# Issues in Implementing Virtual Memory

Kevin Elphinstone
Stephen Russell
Gernot Heiser
School of Computer Science and Engineering
University of NSW 2052 Australia

*E-mail:* kevine@vast.unsw.edu.au

UNSW-CSE-TR-9411 — 29 SEPTEMBER 1994

**Abstract**

Several factors are rapidly increasing the demands being placed on virtual memory implementations. Large address spaces, increasing sparseness, and novel operating systems are not well supported by traditional tree-based page tables. New approaches are needed to overcome these problems.

This paper examines the advantages and disadvantages of conventional virtual address translation schemes. It then describes the performance costs caused by recent changes in hardware and operating system architectures. While there is much active research directed towards reducing these costs, it is mostly intended to provide better support for Unix style systems. Many issues are still unresolved, particularly those relating to the support of the large, sparse address spaces used by single address space operating systems.

# 1 Introduction

Virtual memory (VM) has been in use for over thirty years [Den70] and paged VM is now used in almost all modern computer systems. Its popularity is easy to explain as it naturally supports concurrent processes with protected memory, reduces the amount of physical memory required for each process, and allows processes that require more memory than is physically available to be executed.

At first glance it would appear that implementation methods for VM are well established and little room is left for further investigation. However, many recent changes in operating system (OS) design and hardware architectures are forcing a reexamination of VM techniques.

The Mungi [HERV94] project is currently constructing an OS based on a shared single 64-bit address space. As part of our research, we have identified several important problems that are not adequately addressed by traditional approaches. Other new approaches to OS design, such as object-oriented systems [SGH+89, SAK+89], have highlighted similar problems.

This paper presents a review of current trends in VM implementation techniques and identifies the problems of traditional tree-based translation schemes. Section 4 examines recent alternatives that have been proposed to overcome these problems. As we argue in our conclusion, however, none of the existing approaches have so far met the demands of modern OS design.

# 2 Background

In the late 1980's microprocessors had remarkably similar memory management architectures[Mil90]. They all supported 32 bit paged virtual memory management via a *translation lookaside buffer* (TLB) and hierarchical page tables.

In conventional tree-based *page tables* (PT), a virtual address is broken into several parts. The low order bits of the address are used to index pages, while the high-order bits are used as a set of indices into a multi-level tree. Each virtual memory access traverses the tree to acquire the frame number of the page. To reduce the cost of performing translations, a translation lookaside buffer caches recently used PT entries.

Two or three level trees allow 32-bit virtual address spaces to be implemented with low overhead. They also support sharing of portions of virtual memory between processes by sharing tree nodes, simplify checkpointing, and allow performing operations such as changing protection by modifying only higher level tree entries.

Inverted PT and hashing schemes have also been used, but such approaches

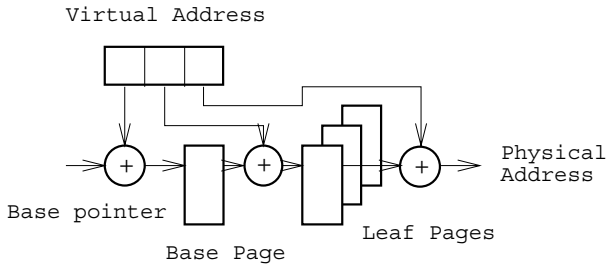


Figure 1: Hierarchical Page Table

have until recently not been very popular. They are, however, being re-examined in light of the demands of new OS designs and will be discussed further in Section 4.

# 3 Recent Developments

Recent progress in microprocessor design and operating system structure has changed the way virtual memory is used and supported. The simple structure of VM management described above is no longer efficient, or even adequate. The following is a description of recent advances in system architecture and how they effect traditional VM systems.

## 3.1 64 bit Processors

The advent of 64 bit microprocessors [Hei93, Sit92] has an immediate effect on the VM system. The need to translate a larger number of virtual address bits enlarges the silicon required for the TLB and affects the data structure used to store mappings between virtual and physical memory.

The methods used in traditional 32-bit systems result in a five-level PT tree for 64-bit address spaces. The additional levels increase the cost of performing translations in the case of TLB misses. In order to reduce the effect of these overheads, DEC suggest for the Alpha [Sit92] a three-level tree which is mapped in the kernel virtual address space. Nodes in this tree are large virtual structures but are only partially allocated.

The MIPS R4000 [Hei93] uses what is in effect a two-level tree which is also mapped into the kernel virtual address space as a large virtual array.[1] Each region

[1] The array is extremely large ($\approx 2^{52}$ entries) but *extremely* virtual.

of allocated virtual address space is assigned a number of physical frames to contain its PT mappings. Entries for each of these regions are kept in the TLB and are mapped so as to appear at the correct locations in the virtual array. A simple hardware mechanism is provided for quick indexing of the virtual array by the TLB refill handler.

These approaches work well for the UNIX model in which there is a small number of segments of allocated virtual addresses (code, stack and heap). However, address spaces are becoming sparser, for reasons that will be explained in more detail in Section 3.3. Multi-level tree approaches suffer from severe internal fragmentation in handling sparse address layouts. The extreme case is an address space layout that requires one page in each level in the page table tree for each page of user data. In this case, user data represents only 20% of memory used with a five-level tree. Even the virtual array approach has difficulties because of the need to use a TLB entry for each segment to prevent misses in the refill handler.

The approaches that have been tried by DEC and MIPS are adequate for traditional operating system models, but they are not suitable in the new environments that are made attractive by 64-bit addresses.

## 3.2 Large Physical Memories

The amount of physical memory installed in a modern workstation is increasing rapidly. Six years ago a typical workstation was shipped with 4 MB of RAM and was expandable to approximately 32 MB. Today's systems ship with 32 MB of RAM and expand to at least 128 MB, and 512 MB is not uncommon. The rate of increase is unlikely to slow as DRAM size quadruples every three years and memory requirements of programs increase 1.5 to 2 times every year [PH90].

The increase in memory requirements of applications has led to larger application working set sizes. Current TLB's have not kept pace with this increase in memory usage. The average microprocessor TLB size was 64 entries several years ago, with a 4 KB page size. This gave a TLB coverage of 256 KB, which represents a significant portion of the 4–32 MB of older systems.

Current generation microprocessors have approximately 80 TLB entries with a typical page size of 8 KB. TLB coverage in this case is 0.6 MB, which may be a reasonable proportion of base memory sizes of 32 MB, but is only 0.2% of 512 MB.

Inadequate TLB coverage causes more frequent TLB misses. Applications that spend 5–20% of their run time servicing TLB misses are now common, with extremes of 40% not unheard of [HH93].

Large physical memories also increase the size of the per-page translation entries in the page table. The number of bits required to index all physical frames is

now greater than the 26 or so[2] available in a 32 bit PT entry. The next logical step is to use 64 bit PT entries per page, which doubles the storage overhead per physical page.

In summary, the increase in physical memory size has revealed inadequacies in current TLB coverage, and increased the memory required to store translations.

## 3.3 Sparse Memory Usage

The memory layout of a UNIX process is well-known. It consists of the program text and data at the bottom of the available user address space. The heap used for dynamic memory allocation sits above the text and data segment and is free to grow upwards during program execution. The stack lies at the top of the user address space and is free to grow downwards.

This process memory layout effectively divides the address space into two contiguous regions which grow towards each other. This layout is efficiently supported by hierchical page tables for two reasons; contiguity means that the page table is densely populated and thus space efficient, and TLB misses on the page table itself can be minimised by reserving a few entries in the TLB to map the areas of the page table corresponding to the known text, heap and stack layout of a process.

The following subsections describe features of modern operating systems which result in virtual memory being used in ways that differ significantly from the UNIX model that the traditional VM system has supported. These features tend to consume more virtual address space and populate it more sparsely.

### 3.3.1 Memory Mapped Files

Many operating systems provide memory-mapped files, which map portions of files into the virtual address space of a process. This allows I/O operations to be performed as simple memory accesses rather than explicit system calls. While this feature is convenient to programmers, it increases virtual memory usage and results in higher TLB miss rates due to inadequate TLB coverage. Memory-mapped files are usually large and few in number, and so have relatively small impact on PT fragmentation. As well, the OS is free to map these files as it sees fit, and so they can be mapped contiguously. These factors reduce the impact of memory-mapped files on PT size and fragmentation.

[2] Page table entries may also contain *modify*, *reference*, and *valid* bits together with cacheability information and operating system specific bits.

### 3.3.2 Persistent Systems

Persistent systems [ABC+83] allow arbitrary data structures, including pointers, to be stored in a name-space that effectively replaces a file system.

Some persistent systems do not permanently allocate addresses to objects but simply map them into a process' address space at a suitable free region and then swizzle [Wil91] the pointers to compensate for the changed location. These systems have similar impact on address-space management as the memory-mapped files described above.

Other systems, like Monads [RA85], allocate a permanent address for each object. It is common in these systems to not re-use addresses to ensure uniqueness over time. This results in a sparse trail of currently allocated objects in a large virtual address space, whereas a reuse policy results in a much smaller virtual address range which is densely packed [Elp93, HERV94]. In either case, persistent systems lead to a large number of potentially quite small objects within a virtual address space. As has been argued earlier, current translation schemes cannot support such sparse small-grain address spaces efficiently.

### 3.3.3 Single Address Space OS

A particular class of persistent systems are single address space operating systems [HERV94, CLBHL92, MWO+93]. These use permanent address allocation and thus suffer from the virtual address space fragmentation described above.

One of the major properties of these systems is their separation of translation and protection. A single address space mapping is used for all processes, and so the translation table may have to contain mappings for all of the objects being referenced by the processes on a host. This leads to a drastic increase of the number of objects that must be managed, and further fragments the page table.

The protection models in these systems are generally based on lists of capability that are maintained for each process [VRH93]. The capabilities are used at run-time to establish mappings with the appropriate protection bits. As activity switches between processes it is necessary to quickly change the protection bits without affecting the translation. As well, there needs to be support for quickly changing the protection domain of a process for privileged operations, and to support object-oriented programming models.

The implementation of these features on current 64-bit architectures with software-loaded TLBs is problematic, because the TLB entries are intended for individual address spaces which combine translation and protection information. It is therefore necessary to modify the TLB refill handler to combine information from a global translation table with the appropriate process-specific protection information. The

question of appropriate data structures for these two tables is still unresolved.

### 3.4 Summary

The combination of large address spaces, large memories, and sparse memory usage is stretching the limits of conventional approaches to virtual address translation. In particular, we are reaching the limit of TLB performance, while at the same time having to manage the increasing size of translation tables. The traditional approach to VM also does not cope well with the separation of translation and protection needed by single address space systems.

## 4 Current Research

Current strategies for solving the problems identified above can be divided into two groups. The first strategy involves changes to the page table data structure to minimise TLB refill time and to cope with sparse address spaces and new protection systems. The second involves changes to the TLB to minimise the number of misses.

### 4.1 Page Table Structures

One approach to cope with large address spaces is to separate address translation from backing store management. The page table changes from one that maps all valid virtual addresses of a process to one that only needs to map those pages resident in memory. This allows a structure designed fo fast TLB reloads to be used, but requires a separate data structure for finding the backing store address of non-resident pages. An obvious approach to fast lookup is to use hashing to search a table of resident pages. Two basic techniques have been used: *inverted* page tables and directly *hashed* page tables.

#### 4.1.1 Inverted Page Tables

Inverted page tables (IPT) are characteristic of large address space architectures such as the System/38 [IBM78], the 801 [CM88] and the HP Precision Architecture [Lee89]. These architectures used *short form* addresses together with address space registers to generate *long form* addresses of 40 to 64 bits in length. Translation of these long form addresses was the motivation for using hashing techniques.

An IPT consists of two parts (see Fig. 2). The *frame table* contains an entry for each physical frame, which contains the number of the virtual page occupying that frame and associated protection information. A separate *hash table* is used to map page numbers to the corresponding entries in the frame table.

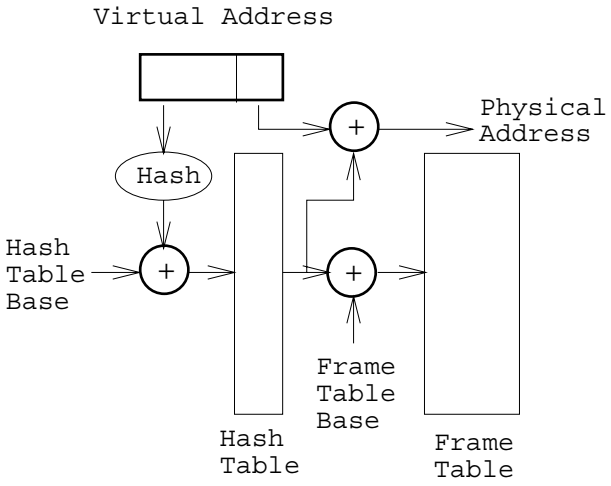


Figure 2: Inverted Page Table

The advantage of IPTs is that their size scales linearly with physical memory size. IPT size is almost independent[3] of virtual address space size and more importantly, independent of the distribution of occupied virtual address space.

A disadvantage of IPTs is that they are heavy weight structures compared to hiearchical page tables for those processes requiring only a small amount of memory. Systems using IPTs overcome this by using one global IPT and sharing it between all processes.

A global IPT limits the system to one virtual to physical translation per physical page at any instant in time. Sharing is achieved via a global address, not aliasing. If aliasing is required for other reasons, or sharing is required with different protection levels, then IPT entries need to be changed on context switches. This makes them less attractive for single address space operating systems becaause all entries need to be changed to alter protection domains.

IPTs do not support sub-block TLBs, which will be described in Section 4.2.2. For example, the R4000 uses a sub-blocking TLB which supports 8 KB page TLB blocks containing two 4 KB page physical frame entries. To refill this using an IPT would require either a lookup for each 4 KB page, or require 4 KB pages be consecutive in memory forming 8 KB blocks. The former method is too slow, and the latter effectively increases the page size to 8 KB, removing the advantages of sub-

[3] Inverted page tables actually scale logarithmically with virtual address size. This is relatively insignificant compared to the linear relationship with physical memory.

blocking the TLB.

The IPT is indexed by physical frame number. This requires physical memory to be contiguous, as holes in the physical memory layout require empty entries in the page table in order to preserve the indexing. This can be expensive if memory mapped I/O devices cannot be packed efficiently.

### 4.1.2 Hashed Page Table

The hashed page table (HPT) [HH93] has been proposed to overcome some of the limitations of the IPT. It is a more general variant of the hardware technique for TLB replacement in the MONADS-PC [RKA92].

In a HPT (see Fig. 3) only a single table is used. The entries contain page number and frame number and protection information, and are indexed using hashing. Collisions are resolved using methods such as chaining.

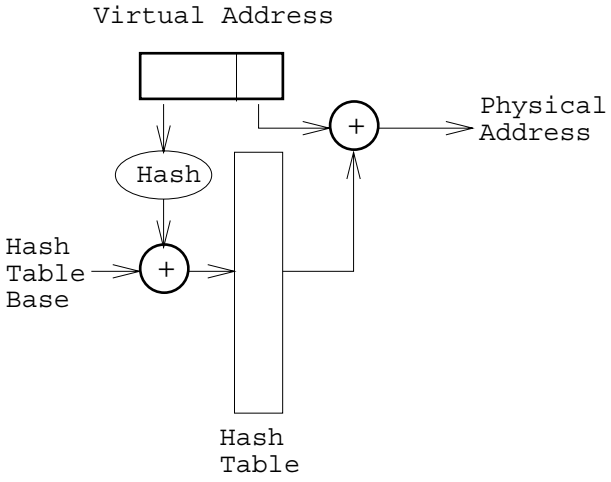


Figure 3: Hashed Page Table

The HPT has the same primary advantage as the IPT; its size is independent of virtual address space considerations. Like the IPT, the HPT is also a heavy weight data structure more suitable for global use than as a per process structure.

Aliasing is supported by a HPT. Since the hash structure contains the physical frame number, it is simple to insert extra hash entries for different virtual addresses that reference the same frame number. Alias entries consume otherwise empty space and create multiple dirty and reference bits so global addresses would

still be the preferred way to share data.

HPTs are suitable for sub-block TLBs. In the case of the R4000, the hash can be performed on the 8 KB block locating an entry containing two 4 KB physical frame numbers. No restriction is placed on the frame numbers so the advantages of sub-blocking remain with only one lookup required.

The HPT is unaffected by the physical address space layout. The structure is not indexed in any way by physical frame number. This removes the need for unusable entries corresponding to holes in the physical address space.

The HPT is better suited to software implementation than the IPT. This is important due to the current predominance of software loaded TLBs. A software IPT needs to hash the faulting address to get the index from the hash table into the page table, then, assuming a match, shift the index into a form suitable for the TLB and combine it with the protection bits in the page table. Frame numbers in the HPT can be stored together with protection bits in the exact format required for TLB refill. TLB refill simply becomes a matter of hashing the faulting address to find the TLB entry and loading it into the TLB.

The HPT is demonstrably faster than the IPT [HH93]. This is mainly due to the IPT requiring access to two tables with two possible cache misses. The HPT only accesses one table with both the virtual address and frame number lying in the same cache line. This reduces potential cache misses to at most one.

### 4.1.3 Guarded Page Tables

The main drawback of the previous two approaches is that they do not support multiple page sizes, since the indexing of the table is done by applying a hash function to a fixed part of the virtual address. As a result, it is relatively expensive to perform operations such as modifying protection, or inserting or deleting objects, because they need to be done frame by frame.

An alternative approach has been proposed by Liedtke [Lie94]. It is a tree-based scheme called *guarded page tables* (GPT) that combines the advantages of relatively fast translation table updates for variable size objects without suffering from excessive fragmentation as a result of sparse allocation policies.

A GPT is similar to a conventional tree-based page table, except each page table entry is supplemented with a variable length bit string termed a *guard* (Fig. 4).

The topmost level of the page table is indexed using the upper bits of the virtual address. The guard is then compared with the most significant bits remaining in the virtual address. If these bits match, the pointer contained in the entry is followed to the next level, where the remaining virtual address less the matching bits is used as an index to repeat the process. This continues until all virtual address bits are exhausted, in which case the entry contains the translation information required. A

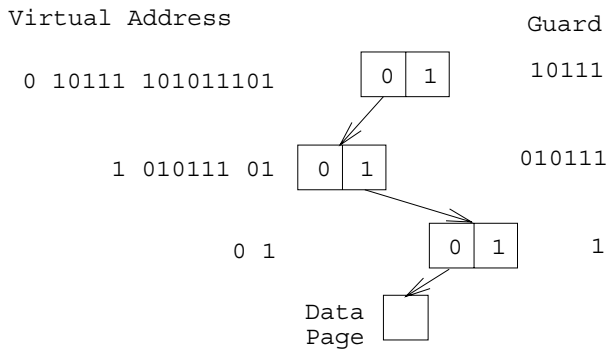


Figure 4: Guarded Page Table

traditional page table tree can be considered to be a GPT with empty guards. A more detailed description of the translation process can be found in [Lie94].

It can be shown that at most $2k$ entries are need to map $k$ pages. Page table size is thus a function of the number of mapped pages, and is independent of address space size and layout. The scheme also has the advantage of supporting multiple page sizes from 16 bytes upwards in powers of 2. Being a tree, the page table retains all the advantages of conventional page tables as described previously in section 2.

This scheme is yet to be implemented, so it is unclear whether performance claims of being at least equivalent to conventional trees will eventuate. The lookup process is quite complex, involving variable length bit string manipulation which may turn out too costly for software implementations. However, the increase in TLB coverage offered by variable size pages may significantly reduce the number of TLB refills which may offsett the reload costs. Further investigation is definitely warranted as the scheme has many advantages if performance proves satisfactory.

## 4.2 TLB usage

Changes to TLB usage aimed at improving performance can be separated into software techniques to manage the TLB more effectively, and hardware changes to minimise misses and increase coverage.

### 4.2.1 Software

TLB management improvements can be divided into optimising replacement and placement of TLB entries, and minimising refill times. The likely improvements

in performance that can be achieved by replacement policy are minor [UNS+94, CBJ92]. Hardware designers typically choose random replacement, as the cost involved in implementing other policies in hardware outweigh the performance benefits to be gained. Performance improvements through software involvement in replacement policy are doubtful.

The costs of software misses are a tradeoff between speed and memory consumption. A sparsely occupied virtual page table array provides the fastest lookup for refill. Mapping the occupied portions of the array will have a detrimental impact on TLB performance if many objects need to be mapped. Refill times using hashed page tables are slightly higher, but this method is better suited to large or sparse address spaces [HH93].

Software control of placement policy does have an effect on performance. Operating systems reserve TLB entries for either very frequently used pages, or pages whose TLB refill costs would be excessive. The TLB is effectively divided into a region for operating system use and the rest is used for user-level mapping. The reserved entries typically map user page tables and kernel data. Changes in memory usage described in Section 3.3, and also identified by [NUS+93, UNS+94], have increased the amount of kernel data and active page tables.

The optimal number of reserved entries is a function of application and user-level server activity. Dynamically changing the number of reserved entries is proposed by Uhlig et al. [UNS+94]. They use a simple scheme to tune the partition boundary between reserved and user-mapping entries depending on the activity of the machine. Their results indicate that dynamic partitioning performs better than static partitioning even when the partition is statically tuned to the application.

Software improvements to TLB performance appear to be limited to data structure optimisations for refill and TLB placement policy optimisations. Careful consideration of TLB interaction with operating system design and activity [NUS+93, UNS+94] has shown that performance gains are possible in this area.

### 4.2.2 Hardware

Currently, much research effort has been directed towards increasing TLB coverage, the focus being on increasing the number of TLB entries, increasing the page size, or using multiple page sizes.

Increasing the number of TLB entries is an obvious solution, as illustrated by Chen [CBJ92]. However, large fully associative structures are difficult to build. Reducing the associativity to increase the number of entries is a valid technique [UNS+94, TH94]. However, it is not clear whether the number of entries could be increased sufficiently to cover a significant proportion of larger memory sizes. For example, to cover 10% of 512 MB would require 6,554 entries for 8 KB pages.

Larger page sizes appear to have the most potential to increase TLB coverage. Unfortunately, larger page sizes also have the potential to dramatically increase working set sizes due to internal fragmentation. Talluri [TKHP92] illustrated that a moderate increase in page size from 4 KB to 32 KB resulted in an average increase of 60% in working set size. TLB coverage using 32 KB pages still only represents a very small proportion of a large memory; a large 128 entry TLB using 32 KB pages covers less than 1% of 512 MB. Much larger page sizes appear unusable for general use.

The use of multiple page sizes to combat both TLB coverage limitations and internal fragmentation seems to be the best approach. Investigations have shown [TKHP92, CBJ92, TH94] a dramatic decrease in TLB miss overheads with only a moderate increase in working set size. However these studies, together with others [KTNW93, Mog93], are quick to point out that very little research has been done to investigate the issues raised and overheads involved in managing multiple page sizes. Kagimasa et al. [KTM91] describe a system using multiple page sizes in a partitioned address space, though their aim is to reduce storage costs, not TLB overheads.

Talluri [TH94] proposes two sub-block TLB designs which provide some advantages of multiple pages sizes while requiring little or no operating system modifications. In a sub-block TLB, each TLB entry maps a large *superpage* to a number of smaller frames, not all of which need to be resident. This has the advantage of increasing TLB coverage, because each entry maps larger virtual pages but allows paging to occur on smaller blocks.

Talluri only considers medium sized superpages (64 KB) and it is not clear whether their scheme will scale to larger superpages required for large memory machines.

# 5 Discussion

The work described in the previous section is promising, but it has mostly involved simulations of hardware performance. There has been little investigation of the operating system overheads involved in using the new TLB designs. For example, the potential increase in TLB coverage provided by multiple page sizes managed by a GPT-style page table comes at the expense of increased complexity in physical memory and backing store management. As well, reducing the number of TLB entries needed to map large objects requires large transfers during paging.

The new TLB designs are being measured by how well they support microkernel based systems such as Mach. In particular, they are considering the impact of moving kernel data structures and services into user-level servers, which effectively fragments the kernel address space. More work needs to be done to evaluate these designs for other types of operating system architectures such as Mungi.

# 6 Conclusion

Traditional approaches to virtual memory address translation are being severely taxed by the demands of modern operating systems. In particular, the move to 64 bit architectures and the desire to support persistent and single address space systems increase the size and sparsity of virtual address spaces. While there is some work being done currently to improve TLB performance, these proposals do not consider the special needs of systems such as Mungi in which translation and protection functions need to be separated. Much work remains to be done to determine the best approach to implementing sparse address space systems on currently available processors which use software loaded TLBs.

# References


[ABC+83] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26, 1983.

[CBJ92] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *19th International Symposium on Computer Architecture*, May 1992.

[CLBHL92] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, Seattle, 1992.

[CM88] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Den70] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3), September 1970.

[Elp93] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report 9312, School of Computer Science and Engineering, University of New South Wales, November 1993.

[Hei93] Joe Heinrich. *MIPS R4000 user's manual*. Prentice-Hall, 1993.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *17th Annual Computer Science Conference*. Australian Computer Science Communications, January 1994.

[HH93] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[IBM78] IBM. *IBM System/38 technical developments*. Order no. G580-0237. IBM, Atlanta, Ga., 1978.

[KTM91] Toyohiko Kagimasa, Kikuo Takahashi, and Toshiaki Mori. Adaptive storage management for very large virtual/real storage systems. In *18th International Symposium on Computer Architecture*, May 1991.

[KTNW93] Yousef A. Khaldi, Madhusudhan Talluri, Michael N. Nelson, and Dock Williams. Virtual Memory Support for Multiple Page Sizes. In *4th Int'l Workshop on Workstation Operating Systems*, Napa, California, October 1993. IEEE.

[Lee89] Ruby B. Lee. Precision architecture. *Computer*, January 1989.

[Lie94] J. Liedtke. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, SchloßDagstuhl, Germany, September 1994.

[Mil90] Milan Milenkovic. Microprocessor memory management units. *IEEE Micro*, 10(2), April 1990.

[Mog93] Jeffrey C. Mogul. Big Memories on the Desktop. In *4th Int'l Workshop on Workstation Operating Systems*, Napa, California, October 1993. IEEE.

[MWO+93] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel. Technical Report 9, SARC, Dept. Computer Science, City University, London, 1993.

[NUS+93] David Nagle, Richard Uhlig, Tim Stanely, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: a quantitative approach*. Morgan Kaufmann, 1990.

[RA85] John Rosenberg and David Abramson. MONADS-PC - A capability-based workstation to support software engineering. In *Proc 18th Hawaii Int'l Conf. on System Sciences*, 1985.

[RKA92] John Rosenberg, J. L. Keedy, and D Abramson. Address mechanisms for large virtual memories. *The Computer Journal*, 35(4), August 1992.

[SAK+89] Eugene H. Spafford, James E. Allchin, Gregory Kenley, David V. Pitts, and C. Thomas Wilkes. *Anatomy of a Multicomputer: The First Six Years of Clouds*. Academic Press, Boston, MA, 1989.

[SGH+89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: an object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[Sit92] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, Maynard, M.A., 1992.

[TH94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[TKHP92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *19th International Symposium on Computer Architecture*, May 1992.

[UNS+94] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, August 1994.

[VRH93] J. Vochteloo, S. Russell, and G. Heiser. Capability based protection in the Mungi operating system. In *3rd Int'l Workshop on Object-Orientation in Operating Systems*. IEEE, 1993.

[Wil91] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4), June 1991.