly affects both the CPU and I/O processing structures.

By taking away functions normally performed by the CPU and putting them into the memory unit (e.g. virtual memory control, generalized I/O, etc.) we simplify the CPU's design. All the mechanisms associated with these functions (e.g. I/O-related interrupts), which are usually extraneous to the basic processing function of the CPU, can then be removed from the CPU's logic, resulting in a "cleaner", cheaper processor.

It should be noted that since CPU prices keep dropping, it is not of utmost importance to utilize its time to the fullest in order to make it cost effective ([7]). Actually, a system with an AMU with large enough bandwidth coupled with a number (as little as two) of partially utilized CPUs may effectively achieve the same performance of a single, fully utilized CPU.

Miller [8] reviews several studies performed on the degradation of tightly coupled multiprocessors caused by the bus, memory and software contention for various systems (e.g. IBM 360/67, C.mmp, UNIVAC 1108). Her conclusions are that the degradation can be minimized if the mutual exclusion "locks" are each restricted to relatively small "critical" data (not code), the processor/memory concurrency is high (e.g. through high
bandwidth shared-bus), and the processors are symmetric in terms of peripheral equipment access. All of the above conditions can be easily met in the context of an AMU.

This last feature affects also the design of operating systems as it will be noted later.

Turning our attention now to CPU context switching and status saving, we observe that a memory mapping of CPU registers (much in the way Texas Instruments' 990/9900 series [9]) would fit within the AMU concept in the following sense.

Having the CPU registers within a fast cache (20-50 ns access time) would allow the AMU to share task scheduling chores with the CPU on one hand. On the other hand, performance (one reason for having registers within the CPU in the first place) would not suffer greatly in terms of instruction speed.

As far as I/O processing is concerned, its mapping into data structure processing, which in itself can be taken over by the memory management (supported by the memory directive mechanism), completely subordinates I/O to the memory unit. This implies a radical departure from the usual I/O processor(s)-to-CPU(s) relationships common in passive memory systems.

This fact does not affect greatly the I/O processing system at its lowest level (channel programs, controllers, etc.), it rather has a major impact on the interface with the other components of a computer system, namely the CPU and memory. In this context, the CPU is further simplified by eliminating all I/O-related functions from its logic.

Besides alleviating the CPU from the burden of special I/O instructions (as it is done in the PDP-11 [10]), the AMU - through the usage of Memory Directives - enables the "I/O" ports to disassociate themselves from
fixed memory locations, resulting in further flexibility. These predefined memory locations usually associate special meanings to their contents, e.g. cylinder and track addresses (for disks), ready/busy and error flags, etc. All this becomes unnecessary in an AMU environment, shielding the operating system (or user) from all physical I/O considerations.

2.2. OPERATING SYSTEM: In order to illustrate how AMUs can simplify operating systems by assisting on some of their chores (e.g. virtual memory management, task scheduling, etc.), we compare the step-by-step processing of a memory request in a virtual memory, cache-assisted system (e.g. IBM's OS/VS2 [11] on a 370/168), and an AMU-based system. The steps in each system are given in Figures 3 and 4 respectively.

Comparing the two schemes, we see that no functions are being eliminated from the memory access process, rather they are either being removed completely from the CPU (and operating system) or split with the memory unit.

The hardware support for most of the memory functions (e.g. address mapping) already exists outside the CPU (e.g. cache support), only the scope of their applicability is widened by the AMU scheme.

The idea of alleviating the CPU of some of its functions (e.g. I/O) is not new, the CDC 6000 series [12] does it with its Peripheral Processors (PE-3). The innovation in the AMU scheme resides on how the recipient of the
responsibility (the memory), and the additional capabilities which can be implemented using the processing power in its new location.

As mentioned in the Computer Architecture section above, the AMU's capability to service several CPU-s (which might be only partially utilized) to achieve the desired throughput performance also affects the Operating System. Since CPU costs are constantly dropping, it is not longer necessary to multiprogram at a high degree in order to keep the CPU busy at all costs, thus avoiding the partition of the fastest storage levels among too many processes, giving too little space to each. The approach to throughput improvement should shift to emphasize multiprocess ing and larger memory bandwidth (see [8] for details).

The multiprogramming degree has always been a delicate factor, balancing between high CPU utilization and the thrashing (doing more work on information transfers than actual processing) caused by a process lacking enough space to hold its "working space" (set of program segments/pages needed for efficient process execution). Finally, in the subject of I/O, the Operating System can share its burden with the AMU, not only where basic I/O is concerned (e.g. starting channel programs, handling I/O interrupts), but also in the implementation of file access methods (e.g. indexed files). The support for these access methods constitutes a large percentage of the total overhead volume and time introduced by operating systems (Dunigan [13] cites IBM's QS/360 with more than half a million bytes just for access method support routines).

2.3. PROGRAMMING LANGUAGES: The major effect of AMU-s on programming languages is, by far, the elimination of explicit file I/O. This "artificial" feature of high-level programming languages has been always a major problem in asserting the correctness of programs. Also, it is a rather
Therefore, the elimination of file I/O as a means of circumventing the limitations of memory can be considered a step towards better programming languages, and thus "better" programs.

As examples of the above claim consider the widely used algorithmic languages, ALGOL and PASCAL. The original description of ALGOL 60 [14] does not include I/O at all, the claim being that this feature is machine dependent and thus would be specified independently for each implementation of the language.

In the PASCAL case, only very primitive (as compared to PL/1 or FORTRAN for example) I/O operations and files are defined [15,16], with the claim that more sophisticated file structures may be constructed from this basic capabilities. In practice, only sequential files exist in PASCAL, any other organization would have to be simulated by these files or added as a "special feature" to a particular implementation of the language.

Also, the AMU associates semantics with memory accesses, e.g. a write operation which implies output to a sink file (e.g. line printer) results in a set of actions different from those caused by a simple write to a variable in a program.

Furthermore, the concept of Memory Directives parallels that of Abstract Data Types [17,18], in the sense that the actual actions performed on the data are internal to the memory unit, and thus transparent to the user. The access of data is only through allowable operations can therefore be enforced quite simply through the specification of appropriate Memory Directives.
3. THE MEMORY DIRECTIVE MECHANISM

Memory Directives are instructions to the memory unit which are intermixed with the regular (CPU) instructions and data. Some directives are produced by compilers, assemblers and other utility programs (e.g. linkers and loaders), others are created within the memory unit itself (e.g. I/O device handling).

These directives provide the main tool by which several functions, traditionally implemented by the CPU-operating system tandem, can be taken over by the memory unit. Furthermore, their capabilities are not limited to carrying out existing functions (e.g. virtual memory support), but rather their potential make feasible the implementation of new functions at an acceptable cost.

The design of the invocation and processing mechanism to support memory directives is intimately related to the overall design of the AMU, and are to be taken into account when the question of processing capability distribution and communications are tackled.

Memory directives are brought to the attention of the memory unit control when a CPU requests the contents of a memory location which is specially tagged. This tag indicates that the contents of that location is a directive or would point to the location which contains an associated directive.

The memory unit control then reacts to the memory directive and produces the information requested by the CPU in one of two possible ways. If the directive specifies a function needed by the present request (e.g. an indexed record search – see Example below), then it is executed and its results furnished to the CPU (task switches notwithstanding).
In the second case, the directive has no direct bearing on the present request (e.g. a prefetch of a subroutine for later usage). This results in the CPU getting the information it requested while the memory unit continues processing on the service invoked by the memory directive.

It should be noted that memory directives are completely transparent to the CPU, that is to both the Operating System and normal user. Even though a directive is triggered by the CPU (by accessing the address associated with the directive), the fact that it has been invoked, or its execution (carried out completely within the memory unit) are unknown to the CPU, excluding the possibility of an eventual delay in servicing of the access request, or the need for a task switch.

Memory directives can be used to replace traditional file I/O (see Example below); for support of operating system functions (e.g. context switching, communication and synchronization, sharing and protection, etc), and in managing the memory hierarchy to improve its performance (e.g. prefetching). As an example, let's consider the implementation of file I/O using Memory Directives.

**EXAMPLE**

It is possible to simplify further I/O structures over the traditionally accepted, for example over what the PDP-11 [10] does. Instead of having special I/O instructions within the CPU repertoire, the PDP-11 and others use pre-assigned memory addresses to act as interface ports for the various I/O devices. This still requires special treatment of I/O in the sense that those locations are not treated like memory, but rather assign different meanings to their contents (i.e. cylinder and track addresses, etc.).
Using memory directives, we carry the same idea further. These "I/O addresses" do not have to be fixed and neither their manipulation has to be done keeping special fields in mind. A reference to an address which has an associated I/O directive with it will perform the equivalent of I/O, but as far as the user (or the Operating System) is concerned all that was performed was a (memory) READ/WRITE on that location.

Support for this mechanism would have to be supplied by the equivalent of File Descriptors or Data Control Blocks, set up much in the same fashion used today in operating systems, that is at job initiation or by an explicit "File Open" command (which could in itself be performed by a memory directive).

This scheme is not limited to source/sink sequential files (in which case it would act as an I/O spooler), but lends itself to more sophisticated file access methods, e.g. random access and indexed-sequential files.

Taking over some of the Operating System's functions associated with file access methods (e.g. indexed-sequential files, random files, etc.) results in significant savings, both in space and time complexity of the operating system. Dunigan estimates in [13] that the elimination of most access methods from IBM's OS/360 results in savings of about half a million bytes of OS code, this in a system which should be able to run (together with user programs) on a minimum of 32K bytes of main memory. Also, about 55% of the system-supplied utilities directly support the eliminated access methods, thus a significant cut in the CPU's involvement with these chores can be expected.

For the random access method, activating a memory directive and providing it with the record address would suffice to complete the desired operation, again assuming that the necessary tables were set up by an earlier memory directive. In the case of indexed files, all the index tables
lookup procedure can be performed within the memory unit by providing the record key to the appropriate memory directive.

It is worth noticing that since processing the directives does not interfere with CPU processing, their activation can be performed as early as the need for them is realized. For example, in the case of indexed files described above, a search for a record may be initiated as soon as its key becomes known, while processing by the CPU of a previously obtained record is still in progress.

Having a uniform memory space means treating I/O files the same way programs are treated, i.e. they are subject to same basic management rules for moving within the hierarchy (up- and down-staging — see next Section). In particular, addressing a portion of an I/O file would cause the staging up of some related (not necessarily adjacent) part of that file; this is not desirable in the case of output-only files (i.e. a file created by output).

The use of output-only files (e.g. output to a line printer) is to be recognized and handled by creating “empty” space, rather than fetching a block containing no information whatsoever. Again, the complete knowledge, within the memory unit, of the nature of the access, contributes to the "tailored" memory access mechanism.

A memory unit driven by these directives differs from a current virtual memory system, most noticeably in the timing of their related information transfers. In the former case, the timing of transfers, both for demand and prepping, are not necessarily synchronized with CPU requests (for reading or writing) as it is the case with the latter.

The above Example has shown how the I/O structure is affected by the AMU environment, in the next Section we shall see another aspect of the
computer system for which the AMU approach supplies an improved framework, namely memory unit management.

4. MEMORY UNIT MANAGEMENT

We have seen through the Example in last Section how the AMU environment, together with the memory directive mechanism, combine to offer a better configuration for the I/O structure.

The potential for computer system improvement under the AMU approach is very promising and touches many of its aspects. We show here how an AMU would improve performance by implementing hierarchy management techniques which, although recognized as desirable, are prohibitive (in terms of CPU overhead) in a passive memory environment.

Also brought to attention are the advantages of encapsulating all the tasks which pertain to a given system function (e.g. sharing and protection), within a single module in the computer system.

Prefetching: The staging of information to faster levels in anticipation of its usage can improve the performance of a processor, since its requests to memory can be serviced within less time. Smith shows in [19] how a simple technique, such as sequential prefetching, when applied to the fastest storage level (cache) can improve the processor's instruction execution rate by 10 to 25%. The machines he specifically considers are the IBM 370/168 and Amdahl 470V/5.

A review of other work dealing with prefetching effects on performance can also be found in Smith's paper, covering aspects such as I/O prefetching and buffering techniques, database systems, non-sequential and array-related paging.
The above study cites the high CPU overhead as the main reason for not applying the same technique to other levels below the fastest. This argument does not carry over to the AMÜ case, where very little or no CPU overhead is incurred.

In a study performed by Stuttgen [20], in which prepaging techniques were analyzed, a significant feature came to attention:

The "forward window size", i.e. how far ahead does the prepaging algorithm has to "look", can be as little as 50 references (see Figure 5), certainly feasible for a compiler to predict in most cases. Moreover, the addition of an extra paging channel can reduce this size to 10-20 without degrading performance (see Figure 6).

---

**Down-Stage:** Migration of data which is not being used to slower levels results in the equivalent of an archival system (i.e. moving unused information to off-line storage), without the usual administrative pains associated with it. This is actually the functional opposite of prefetching, and it can be looked at as the extension, to other levels below the fastest ones, of page replacement algorithms.

The generation of backups, in itself an important system function, may be integrated in this context and implemented independently from any managerial procedures, allowing for undisrupted service. Also, the backup frequency and extent can be controlled down to individual files or groups.
of files, making the whole procedure more reliable and efficient.

Furthermore, backups are always current as they are performed in an "incremental" fashion, i.e. updating only those files which have been changed or created since last backed up. This results in reduced backup-related work since unchanged files need not be copied again and again.

**Store-Through:** The AMU environment enables the implementation of a pseudo store-through mechanism using the cheaper write-back technique. This is accomplished by giving the hierarchy management facility the capabilities to clean-up "dirty" pages before they would have to be written back during a page replacement procedure. The procedure is based upon "sneak writes" (stealing unused cycles in the transfer mechanism) of those pages which are the most likely candidates to be replaced, using the active units which control transfers between each adjacent pair of levels, whenever they are idle for a reasonable period.

Finding out when the transfer control units are idle, or are expected to be, requires the cooperation of a centralized global controller and the local units. The global controller follows the various states of the memory unit's "customers" (CPU-s), and in some cases can anticipate the general direction they will follow in the immediate future, as far as memory requests are concerned.

For example, when a new segment (say containing a subroutine) has been explicitly prefetched (see below), a move to that segment is anticipated and thus some of the transfer units will undoubtedly be idle. On the other hand, if more than one path exists for direct transfers between two levels, one of the extra paths can always be used for this "cleaning" activity, if idle when the need arises.
This "sneak page cleaning" is completely transparent to the CPU as it is done without notifying (or requiring any assistance from) it. In this manner the number of "demand" write-backs (i.e. those executed when a dirty page is selected for replacement) is minimized. Notice that this capability is effectively reducing the fetch time for a page, since the page it would be replacing does not have to be written back before its space is taken over by the new page.

Store-through is a desirable feature in storage hierarchies, mainly because of the conceptual simplicity obtained in the hierarchy management facility. This is specially due to the absence of data consistency problems which in itself makes sharing of information much easier. The usually negative effect on performance associated with this feature [21] (in a passive memory environment), makes it only marginally cost effective, a fact which does not come into play when using the technique described above.

Memory Sharing and Protection: Both these functions can be completely absorbed by the AMU. This allows for non-homogeneous processors and/or operating systems to be tightly coupled on one hand, while not being forced to use one particular memory access scheme (both physically and logically). Actually, the memory can appear as organized quite differently to each processor using it, this being accomplished through the use of Memory Directives, in a manner analogous to the memory unit in the Nanodata's QMX machine [22].

The encapsulation of all security and integrity related functions, together with the virtual memory implementation, within a single functional unit (the Memory), can only enhance their reliability and simplify their implementation. Furthermore, the same mechanisms can be used for controlling access to files, since I/O functions are also performed through the AMU.
The above examples have shown us how an AMU can provide the capabilities to

* Supply large amounts of storage at low cost/performance.
* Alleviate the load on CPU(s) and operating system(s) of paging, staging and memory mapping considerations, resulting in improved security and easier sharing.
* Screen the operating systems and the user from File I/O, by providing a one-level memory and supporting File Access Methods.
* Support parallel data migration without CPU intervention (e.g. backup, down-staging and prefetching).

In addition, the AMU can provide support for novel storage structures (logically addressable) to accommodate:

- Permanent Storage
- Queues
- Sink and Source sequential files
- Archival Storage
- Transportable Media
- Primitives for Communication and Synchronization (i.e. mailboxes and semaphores) among processes running on one or several CPU-s.

5. DESIGN ASPECTS

Being a radical departure from traditional computer systems, the AMU concept implies major changes on all components of the system. These have to be performed using the capabilities of the AMU as the unifying and driving force.

Following are some questions which will be tackled during the design and evaluation phases of an AMU-based computer system, grouped by the
5.1. COMPUTER ARCHITECTURE: Besides the obvious changes to CPU architecture, the main design questions are those related to the memory unit itself. Within these, the distribution of the processing capabilities and tasks over a number of functional units, becomes critical to the AMU approach.

Since the managing of the memory unit is partitioned into separate tasks for execution by these functional units, the problems of communication and control interaction among all the processing modules becomes the main concern in the design of the memory unit.

Also to be addressed is the procedure(s) by which the technologies and sizes of the storage levels in the memory hierarchy are to be chosen. The interconnections among these levels and their associated processing elements (and their technology) are also subject to a selection procedure.

As far as the I/O structure is concerned, the controllers and devices are likely to remain unaffected, except maybe for further buffering; on the other hand, it is not clear how the present design of the "channel" functions fits within the AMU concept.

5.2. OPERATING SYSTEMS: A major redesign of the Operating System is expected, mainly because of the migration of functions and responsibilities from the CPU to the AMU. The main question here is which functions, or parts thereof, are to take part in this migration.

Here, Memory Directives will play a major role, being the vehicle for the flexible implementation of most operating system functions. In this context, adaptation of file access methods and their support to make use of the AMU potential, should prove to be a major endeavour.
5.3. PROGRAMMING LANGUAGES AND UTILITIES: The central problem in evaluating the several design alternatives to answer the above questions, is the definition and gathering of test data to drive simulations and check the accuracy of theoretical models.

The specification and implementation of compilers, assemblers and loaders (among other utilities), which produce code for an AMU-based computer system, is a major step towards the solution of the test data problem. It should be noted though, that these utilities are affected in turn by the various design decisions in the other areas mentioned above, and therefore we expect a strong interaction among all relevant fields to be touched by the overall AMU design.

6. SUMMARY

We have defined the concept of Active Memory Units and the environment created by their presence in a computer system. Its place among new directions in computer systems was examined, and its potential impacts on the fields of Computer Architecture, Operating Systems and Programming Languages were detailed.

The Memory Directive mechanism and its place within an AMU-based computer system were examined. The very idea of these directives is intimately connected with the basic concepts of the AMU approach to the overall design of a computer system, in all its aspects. An example detailing the usage of Memory Directives in the field of file I/O was given.

The management of the AMU itself presented several techniques which are particularly suitable for implementation in its environment, resulting in potential performance improvements, as well as other advantages (e.g. improved security). Some thoughts on the design, testing and evaluation of
an AMU-based system were also given.

7. ACKNOWLEDGEMENTS

I would like to thank Prof. Gideon Frieder of SUNY/Buffalo for suggesting the topic for this research, and for his many contributions to it.

8. REFERENCES


<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Size</th>
<th>Speed</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cache</td>
<td>0.1 MB</td>
<td>&lt;75 nsec</td>
<td>MOS/Bipolar RAM</td>
</tr>
<tr>
<td>2</td>
<td>Main Memory</td>
<td>10-100 MB</td>
<td>500-900 nsec</td>
<td>MOS RAM</td>
</tr>
<tr>
<td>3</td>
<td>Pseudo-RAM (PRAM)</td>
<td>10-1000 MB</td>
<td>1-10 msec</td>
<td>Magnetic Bubbles, CCD's.</td>
</tr>
<tr>
<td>4</td>
<td>DASD</td>
<td>500-5000 MB</td>
<td>10-20 msec</td>
<td>Disks &amp; Drums</td>
</tr>
<tr>
<td>5</td>
<td>Mass Storage</td>
<td>5000-50000 MB</td>
<td>0.1-5 sec</td>
<td>IBM 3850 - like</td>
</tr>
<tr>
<td>6</td>
<td>Archival/External</td>
<td>∞</td>
<td>minutes</td>
<td>Tapes</td>
</tr>
</tbody>
</table>

**Figure 1** - AMU Storage Hierarchy
Global Control
- External Interface
- Depth determination

Directives Processor
- Memory Directives
  Interception and Processing

Local Control
- Address Mapping
- Local Transfers & Buffering

Shared Control
- Control for slower storage
  (e.g., requiring channel activity)

Figure 2 - AMU Control Structure
<table>
<thead>
<tr>
<th>Step</th>
<th>Server</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU</td>
<td>Process requests contents of a memory location (say ADDRESS).</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cache</td>
<td>Search for ADDRESS - If successful, done.</td>
<td>CPU is waiting</td>
</tr>
<tr>
<td>3</td>
<td>Main memory</td>
<td>Search for ADDRESS - If successful, move contents to CPU &amp; containing block to cache. Otherwise, signal &quot;page fault&quot;.</td>
<td>Transfer is done in parallel, with inter-leaving, making up for the speed difference.</td>
</tr>
</tbody>
</table>
| 4    | CPU    | Page Fault handling:  
- replacement algorithm  
- set-up for physical transfer(s) | After the set-up is finished, the CPU usually switches context to another task. |
| 5    | I/O channel | Physical transfer. When done, signal to CPU |          |
| 6    | CPU    | Handle I/O interrupt and schedule process for resumption |          |
| 7    | CPU    | Resume execution of process which caused the page fault | Time of resumption depends on relative priorities & scheduling algorithms |

*Figure 3: Memory Request in a Virtual Memory, Cache-Assisted System*
<table>
<thead>
<tr>
<th>Step</th>
<th>Server</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU</td>
<td>Process requests contents of a memory location (say ADDRESS)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AMU</td>
<td>If request can be satisfied within time T, process it &amp; supply results to CPU. Done. Otherwise signal &quot;memory fault&quot;.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AMU</td>
<td>Memory fault handling: - switch CPU contexts - set-up and perform physical transfer(s)</td>
<td>Switch is direct (using memory-mapped CPU status), or indirect (interrupt). Actual transfer may be delegated to I/O channels.</td>
</tr>
<tr>
<td>4</td>
<td>AMU</td>
<td>ADDRESS has been brought to within threshold T. Update scheduler's tables to signal requesting process Ready</td>
<td>Concurrency of transfers with other CPU requests is achieved by spreading tasks over separate functional units.</td>
</tr>
<tr>
<td>5</td>
<td>CPU</td>
<td>Resume execution of process which caused the memory fault</td>
<td>Immediate resumption can be forced in the memory-mapped status case.</td>
</tr>
</tbody>
</table>

Figure 4: Memory Request in an AMU-Based System
Figure 5: Processing Time vs. Forward Window Size ([20]).

Figure 6: Performance Comparison of One and Two-Paging Channels ([20]).