Addressing Mechanisms for Large Virtual Memories

J. ROSENBERG[1], J. L. KEEDY[2] AND D. ABRAMSON[3]

[1] *Department of Computer Science, University of Sydney, NSW 2006, Australia*

[2] *University of Bremen, Germany*

[3] *Division of Information Technology, CSIRO, Melbourne, Australia*

Traditionally there has been a clear distinction between computational (short-term) memory and filestore (long-term memory). This distinction has been maintained mainly due to limitations of technology. Recently there has been considerable interest in programming languages and systems which support orthogonal persistence. In such systems arbitrary data structures may persist beyond the life of the program which created them and this distinction is blurred. Systems supporting orthogonal persistence require a persistent store in which to maintain the persistent objects. Such a persistent store can be implemented via an extended virtual memory with addresses large enough to address all objects. Superimposing structure and a protection scheme on these addresses may well result in them being sparsely distributed. An additional incentive for supporting large virtual addresses is an interest in exploiting the potential of very large main memories to achieve supercomputer speed. This paper presents hardware and software mechanisms to implement a paged virtual memory which can be efficiently accessed by large addresses. An implementation of these techniques for a capability-based computer, MONADS-PC, is described.

Received January 1990, revised March 1992

1. INTRODUCTION

Virtual memory was first introduced on the Atlas computer[1] as a technique for extending the apparent size of the memory by encompassing part of the secondary store. Hardware and software techniques were developed to automatically handle the transfer of code and data between main and secondary memory, thus relieving the programmer of this task. These techniques are now well understood and most modern computers include support for virtual memory, usually based on fixed-sized pages. Such implementations of virtual memory are designed to support the computational data of executing programs (i.e. the stack, code and temporary data structures) with the long term (permanent) data being accessed via a separate filestore. Thus the address size supported is quite modest, typically no greater than 32 bits.

However, there has been considerable interest in supporting much larger virtual addresses. This interest stems from an extension of the virtual memory concept first introduced in the MULTICS system[2] and later in the IBM System/38.[3] The extension involves the introduction of techniques which allow files to be addressed as segments in the virtual memory, thus eradicating the need for a separate filestore. More recently this concept has been extended further to allow arbitrary data structures to be created and manipulated in a uniform manner, regardless of how long they persist. This concept of orthogonal persistence has been incorporated in several programming languages, resulting in a flexible and powerful programming environment.[4–8] Such systems usually store all objects (code and data) in a persistent store which automatically takes care of the transfer between main and secondary memory.[9,10] Systems supporting orthogonal persistence have several advantages over conventional systems. These can be summarised as follows:

- The number of transfers between the main and secondary memories can be considerably reduced. For example in most conventional systems the decision to start executing a new program requires that the entire program be transferred from its permanent location in filestore to a new temporary program image in the computational store. If the computational memory is organised as a virtual memory there is a strong possibility that most of the pages or segments of a large program will then be removed by the virtual memory mechanism out of the main memory before being used or accessed and will only later be returned to main memory as page faults occur. Thus in such systems, the program load operation largely results in the movement of program parts from one area of the disc storage to another area. This not only affects the loading of user programs, but also the initial loading of the operating system. (Some virtual memory systems, e.g. Burroughs B6700,[11] avoid this problem.) Such load operations do not occur at all in persistent systems because the code is maintained within the persistent store.
- Persistent systems avoid duplication of mechanisms. Two significant examples of this are the dual protection mechanisms which have to be supported (one to protect running programs from each other, the other to guarantee privacy of files), and the dual synchronisation mechanisms (e.g. semaphores in the computational memory, file or record locks in the filestore). In both cases the same fundamental problems are being solved and in both cases the filestore mechanism is usually clumsier and less efficient because it cannot be supported directly in hardware or microcode.
- Persistent systems allow the same software to be used to manipulate temporary and permanent data. Distinct traditions have grown up about the organisation of data structures in computational memory (e.g. lists, queues, trees, stacks, hash tables, arrays) and in the filestore (e.g. sequential, index sequential, direct, random, hash random, tree-structured files). These are usually taught to students in separate parts of a computing course, and are

regarded by many programmers as quite unrelated techniques. Yet fundamentally they rely on the same principles and can profitably be considered as variants of a single set of models for organising data. It is only because computational data and filestore data are traditionally accessed using different mechanisms that the two are handled in different ways.

A persistent store can be implemented as a virtual memory. However, since all data (both temporary and permanent) resides within this store, the virtual addresses must be large enough to address all storage connected to the machine. Current disc technology would suggest that addresses in the order of 40 bits would be required. This assumes a flat name space. If, as is more likely, the name space is structured then even larger addresses will be required. Since any program can potentially generate any address, some central sharing and protection scheme must be superimposed on the addressing scheme.

An attractive way of achieving this is to use capabilities, an idea first proposed by Dennis and Van Horn.[12] Ideally capabilities identify objects using unique system-wide non-reusable names, thus ensuring that a capability which refers to a deleted object cannot be used to access a new object. To effect this the names in capabilities must be large enough to identify uniquely all the objects which exist during the lifetime of the system, and must therefore in most cases be substantially larger than 40 bits. Presentation of a capability provides proof of the right to access the named object (possibly in a restricted manner, as indicated in an access rights field). For this reason capabilities may not be directly constructed or modified by programs. There are several ways of protecting capabilities, including tagging,[13,14] segregation,[15,16] partitioned segments[17] and passwords.[18]

The names in capabilities can either be used to look up the addresses of objects in some sort of central object table,[3,16,18] or they can be the actual virtual addresses of objects in a virtual memory.[19] In either case the name space of objects will eventually become very sparsely distributed and in the second case the system is left with the problem of mapping large sparsely distributed virtual addresses onto main memory and disc addresses.

An additional incentive for considering the problem of supporting large virtual addresses is that it also becomes possible to have large physical addresses and thus a very large main memory. It has been argued in the literature[20] that a large main memory (in the order of gigabytes) can be used to achieve supercomputer speed for certain applications on a relatively modest speed processor.

This paper considers the problem of mapping large sparsely distributed virtual addresses from both the hardware and software viewpoints. First we review conventional memory mapping techniques, showing that they are inappropriate. A new scheme is then discussed and reference is made to an implementation on the experimental MONADS-PC computer system.[21]

2. CONVENTIONAL TECHNIQUES

The most widely used form of virtual memory organisation is paging, which was first introduced on the Atlas computer.[1] In the Atlas scheme paged addresses were translated via an associative memory with the virtual page number as the key and one entry for each page of main memory. This was appropriate because the main memory size was quite small and the Atlas was not designed for multiprogramming so that virtual addresses were essentially unique. However, as main memory sizes grew it became infeasible to build large enough associative stores and a different technique based on page tables was adopted. A page table is a linear list, indexed by virtual page number, each entry of which describes the current status and location of the corresponding virtual page. The length of such a page table is thus proportional to the number of virtual pages. Usually there is one page table per process to support multiprogramming and to provide protection between processes. Pages may either be in main memory, in which case a main memory page frame number is held in the entry, or on disc, in which case the disk address of the page is held. Often page table entries contain access information (e.g. read/write/execute bits) and information used by the page discard algorithm (e.g. use and modify bits). The page tables themselves are usually held in main (or virtual) memory.

The scheme as described in principle requires an extra memory access (to read the corresponding page table entry) on each memory operation. In order to ensure acceptable performance a high-speed translation lookaside buffer is usually used to hold the most recently accessed address translation entries. These translation lookaside buffers are usually implemented as set associative caches.[22]

If the size of the virtual memory is large (e.g. as a result of supporting a persistent store or memory-mapped files) then it may not be possible (or economical) to hold all of the page table entries in main memory. In this case the page tables themselves must be placed in virtual memory. This implies that there must be page tables for the page tables, which may either be locked down in main memory (if the tables are small enough) or placed in virtual memory and the same structure nested until the page tables are small enough to be locked down. For example, the VAX-11, which has 32-bit virtual addresses and 512-byte pages, has a two-level structure in which the page tables of page tables are locked into main memory. Paging of page tables adds complexity to both the page fault handler and the address translation buffer update mechanism, because an address translation buffer miss or even a page fault can occur when trying to address the page tables.

In the scheme described above the page tables serve two purposes. First, in the case that the required page is not currently in main memory, the appropriate entry indicates its current disc address. This information, which is set up when the disc space is allocated for the page, is relatively static in most systems. Second, for those pages which are currently in main memory, the page table entry holds the main memory address. (Usually a small table, indexed on main memory page number, is used to hold the disc addresses of pages currently in main memory.)

This scheme works well when the virtual address size is relatively small. For example, on the VAX-11 with an address space of 2^{32} bytes and a page size of 2^9 bytes the number of page table entries is 2^{23}. Each of these entries is 2^2 bytes long and thus the size of the page table is 2^{25} bytes or 2^{16} pages. The page table for this page table therefore requires 2^{16} entries and is 2^{18} bytes long, and

thus can feasibly be locked down in main memory. However, if we take the example of 128-bit addresses (which is not entirely unreasonable for a capability-based system) approximately 15 levels of page tables are required before a page table of similar length is obtained. Even if the page size is increased conventional address translation techniques are inappropriate.

A further problem with the conventional page table approach is that unlike the disc addresses, the main memory addresses change whenever a page is removed or brought into memory, requiring the page table to be modified to reflect the new state. This may in turn lead to a (further) page fault, if the relevant part of the page table is not currently in main memory. In order to resolve this page fault another page may have to be discarded, and this may in turn cause further page faults. This process may be nested to several levels, thus complicating the page fault handler.

The fundamental problem with using the conventional page table approach for large virtual addresses is that the length of the page tables is proportional to the size of the virtual memory. This problem arises not only with simple paging schemes but also with segmented or segmented and paged virtual memories which rely on a single table containing the mapping information required both for logical name to main memory address translation and logical name to disc address translation. Similarly it is one of the problems with central object table implementations in capability based computers.[19]

3. TRANSLATING LARGE VIRTUAL ADDRESSES

The key to providing an effective address translation mechanism for very large address spaces is to divorce the mapping of virtual to main memory addresses from the virtual to disc address translation. The former is required for every memory reference while the latter is only required to resolve page faults. By separating the two issues different mechanisms and structures may be applied to the relatively static disc address information and to the volatile main memory address information.

The scheme proposed in this paper and implemented in the MONADS-PC computer uses special purpose hardware for translating the virtual addresses generated by programs. Unlike conventional lookaside buffers, which can only hold a subset of the address translation entries, the proposed address translator holds entries for all the main memory page frames. Consequently a miss is only generated if a page is not resident in main memory, and page tables are not needed at all for the translation of virtual addresses to main memory addresses. An important advantage of this scheme is that page table entries need not be modified when a page is discarded, thus removing the possibility of nested page faults due to updating the page table as described in the last section. If a page fault is signalled then a separate mechanism, with much less critical performance requirements (in view of the much slower access speed of discs and the relative infrequency of the operation compared with main memory accesses), can be used to find the location of the page on disc.

3.1. Address translation hardware

The proposed scheme uses a hash table with embedded overflow to resolve synonyms. Hash tables have been used on other machines to support address translation. The MU6-G[23] used multiple hash tables searched in parallel and the IBM System/38[3] used a single hash table held in main memory. Both of these machines also had a dedicated address translation cache to improve performance. In our scheme the hash table is held in dedicated high speed memory. Each cell of the table contains a key field identifying the virtual page, a main memory page frame number, a link field and various status bits (see Fig. 1).

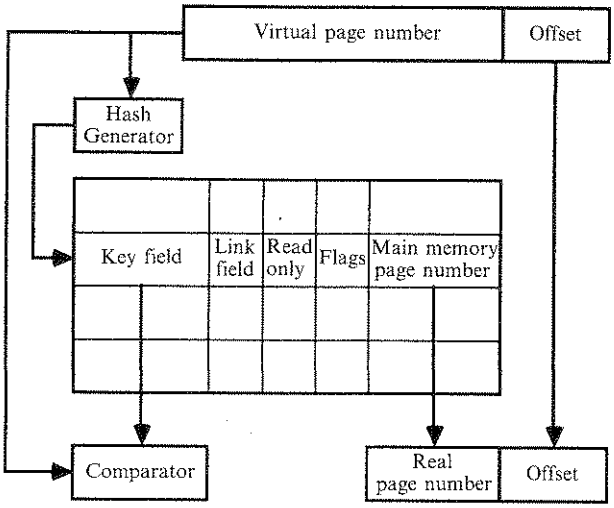

Figure 1. Address translation hardware.

When presented with a virtual address the hardware hashes the address to obtain a cell address within the table. The key field in the cell is compared with the original virtual address, and if there is a match, the main memory page frame number is used to form a main memory address. If there is a mismatch, the link field is used to follow a chain of synonyms. The chain is terminated by an end of chain status bit. If a virtual page is not found in the hash table then a page fault is generated.

Insertion and deletion operations are only performed when a page is loaded into or discarded from main memory. The operations are not particularly complex and are implemented in microcode or in the kernel of the operating system. Because addresses have a system-wide validity the contents of the hash table are not affected by context switches, unlike the MU6-G which appended the process number in order to ensure uniqueness of virtual addresses across processes.

Given that the address translator must be able to translate all virtual addresses of pages in main memory, the table must be at least as large as the number of page frames. In order to keep the number of synonyms small the table must be considerably larger. Assuming a random uniform distribution of cell addresses it can be shown that the average search length is given by the formula:

$$1+(\alpha/2),$$

where α is the loading factor, i.e. the ratio of full cells to total size.[24] For example, with a loading factor of 0.25 (corresponding to a table with a number of entries equivalent to four times the number of pages of main memory), the average search length is 1.125. A significant

feature of this scheme is that the length of the table (and therefore the cost of an implementation) increases linearly with the size of main memory. Furthermore the table width increases only logarithmically with the size of the virtual memory. Therefore increasing from a conventional virtual address size of 32 bits to a very large virtual memory with 128-bit addresses does not significantly increase the cost.

The above discussion has assumed a random uniform distribution of cell addresses. It is unlikely that virtual addresses will be uniformly distributed and therefore they must be hashed to form a cell address. The hashing function is performed on every address translation and thus must be simple. Such a simple function may be implemented by performing an exclusive OR of selected bits.

If the virtual address is found in the head of a synonym chain then the address translation time is the same as on a conventional lookaside buffer. The fetching of entries in a chain may be overlapped with the key comparison, minimising the overheads of following a chain. The address translation time is relatively insensitive to the size of both virtual and main memory addresses.

The MONADS-PC implementation supports 60-bit virtual addresses which are translated into 23-bit main memory addresses. Alternative address mapping techniques with similar properties are described in references 25 and 26.

3.2. Page fault resolution

The hardware address translator described in the last section translates virtual addresses for pages which are present in main memory. If a virtual page is not in memory, then it must be obtained from secondary storage.

In a persistent system, a virtual address can refer to any byte on any disc connected to the machine. A first step in locating the page on disc is to determine the (logical or physical) disc number. A simple but feasible scheme is to use a selected number of high order virtual address bits to indicate the disc number. With a suitably large virtual address size (e.g. 128 bits) this need not limit the number or size of discs, and it avoids having a complex mapping structure for maintaining the association between virtual addresses and disc numbers.

It is convenient to divide the virtual address space of a disc into areas roughly corresponding to logical entities (e.g. files, programs) existing on the disc. We shall call these areas address spaces, identified by address space numbers. Each address space is divided into fixed size pages. Thus a virtual address can be viewed as consisting of four parts. The MONADS-PC virtual address structure is shown in Fig. 2.

Disc no.	Address space no.	Page no.	Offset

Figure 2. A virtual address.

A result of fixed partitioning of addresses is that the virtual address space becomes fragmented. However, no real storage is lost or wasted, all that is discarded are ranges of virtual addresses. Since virtual addresses are deliberately very large (so that even at the maximum rate of usage they will not be exhausted during the life of the system) and not reused this is not a serious problem. A major advantage of fixed partitioning is that it allows an object within an address space to expand to the maximum size without reorganisation of addresses. A potential disadvantage of the use of large addresses is that the size of code and data may be substantially increased due to embedded addresses. However, the effects of this can be greatly reduced by implementing an addressing architecture on top of the virtual memory which mainly uses local (within address space) addresses. Such an architecture is fully described in Reference 27.

Each address space has a separate page table of disc addresses for its pages. In contrast with conventional virtual memory schemes the address translation hardware does not access these page tables, and consequently they do not need to have a single permanently fixed format, but can have varying formats determined by the virtual memory software. Thus if pages are placed in contiguous disc blocks for example, a page table might consist simply of a start address on disc and the number of pages. Such an economic page table cannot be achieved with conventional techniques. In the case where disc block allocation is completely dynamic a separate entry is required for each page; the size of such a page table is dependent on the size of the address space and the page size. In the MONADS-PC computer the maximum length of an address space is 256M bytes (i.e. 2^{28} bytes) and the page size is 4K bytes (i.e. 2^{12} bytes). This requires a page table length of 2^{16} entries, which must be placed in virtual memory.

Unlike conventional page table entries, the entries in these tables do not need additional status bits (presence bit, access rights, use and modify bits, etc.). It therefore becomes possible to restrict the size of an entry to a 16-bit relative disc block number, thus allowing a maximum logical disc size of 2^{16} blocks each of page size 2^{12}, i.e. 256M bytes (the maximum size of an address space). (The mapping of several logical discs to a single physical disc, which requires only a trivial extension to the scheme, ensures that larger physical discs can be used.) With 16-bit page table entries the maximum size of the page table for an address space is 2^{17} bytes.

In order to maintain the disc addresses of the pages of such a page table, which we shall designate the primary page table of an address space, a secondary page table is required. Even for a full-size address space this secondary page table is small (64 bytes). Both the primary and the secondary page table must be addressable in the virtual memory and the latter must be rapidly locatable by the page fault handler. This can be conveniently achieved by placing the secondary page table at a pre-defined location in the address space itself (allowing the page-fault handler to form its address directly from the virtual address which caused the original page fault). It should be noted, however, that there is no requirement for the secondary page table to be locked down in main memory.

In the MONADS-PC computer the secondary page table is in fact placed in the first page of the address space. However, because the primary page table is potentially large and of variable length it cannot sensibly be placed at the beginning of the address space. In the MONADS-PC it is therefore placed at the end of the address space and grows backwards towards the data.

Many of the address spaces of a system, corresponding to small files and programs, may contain only a small

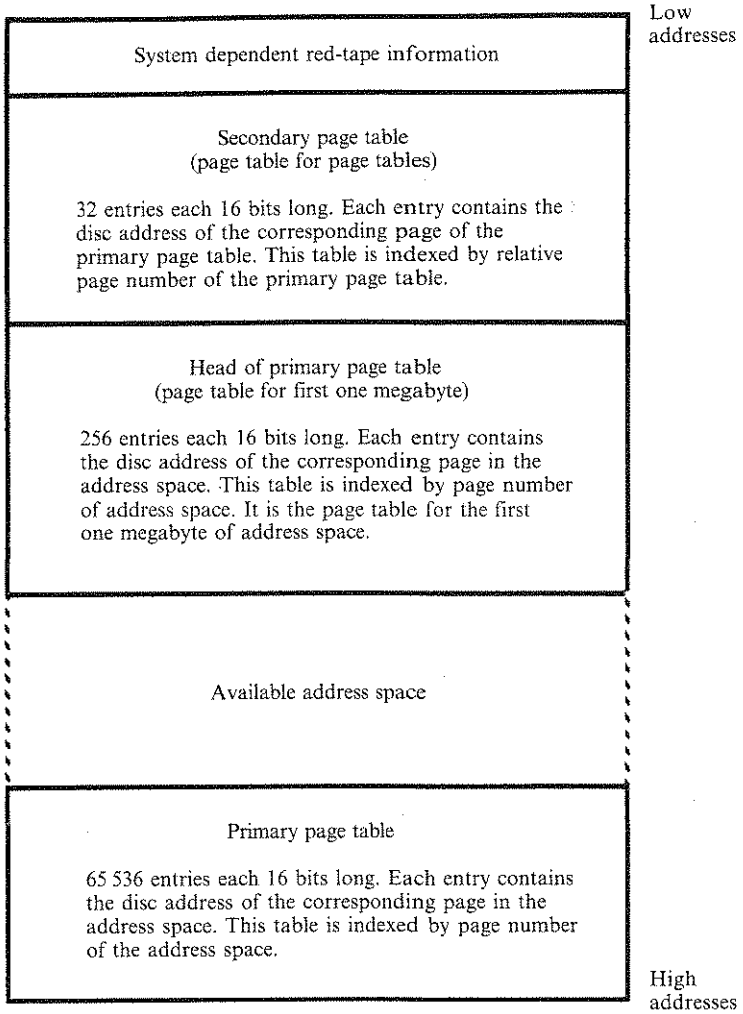

Figure 3. Address space structure.

amount of data. To reduce the number of disc accesses needed to resolve page faults in such cases, the MONADS-PC page handler places the first 256 entries of the primary page table alongside the secondary page table in the first page of the address space. Thus for an address space shorter than one megabyte (i.e. 256 4K byte pages) all the paging information is in the first page. Indeed, for a very small address space with less than about 3.6K bytes of data, the data is held in the same page as the paging information, so that the page fault is automatically resolved in the course of accessing the paging information. This scheme, as used in the MONADS-PC system, is illustrated in Fig. 3.

Because the primary page table is stored at known locations in an address space, and the virtual address of each entry can be calculated by the page fault manager, the scheme actually operates as follows. When a page fault occurs, the primary table entry for the page is read. If this causes a further page fault (i.e. the required page of the primary page table is not in main memory), the secondary page table entry is read. At this stage it is also possible that a further page fault can occur, in which case a separate mechanism must be used to find the disc address of the page containing the secondary page table, i.e. page zero of the address space. In the MONADS-PC system, address space zero on each disc serves as a disc directory, containing an entry indicating the disc address of page zero of each valid address space on the disc. Because address space numbers are sparsely distributed and large, a hash table provides an appropriate lookup mechanism for the disc directory. The actual organisation of the disc directory in the MONADS-PC system is illustrated in Fig. 4.

Address space zero of a disc (i.e. its directory of address

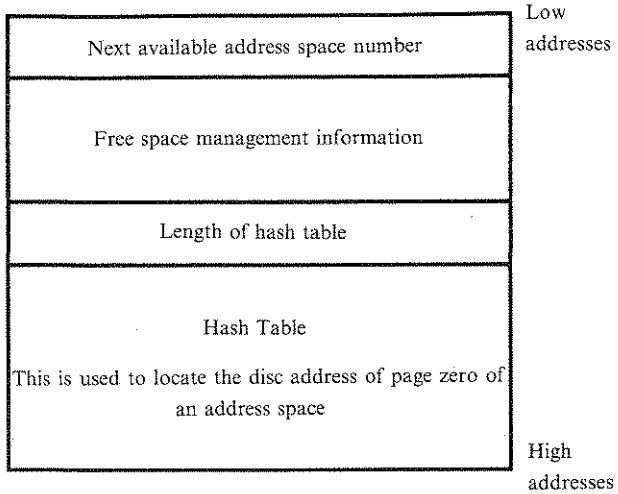

Figure 4. Disc directory structure.

spaces) is not treated in a special manner and has a primary and secondary page table in a similar fashion to other address spaces. However, it is necessary to place page zero of address space zero in a well known location so that the secondary page table of address space zero may be located. (The MONADS-PC kernel automatically reads and locks into memory page zero of address space zero for each mounted disc.)

This structure supports non-contiguous allocation of disc space in fixed size blocks in the same manner as most modern file systems. Consequently free space management is no more complex than in conventional systems and the usual techniques (e.g. a linked free list or a free space bit map) can be applied. Address space zero is a convenient place to hold free space information (Fig. 4).

Although the disc number is embedded in a virtual address there is no requirement that the number identifies a physical drive. The kernel must maintain a mapping between each removable disc and the drive on which it is mounted. In addition physical discs may be divided into a number of logical discs, each having its own disc number. Such a mapping scheme allows considerable flexibility in the organisation and allocation of disc space.

4. CONCLUSION

This paper has demonstrated that attempts to scale up conventional virtual memory techniques to support very large virtual addresses result in inefficiencies and clumsiness. New mechanisms are required. The techniques described in this paper have been implemented in the MONADS-PC system and are able to translate 60-bit virtual addresses into 23-bit main memory addresses in a comparable time to conventional address translation hardware. The cost of increasing the virtual address size, to say 128 bits, would be insignificant in terms both of the amount of hardware and of execution speed. Similarly the software structures are largely unaffected by any change in the virtual address size.

As was mentioned earlier, an interesting side effect of having large virtual addresses is that it becomes possible to have very large physical addresses and thus a large main memory. The MONADS project is currently investigating the development of a massive memory machine based on an extended version of the above architecture.[28] The prototype implementation, known as MONADS-MM, is based on the Sun SPARC processor with extended addressing hardware and has 128 bit addresses and support for up to 64 gigabytes of main memory.[35]

The discussion has been limited to the basic page management level of a uniform virtual memory. It is possible, to superimpose a more structured view of the memory onto this scheme. In the MONADS-PC system each address space (corresponding to a file, a code module, a stack or a local heap) is organised as a collection of segments addressed via capabilities, in suc a way that both large and small segments can b efficiently supported without internal memory fragme tation.[26,29] At run-time capabilities are loaded into fa registers, equivalent to base registers in convention systems. Thus the entire address decoding and translatic time is comparable with conventional virtual memo systems.

There are a number of unresolved issues which requi further research. The paper has not discussed the high level management of the virtual memory, in particula recovery after a system crash. Major inconsistencies m arise between the volatile main memory image of da and the permanent disc copy. This is a similar problem that encountered in conventional database manageme systems and it is possible to employ existing solutio such as shadow paging[30,31] and page time-stamps.[32] Su a scheme for the MONADS-PC system is described Ref. 36.

Another management issue is the page discard alg rithm. It is not clear that conventional algorithms (e LRU) apply where there is a mix of many different typ of data with varying access patterns. In particular should be possible to take advantage of known acc patterns for permanent data (e.g. sequential access) a to pre-page the required data. This requires furth analysis.

An advantage of a uniform virtual memory and t proposed address translation mechanism is that it m be extended to encompass a network of machines sharing the one uniform addressing scheme.[33,37] This similar to the distributed shared virtual memory d cribed by Ki Li.[38] The extensions are relatively sim and basically involve extending the virtual address form shown in Fig. 2 to include a node number. Usually node number indicates the physical machine on wh this address space resides. On a page fault the virt memory manager checks to see if the required page located on a disk connected to this machine. If not, th a request is sent over the network to the machine hold the page and the page is transmitted to the request machine. A coherency algorithm (similar to that e ployed with multiple caches) is used to guarantee integrity of the data. By dynamically maintaining appropriate set of tables it is also possible to allow b discs and individual address spaces to be moved betw machines invisibly[34,39,40] and to provide resilience. S a network architecture removes many of the inc sistencies and complexities found in conventic networks.

Acknowledgements

The authors wish to thank the PISA team, particula Professor Ron Morrison, for reading many earlier dr of this paper. Their constructive comments a suggestions have resulted in substantial improvement the paper.

REFERENCES

1. T. Kilburn, D. B. E. Edwards, M. J. Lanigan and F. H. Sumner, One level storage system. *IRE Transactions on Electronic Computation* **EC-11** (2), 223–234 (1962).
2. E. I. Organick, The MULTICS system: an examination its structure, MIT Press, Cambridge, Mass. and Lond (1972).

3. V. Berstis, C. D. Truxal and J. G. Ranweiler, System/38 addressing and authorization. *IBM System/38 Technical Developments*, pp. 51–54 (1978).
4. M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm and R. Morrison, An approach to persistent programming. *Computer Journal* **26** (4), 360–365.
5. M. P. Atkinson, P. J. Bailey, W. P. Cockshott and R. Morrison, PS-Algol Reference Manual. Universities of Glasgow and St Andrews, Report PPRR-12.
6. A. Dearle, Q. Cutts and G. Kirby, Browsing, grazing and nibbling persistent data structures. *Proceedings 3rd International Workshop on Persistent Object Systems, Newcastle, Australia*, pp. 96–113 (1989).
7. A. J. Hurst and A. S. M. Sajeev, A capability-based language for persistent programming: implementation issues. *Proceedings 3rd International Workshop on Persistent Object Systems, Newcastle, Australia*, pp. 186–201 (1989).
8. R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle and M. P. Atkinson, The Napier type system. *Proceedings 3rd International Workshop on Persistent Object Systems, Newcastle, Australia*, pp. 253–270 (1989).
9. A. Brown, Persistent Object Stores. Universities of Glasgow and St Andrews PPRR-71 (March 1989).
10. M. Shapiro and L. Mosseri, A simple object storage system. *Proceedings of the 3rd International Workshop on Persistent Object Systems, Newcastle, Australia*, pp. 320–327 (1989).
11. E. I. Organick, *Computer Systems Organisation, the B5700/6700 Series*, Academic Press, New York (1973).
12. J. B. Dennis and E. C. Van Horn, Programming semantics for multiprogrammed computations. *Comm. ACM* **9** (3), 143–145 (1966).
13. E. A. Feustal, On the advantages of tagged architecture. *IEEE Transactions on Computers* **C-22** (7), 644–656 (1973).
14. G. J. Myers and B. R. S. Buckingham, A hardware implementation of capability-based addressing. *Operating Systems Review* **14** (4) (1980).
15. W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp: an Experimental Computer System*. McGraw-Hill, New York (1981)
16. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operation System*. Elsevier North Holland, Inc. (1979).
17. A. Jones, Capability architecture revisited. *Operating System Review* **14** (3) (1980).
18. M. Anderson, R. D. Pose and C. S. Wallace, A password-capability system. *Computer Journal* **23** (1), 1–8 (1986).
19. J. L. Keedy, An implementation of capabilities without a central mapping table. *Proceedings 17th Annual Hawaii International Conference on System Sciences*, pp. 180–185 (1984).
20. H. Garcia-Molina, A. Park and L. Rogers, Performance through memory. *Proceedings SIGMETRICS Conference*, pp. 122–131 (May 1987).
21. J. Rosenberg and D. A. Abramson, A capability-based workstation to support software engineering. *Proceedings 18th Annual Hawaii International Conference on System Sciences*, pp. 222–230 (1985).
22. A. J. Smith, Cache memories. *ACM Computing Surveys* **14** (3), 473–530 (1982).
23. D. B. E. Edwards, A. E. Knowles and J. V. Woods, MU6-G: a new design to achieve mainframe performance from a mini-sized computer. *Proceedings of the 7th Annual Symposium on Computer Architecture. Computer Architecture News* **8** (3), pp. 161–167 (May 1980).
24. R. Morris, Scatter storage techniques. *Comms ACM*, pp. 38–43 (January, 1968).
25. S. S. Thakkar and A. E. Knowles, Virtual address translation using parallel hashing hardware. *Proceedings Supercomputing Systems Conference*, pp. 697–705. IEEE, Florida (1985).
26. K. Ramamohanarao and R. Sacks-Davis, Hardware address translation for machines with a large virtual memory. *Information Processing Letters* **13** (1), 23–29 (1981).
27. J. Rosenberg and J. L. Keedy, Object management and addressing in the MONADS architecture. *Proceedings of 2nd International Workshop on Persistent Objects Systems*, Appin, Scotland (1987), available as PPRR-44, Universities of Glasgow and St Andrews.
28. J. Rosenberg, D. M. Koch and J. L. Keedy, A massive memory supercomputer. *Proceedings of 22nd Annual Hawaii International Conference on System Sciences*, pp. 338–345 (1989).
29. J. L. Keedy, Paging and small segments: a memory management model. *Proceedings 8th World Computer Congress (IFIP-80), Melbourne*, pp. 337–342 (1980).
30. D. Chamberlin *et al.* A history and evaluation of system R. *Comms ACM* **24** (10), 632–646 (1981).
31. R. A. Lorrie, Physical integrity in a large segmented database. *ACM Transactions on Database Systems* **2** (1), pp. 91–104 (1977).
32. S. M. Thatte, Persistent memory. *Proceedings of IEEE Workshop on Object-Oriented DBMS*, pp. 148–159 (1986).
33. D. A. Abramson and J. L. Keedy, Implementing a large virtual memory in a distributed computing system. *Proceedings 18th Annual Hawaii International Conference on System Sciences*, pp. 515–522 (1985).
34. P. Broessler, F. A. Henskens, J. L. Keedy and J. Rosenberg, Addressing objects in a very large distributed virtual memory. *Proceedings of IFIP Amsterdam Conference on Distributed Systems*, pp. 105–116 (Amsterdam, 1987).
35. D. M. Koch, and J. Rosenberg, A secure RISC-based architecture supporting data persistence. *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Bremen, West Germany, May 1990*, pp. 188–201, Workshops in Computing series, Springer, Heidelberg (1990).
36. J. Rosenberg, F. A. Henskens, A. L. Brown, D. Munro and R. Morrison, Stability in a persistent store based on a large virtual memory. *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Bremen, West Germany, May 1990*, pp. 229–245, Workshops in Computing Series, Springer, Heidelberg (1990).
37. F. A. Henskens, A capability-based persistent distributed shared memory, *Ph.D. Thesis*, University of Newcastle, Australia (1991).
38. K. Li, Shared virtual memory on loosely coupled multiprocessors, *Ph.D. Thesis*, Yale University (1986).
39. F. A. Henskens, Addressing moved modules in a capability-based distributed shared memory. *Proceedings 25th Hawaii International Conference on System Sciences, January 1992*, pp. 769–778 (1992).
40. F. A. Henskens, J. Rosenberg and M. R. Hannaford, Stability in a network of MONADS-PC computers. *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Bremen, West Germany, May 1990*, pp. 246–258, Workshops in Computing Series, Springer, Heidelberg (1990).

Constructing Programs as Executable Attribute Grammars

R. A. FROST
School of Computer Science, University of Windsor, Windsor, Ontario N9B 3P4, Canada

Attribute grammars provide a formal yet intuitive notation for specifying the static semantics of programming languages and consequently have been used in various compiler generation systems. Their use, however, need not be limited to this. With a little change in perspective, many programs may be regarded as interpreters and constructed as executable attribute grammars. The major advantage is that the resulting modular declarative structure facilitates various aspects of the software development process.

In this paper, we show how the attribute grammar programming paradigm can be readily supported by adding four combinators to the standard environment of a lazy functional programming language. We give examples of the use of these combinators and discuss the advantages that derive from integration of the attribute grammar and functional programming paradigms.

Received April 1989, revised December 1991

1. INTRODUCTION

Attribute grammars were introduced by Knuth[20,21] in 1968 as a notation for specifying the static semantics of programming languages. In 1971 Knuth[22] suggested that the attribute grammar formalism might lead to viable declarative programming languages, in which problems are solved in terms of relevant structures.

Since their introduction, a good deal of theory has been developed[7,12] and attribute grammars have been used in various language-processor generation systems.[6,8] However, use of the attribute grammar formalism as a general-purpose programming paradigm has received attention from only a few researchers.[9,14,17,19,30,32]

In this paper we show how the attribute grammar programming paradigm can be readily provided by adding four combinators (higher-order functions) to the standard environment of a lazy, functional programming language. We present a number of examples of the use of these combinators to illustrate various features of the style of programming that they support.

We discuss the advantages and disadvantages of a functional implementation of the attribute grammar paradigm. In particular, we consider the advantages that derive from lazy evaluation and from the higher-order nature of the host language.

We conclude with a brief overview of related work and suggestions for future research.

2. THE ATTRIBUTE GRAMMAR FORMALISM AND ITS USE IN SOFTWARE ENGINEERING

An *attribute grammar* is a context-free grammar, each production of which is augmented with a set of *semantic rules*. Each semantic rule states how the value of an *attribute* associated with a syntactic construct in the production is derived by applying a *semantic function* to values of attributes associated with other syntactic constructs in the production.

The set of attributes associated with a particular syntactic construct can be partitioned into two disjoint sets: *synthesised* attributes and *inherited* attributes. Each semantic rule associated with a production rule P either defines a synthesised attribute of the syntactic construct named on the left-hand side (lhs) of P or defines an inherited attribute of a syntactic construct on the right-hand side (rhs) of P. Synthesised attributes may be regarded as passing semantic data upwards towards the root of the derivation tree. Inherited attributes may be regarded as passing semantic data down the derivation tree.

An example of a simple attribute grammar, involving only synthesised attributes, is given in Fig. 1.

```
numb    ::= "one"
              VAL↑numb = VAL 1
          | etc

summ    ::= numb
              VAL↑summ = VAL↑numb
          | numb "plus" summ'
              VAL↑summ = VAL↑numb + VAL↑summ'

subtr   ::= numb "minus" numb'
              VAL↑subtr = VAL↑numb - VAL↑numb'

comp    ::=   subtr | summ

br_comp ::= "(" comp ")"
              VAL↑br_comp = VAL↑comp

expr    ::= br_comp
              VAL↑expr = VAL↑br_comp
          | "minus" br_comp
              VAL↑expr = - VAL↑br_comp
```

Figure 1. A simple attribute grammar.

- The bold text constitutes a context-free grammar for a simple language of expressions. The notation used is a variant of Backus–Naur form, in which terminal symbols appear inside double quotes.
- An upward arrow signifies that the attribute is synthesised. For example, `VAL↑expr` should be read as 'the `VAL` attribute that is synthesised for the expression'.
- Each semantic rule indicates how the value of an attribute associated with the syntactic construct on the lhs of a production is obtained from attributes of