Intended for further publication.
Do not copy! Do not circulate!

Limitations of Page Table Translation Schemes

Jochen Liedtke
GMD — German National Research Center for Information Technology *
jochen.liedtke@gmd.de

Abstract

This paper examines the current limitations imposed various page table mechanisms to the virtual address space concept. It also estimates to what extent *sparsity, granularity, hierarchy, sharing* and *scalability* of virtual address spaces might be improved. From this point of view, the mechanisms of *multi-level*, *inverted* and *guarded* page tables are analyzed. The results show that the requirements of sparsity, fine granularity, inexpensive sharing and fast operations on large address space regions can all be satisfied by guarded page tables. However, traditional multi-level and inverted page tables each exhibit certain limitations with respect to these requirements. Related problems concerning TLBs, caches and OS book-keeping are pointed out but not discussed.

1 Motivation

Virtual address spaces are based on MMUs and form a basic element of most system architectures. Upcoming 64-bit processors demand better support for sparsity. Object-oriented systems and distributed memory schemes would also profit from more fine-grained address spaces.

To reason about the future use of address spaces, not only in operating systems but also at the application level, we need to understand the actual limitations of the concept and to discuss what improvements might be possible and where the limitations become inherently hard.

This paper concentrates on the limitations arising from page table structures. In section 2 it briefly presents multi-level, inverted and guarded page tables. Section 4 analyzes the costs of these mechanisms in various "dimensions": sparsity, granularity, hierarchy, sharing, scalability.

*GMD I5.RS, 53754 Sankt Augustin, Germany

2 Page Table Mechanisms

Most conventional processors operate with fixed sized 4K or 8K pages, e.g. Intel x86, PowerPC, PA-Risc, Sparc, Mips R4000 and Alpha. (Alpha and R4000 also permit larger pages). Others, e.g. Jouppi [1988], Kagimasa et al. [1991] and Talluri et al. [1992], experimented with larger pages and mixed page sizes. To the author's knowledge, there is no modern processor that supports finer granularity.

2.1 Multi Level-Page Tables (MPT)

In the case of conventional *multi-level page tables*, a virtual address is transformed in several steps. Each one works with a page table whose entries refer either to page tables of the next level or (for the last level) to data pages. Usually, page tables and data pages have a fixed size 2^p.

Figure 1 shows how a virtual address is translated by a 4-level page table tree. In this example, we use 4 entries per page table and denote the address in the 4-adic system so that each page table decodes one digit of the address. The figure shows the translation of the virtual

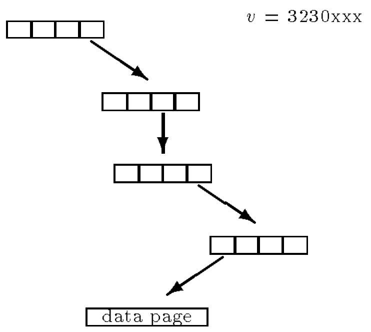


Figure 1: *Multi Level Page Table.*

address 3230xxx. 3230 is the page number and xxx the offset within the page. The leftmost digit "3" is used to index the top level page table. The corresponding entry points to the next level-page table which is indexed by the next digit "2" etc. If the process encounters a nil pointer, it signals a page fault. Otherwise, the translation stops at the bottom-level page table and uses the last pointer as the physical address of the page.

Since they translate only 2 address bits per step, 4-entry tables are unrealistic. In practice, page tables hold, for example, 1024 entries and have a size of 4 K. With these parameters, 64-bit virtual addresses require a 6-level page table tree.

2.2 Inverted Page Tables (IPT)

Inverted page tables (pioneered in the IBM /38 [Pinnow et al. 1982; Cocke 1981], also used in IBM 801 [Chan and Mergen 1988], PA-Risc [Hewlett Packard 1990] and Monads [Rosenberg et al. 1989]) consist of one entry per physical page frame. Each entry contains the virtual address of the mapped page and in some variants a pointer to the corresponding frame. A table entry is chosen by means of a hash function which uses all bits except the offset part of the virtual address v (see figure 2). If the virtual address stored in the table entry matches with v, the physical page is found. Otherwise rehashing is required or a page fault is signaled.

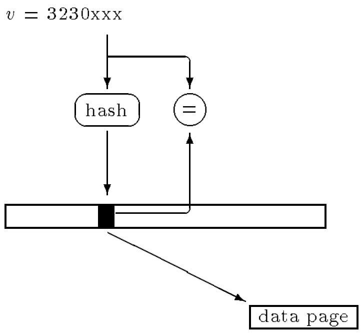


Figure 2: *Inverted Page Table.*

In contrast to simple inverted page tables, *hashed page tables* [Huck and Hays 1993] additionally contain the physical page address per table entry, thus permitting page aliasing.

2.3 Guarded Page Tables (GPT)

The main problem with multi-level page tables is sparsity: many page table entries may be required for non mapped pages. Recall the example from figure 1, where the mapping of page 3230 was shown. Assume that no pages are mapped in the regions [3000…3223] and [3300…3333]. In figure 3 the corresponding nil pointers are marked by •. The second- and third-level page

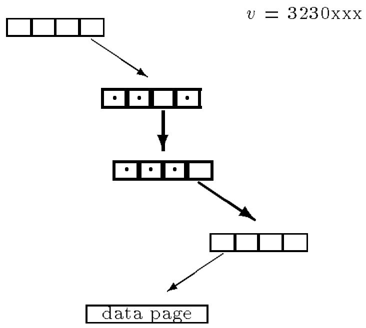


Figure 3: *Sparse Multi Level Page Tables.*

tables are extremely sparse: each contains only one single non-nil entry. Consequently, there is only one valid path through these two tables : when the leftmost digit is "3", the remaining address must have "23" as its next digits; all other addresses will lead to page faults. As shown in figure 4, we can omit the two page tables and

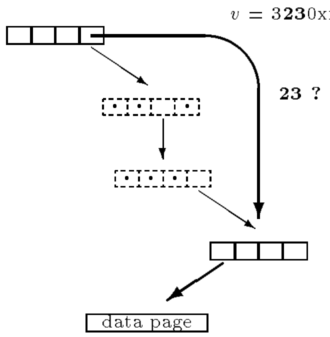


Figure 4: *Guarded Page Tables.*

also skip the according translation steps. Instead, whenever entry 3 of the top level page table is reached, we have to check whether "23" is a prefix of the remaining address. If so, this prefix is stripped off and the translation process can continue at the level 4 page table.

Therefore, each entry is supplemented by a bit string g of variable length which is referred to as *guard*. This is the key idea of *guarded page tables*.

Furthermore, each page table entry contains not only the pointer to the next level table, but also a size specification. It defines the size of either the next level's page table or data page. All powers of 2 are admissible, i.e. a mixture of 2-, 4-, …1024-, …entry page tables and data pages can be used.

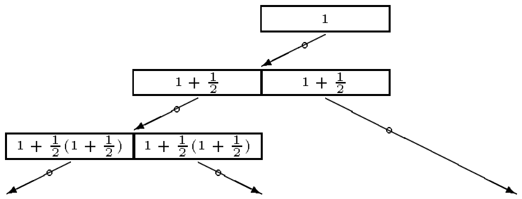
Note that the length of the guards as well as the next level's table size may vary from entry to entry.

Remark: Guarded page tables include conventional multi-level tables as a special case: if the actual guard has length zero, a translation step works exactly like in the conventional mechanism.

A detailed description of the GPT mechanism and of GPT performance can be found in [Liedtke 1994b; Liedtke 1993a; Liedtke 1993c; Liedtke 1994a].

> *Fewer than 2 guarded entries are necessary per page, regardless of the address-space size and page sizes.*

Look at a binary tree of guarded entries leading to k reachable pages. Then each page table (node of the tree) consists of two guarded entries. Nil entries can always be eliminated, because a binary page table with one nil entry is a single-entry table which, by construction, can be omitted by means of a guard. Therefore, the resulting tree has only the minimal number of binary nodes needed to reach all k pages. We start at the root and distribute the costs down to the leaves. The first entry costs one memory cell. If an entry points to another binary page table, we add half of its cost to the costs of each of the two entries:



So the space costs per page are—

$$1+\tfrac{1}{2}\big(1+\tfrac{1}{2}(1+\cdots)\big) \;=\; 1+\tfrac{1}{2}+\tfrac{1}{4}+\cdots \;<\; 2$$

Therefore, $2k$ entries are enough for k reachable pages. This upper bound on the tree size is independent of the address-space size and of the set of reachable pages.

The performance of the translator depends on the number of required translation steps. To achieve high performance, it is essential to reduce the depth of the page-table tree.

It turns out that, to translate an n-bit wide page number, we can easily construct a small tree so that all pages have a depth of at most $n/2$,

> *The steps can be further reduced to $n/4$ by means of parallel translators which handle 4 adjacent entries simultaneously.*

For practical reasoning we consider 64-bit virtual addresses. 16-byte pages require 60 address bits to be translated. From the previous section we know that this can be done by small guarded page table trees in at most $60/4 = 15$ steps, provided a 4-associative translator is used; but 15 are still too many steps.

Side Entry Point Caching

Fortunately, we can drastically reduce the average number of steps by caching side entries in the translation tree. We divide the address space into equal-sized regions and cache the optimal entry points of accessed regions (see figure 5).

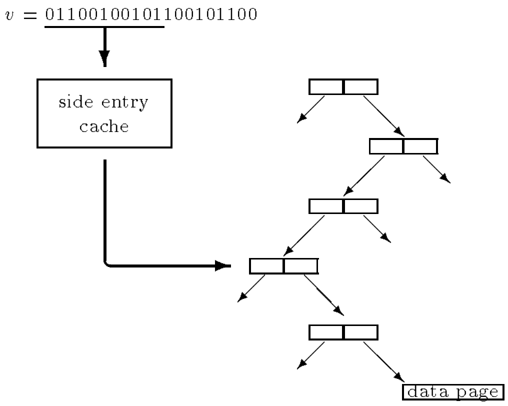


Figure 5: *Side Entry Translation.*

Whenever a complete translation process comes to a point where for the next step less than e.g. 24 bits of the virtual address are left for decoding, the actual entry is stored in the side entry cache under the corresponding 16 MB region number (even if it covers more than 16 MB). A subsequent translation of an address in the same region will lookup the side entry cache by the same region number and find the cached entry.

If we use, for example, 16 MB regions, a side entry cache hit reduces translation to at most 3 steps for 4 K pages and 5 steps for 16 byte pages. This is comparable to conventional systems. Note that side entry caching takes a certain region-level locality for granted.

User Level Mapping

Fine-grain mapping permits access control on the level of logical storage objects. In this way, it can be used reasonably both in the area of classical imperative programming languages and in object-oriented and declarative languages, in particular for distributed or massively parallel systems. Typical applications are *virtual aliasing* and *call-on-reference* (calling a user-defined procedure upon access).

GPTs permit safe user-level remapping operations without OS intervention. They can be used for associating specific access semantics to address space regions, for example, 'delay upon read access' (variable value has not yet been computed), 'signal upon write access', 'remote object invocation', 'access by proxies' [Shapiro 1986] or simply 'access protocol'. Remote object access can be implemented by local object invocation methods. Memory accesses are performed directly in the local case and algorithmically in the remote case. Distinction is by hardware.

3 Cost Measures

We use optimal worst case models whenever possible. This means, for any point in the architectural design space, we choose one or more cost measures and evaluate them assuming that the architecture is used in the most expensive way (worst case) but that the architecture is implemented optimally for this case. For illustration: when discussing the space costs of a certain granularity, we assume that all objects are of this minimal size (worst case), but chose page and page table size such that the space costs are minimized.

Exploding costs then indicate architectural limitations. If better case models behave similar, we have to suppose the limitations to be hard.

3.1 Memory Overhead

Our main cost measure is *memory overhead* which is defined as

$$\frac{\text{required memory} - \text{application memory}}{\text{application memory}} .$$

A memory overhead of 100% means: for an application requiring 1M of memory for its code and data, an additional 1M is needed for page tables and unused memory due to fragmentation. We assume that memory overheads up to 50% will probably be tolerated and beyond 100% will probably not; 50% – 100% is a grey area.

Note that the memory overhead is solely determined by the set of pages which are actually mapped in main memory. Pages which do not reside in main memory are considered to not require page table space in main memory. Accordingly, an address space may also suffer from sparsity, if only few of its pages are mapped to main memory.

3.2 TLB Enlargement

Our second cost measure is the *TLB enlargement*. We assume a fully-associative TLB supporting single sized pages of 4K and take its size as 1. Now we estimate for different page sizes, what TLB size is required to achieve a comparable performance.

Assume a given program based on 4K pages. We transform this program into a sematically equivalent one using a new page model. This transformation maps the set of originally used 4K pages to a new set of pages covering the same same amount of memory. Furthermore, we assume that the original program's TLB working set consist of at most n pages, where n is the number of TLB entries. By the above transformation, these working sets are mapped to new working sets of at most n' pages. The intuitive idea is that a new TLB of n' entries will exhibit roughly the same miss ratio on the transformed page set as the n-entry TLB on the original page set. Mathematical reasoning[1] confirms that we can expect a non-worse miss rate when using a fully-associative TLB of n' entries for the transformed program. Then, the required TLB enlargement is $\tau = n'/n$.

We assume that TLB enlargements up to 2 will probably be tolerated, but beyond 5 certainly not.

4 Examination

Three classes of MMU mechanisms are examined: multi-level page tables (MPT), inverted/hashed page tables (IPT) and guarded page tables (GPT). The basic problem is to find out in which range each facility's

[1] For a given point in time, let p_i be the access probabilities for the n pages mapped by the original TLB. Then the miss probability is $q = 1 - \sum_{i=1}^{n} p_i$. If the replacement mechanism of the new TLB would ensure (what is not realistic) that the new TLB always holds exactly the transformed working set of n' pages (with access probabilities p'_i), it would have the same miss probability $q' = q$, since $\sum_{i=1}^{n'} p'_i = \sum_{i=1}^{n} p_i$.

A bit more realistically, we assume that the original TLB is controlled by LRU replacement and the new one by optimal replacement. Optimal replacement may exchange any of the transformed pages with a new page not belonging to the transformed working set, but only if $p''_i \geq p'_i$, where p''_i is the reference probability of the new page. This implies $\sum p''_i \geq \sum p_i$ so that the miss probability of the new TLB is at most q.

Of course, optimal replacement is not realistic, but cache simulation experiences show that LRU replacement is very similar to optimal replacement provided the cache is large enough. Therefore, $q \approx q'$ should also hold, if the new TLB uses LRU replacement.

performance will be acceptable and where it becomes certainly unusable.

4.1 Sparsity

In 64-bit systems, address spaces will be in many cases sparsely occupied. Then pages are allocated with large gaps in the virtual address space.

Assume that each page is followed by a large gap. We discuss the memory overhead $\omega(s, g)$ in this situation, depending on page size s and gap size g. Models with page sizes of 4K, 1K, 256 and 64 bytes are examined.

Although in practice, mapped pages are never equally distributed over the address space, this simple model is well-suited for scaling sparsity, since the costs of other distributions can easily be derived: If n consecutive pages are followed by one gap, the overhead is $\omega(ns, g) = \omega(s, g)/n$. In the case of a set of n pages and gaps, the total overhead is $\sum s_i \omega(s_i, g_i) \ / \ \sum s_i$.

Figure 6 illustrates that IPT and GPT support arbitrary sparsity. IPT size depends only on the physical memory size, and GPT never requires more than 2 guarded entries per mapped page, regardless of how the pages are mapped.

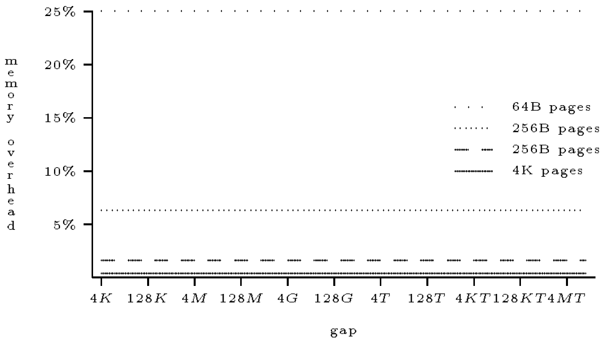


Figure 6: *IPT and GPT Sparsity*

Sparsity is very expensive in conventional MPTs. Figure 7 shows that 4K pages require 100% overhead at 4M gaps. Finer granularity reduces this already very narrow limitation dramatically: 1K pages to 256K, 256-byte pages to 16K gaps and 64-byte pages do not even permit 4K gaps.

Sparsity is expensive because multiple and large (4K) page tables are needed for one mapped data page. Smaller page tables would reduce the overhead but increase the tree depth, i.e. the costs per address translation. The latter problem can probably be avoided by a side entry cache as used for guarded page tables (see section 2.3). Then 64-byte (16-entry) page tables would be possible and dynamically perform like GPTs.

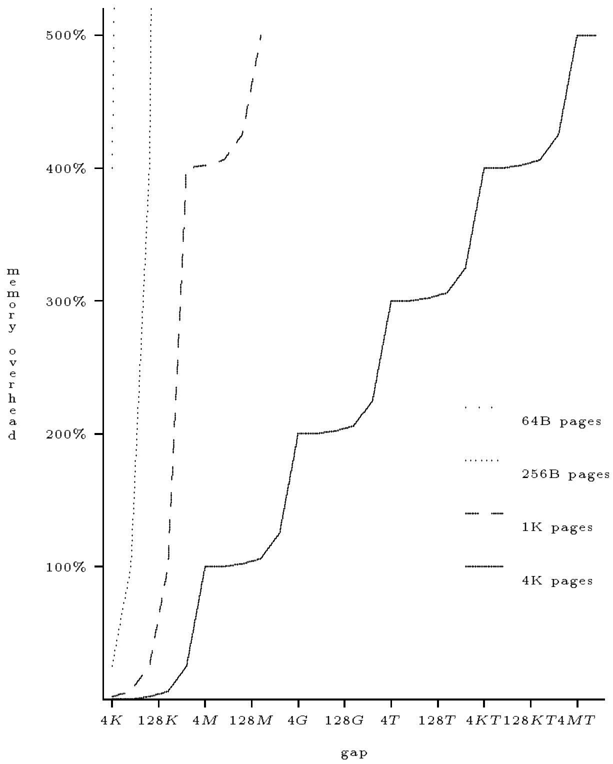


Figure 7: *MPT (4K page tables) Sparsity*

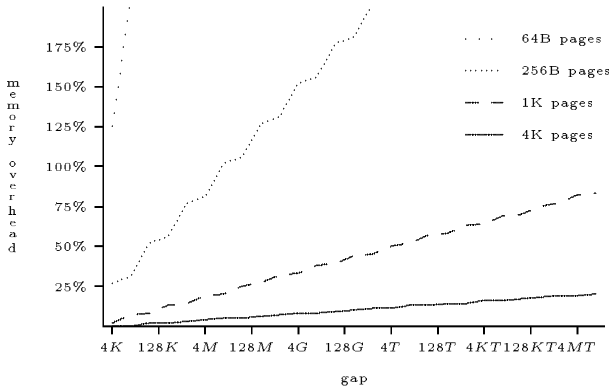


Figure 8: *MPT (64-byte page tables) Sparsity*

Figure 8 shows the sparsity overhead in the case of MPTs with 64-byte page tables. (Note the different scale of the y-axis.) Sparsity becomes feasible with 4K pages, perhaps also with 1K pages. Finer granularity cannot be combined with sparsity.

4.2 Granularity

Fine granularity is of interest for object oriented systems, databases, virtually shared memory and – if it is really fine-grained – even for user level programs.

Note that disk granularity does *not* restrict virtual memory granularity. VM can be used for logically structuring memory, for aliasing, access control, sharing between address spaces, distribution, objects of flexible size etc. All this is not related to disk traffic. Furthermore, techniques like log-structured files, clustering and disk caches decouple sector size and page size.

For evaluating the memory overhead, we need a data model. Considering pages only is inappropriate, since this would ignore the effects of fragmentation and object size distribution.

Instead, we use a model based on objects of 7 "typical" sizes, 64K, 16K, 4K, 1K, 256, 64 and 16 bytes. An average object in class s has size $s + \min(s, p)/2$, where p denotes the page size. For example, an average 16K-object has size 18K when using 4K pages, 16.5K when using 1K pages.

As a scale for granularity, we use the smallest supported object size. A granularity of 1K therefore means that the system holds objects of 64K, 16K, 4K and 1K. Since the overhead depends on the object size distribution, we examine four distributions. In 'WC', all objects in memory belong to the most expensive class, i.e. only objects of the smallest permitted size are present. This model yields lower (pessimistic) bounds; no architecture will perform worse than 'WC'. In '1/1', each class occupies the same amount of memory, i.e. there are 4 times as many 16K- as 64K-objects present. In '1/2', the next finer granular class occupies 1/2 of the memory, i.e. the ratio of 64K-, 16K- and 4K-objects is 1:2:4; in '1/3', the corresponding ratio is $1 : \frac{4}{3} : \frac{16}{9}$.

The results (figures 9, 10, 11, 12) show:

- The curves of 'WC', '1/1', '1/2' and '1/3' are located relatively close together. Therefore, most other architectures (distributions) will probably be about in the same range. We should not expect to find a general architecture permitting considerably finer granularity than '1/3'.

- MPT overhead strongly depends on sparsity. MPTs can only be used if very low sparsity is guaranteed.

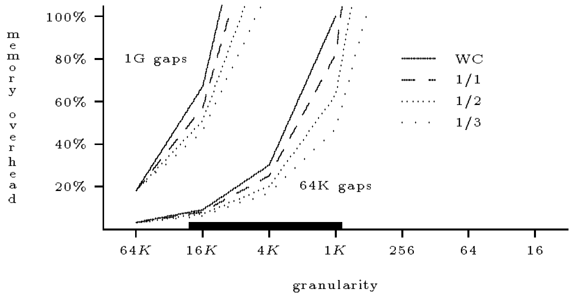


Figure 9: *MPT (4K page tables) Granularity*

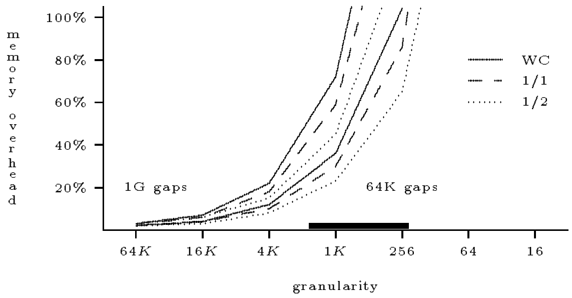


Figure 10: *MPT (64-byte page tables) Granularity*

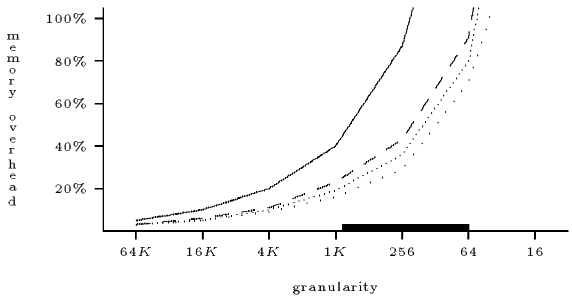


Figure 11: *IPT Granularity*

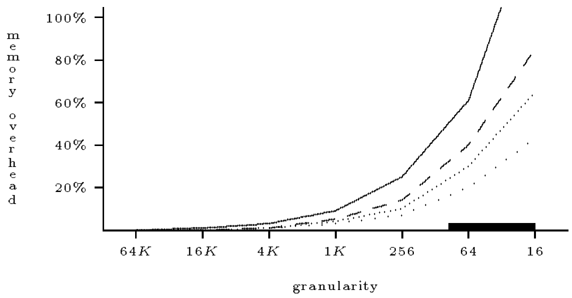


Figure 12: *GPT Granularity*

- MPT-4K enables only coarse granularity. For the other mechanisms, the critical limits (50% to 100% overhead) are:

MPT-64	2048...256
IPT	1024...64
GPT	128...16

- For any granularity and for each distribution, GPT requires the lowest overhead.

Although the more static memory overhead seems to permit fine granularity, dynamic overhead may restrict it. If a TLB of n entries is appropriate for 4K pages, as discussed in section 3.2, we must expect that 1K pages will require at least a $4n$-entry TLB. Nevertheless, for

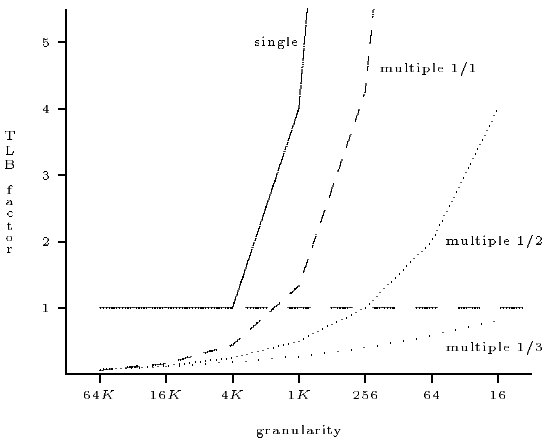


Figure 13: *TLB factor, single and multiple page sizes*

a single page size, this simple model shows (figure 13) that granularities below 2K or 1K become too expensive. The main reason for this bad performance is the uniformity of pages: larger objects have to pay the same price as smaller ones. This disadvantage can be avoided by mixed page sizes as supported by the GPT mechanism. Provided that the TLB manages mixed page sizes, 4K-objects then perform as well as in the conventional case and larger objects even better. (Not only in terms of required TLB entries, but also in TLB misses.) Figure 13 shows the required TLB size compared to a conventional 4K architecture. We examine '1/1', '1/2' and '1/3' object size distributions *which now refer to the average TLB working set.* Note that these TLB distributions are determined solely by the dynamic program behavior and do not at all depend on the static distribution of objects in physical memory.

Not surprisingly, TLB utilization strongly depends on the actual distribution. In the '1/3' case it is always even better than in the conventional case, whereas '1/1' restricts granularity to approximately 256 bytes. So programs in fine granular architecture will have to avoid working sets with "too many" small objects. Whether we can find such programming models cannot be answered with certainty today. (We must not believe in benchmarks until there are concrete fine granular architectures and practically relevant applications adapted to it.) Nevertheless, I personally believe that many interesting programming models will perform like '1/2' or better.

- TLBs must support mixed page sizes efficiently.
- We have to look for programming models that behave like '1/2' or better.

4.3 Hierarchies

There are at least three time critical operations on sets of pages: transferring a multi-page object or even a complete region of virtual memory inter or intra address space, locking and unlocking it. Fast locking/unlocking is required for

1. atomic updates or state changes, e.g. swap in/out,
2. access control and monitoring,
3. lazy operations, e.g. copy on write.

MPTs and GPTs support hierarchies and the mentioned operations on regions by their tree structure. Locking a 4G region requires only the modification of 1 page table entry. Operations on arbitrary regions of size n cost $O(\log n)$. Since the IPT mechanism does not support hierarchies, operations on n pages require $\min(n, r)$ modifications of page table entries, where r is the number of physical frames.

Based on 256M physical memory and 256-byte pages, figure 14 shows the necessary number of page table entry modifications depending on region size. For MPT and GPT the numbers are worst case costs; in best case only 1 modification is required. Note the logarithmically scaled y-axis.

For non-small regions, IPT and even MPT-4K is too inefficient. (There is a folklore argument saying that remapping costs are dominated by basic hardware and OS overhead. This argument is wrong when dealing with new architectures: a system call instruction can be made as efficient as any indirect call and OS overhead can be kept low, as Synthesis [Massalin 1992] and L3 [Liedtke 1993b] show.)

- MPT-64 and GPT enable hierarchies and fast operations on regions.
- IPT does not.

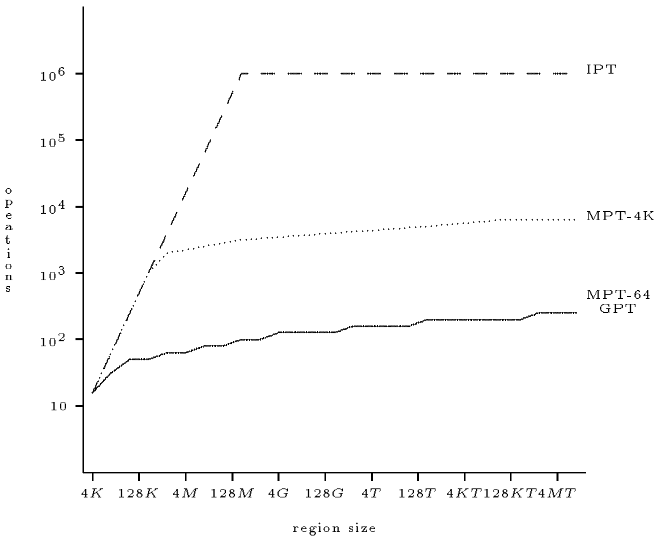


Figure 14: *Region operations*

4.4 Sharing

Pages can be shared intra- or inter-address-space. A single shared page always requires 1 additional page table entry per shared occurrence. Multi-page regions can be shared far cheaper in the MPT and GPT cases, by means of sharing complete page table subtrees. The 'hashed page table' [Huck and Hays 1993] IPT variant also permits sharing, but needs a constant overhead of 1 additional entry per sharing occurrence and per page.

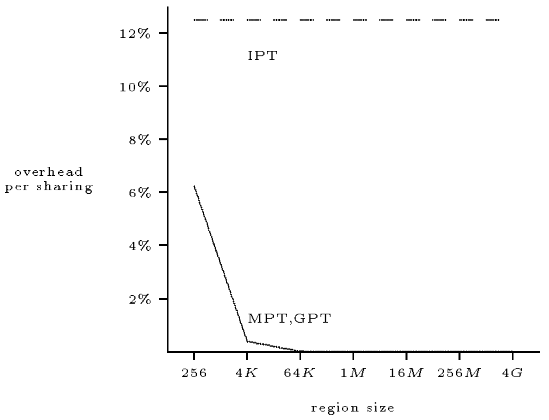


Figure 15: *Region sharing, 256-byte pages*

IPT permits only very limited sharing of this type. 8-fold sharing in this example requires already 100% overhead. For one special architecture, the pure single address space system, the IPT overhead can be reduced, but only for a small number of protection domains or very specialized protection models.

To illustrate the effect of page table mechanisms on conventional 32-bit address spaces and 4K pages, table 1 shows the required page table entry (PTE) bytes per data page in the case of a large server with 1000 address spaces of 128 MB in size. In the MPT case, we assume

scenario		mapped contingent (equal-distributed)			
		100%	50%	10%	2%
A	MPT	4	8	40	200
	IPT	16	16	16	16
	GPT	16	16	16	16
B	MPT	4+128	8+256	40+1.3K	200+6.4K
	IPT	16K	16K	16K	16K
	GPT	16+16	16+32	16+160	16+800
C	MPT	4K+128	8K+256	40K+1.3K	200K+6.4K
	IPT	16K	16K	16K	16K
	GPT	16+64	16+128	16+640	16+3.2K

Table 1: *PTE-bytes per Mapped Page.*

a two-level page table with 4K per table and 4 bytes per entry. GPT and IPT entries consume 8 byte each, but for IPTs we assume that only half of the available entries can be used to avoid clashing.

In scenario A, no pages are shared between the 1000 address spaces. In scenario B, we look at one 4M region which is 1000-fold shared. In the MPT case, scenario B profits from the fact that sharing can be implemented by sharing a complete lower level page table (which exactly spans 4M). In scenario C, a 1M region is shared 1000-fold. Now the lower level MPT table can no longer be shared. GPT always profits from sharing subtrees, but IPT never does.

4.5 Scalability

Can we scale address spaces? Are they lightweight resources which can e.g. be used for implementing small active objects?

Multiple address spaces can be implemented by extending the 64-bit virtual addresses by an address space number. The TLB and the page-table walker then handle e.g. 80-bit addresses, where the most significant 16 bits are taken from the actual address space register. The basic overhead for an address space containing 1 mapped page is:

MPT-4K	24576 B
MPT-64	1024 B
IPT	32 B
GPT	16 B

Only IPT and GPT require really low overhead, but we have to recall that IPT is not good in sharing.

5 Open Questions

Small and differently sized pages create serious problems for TLB and cache architectures. The crucial points are:

1. Can we build large and fast physically tagged caches, when pages are small?
2. Can we build fast caches that tolerate aliasing?
3. Can we build fast TLB's for mixed page sizes?
4. Can we efficiently change regional attributes without flushing the entire TLB?

Ongoing research deals with these questions. It is yet too early for definite answers, but we are not pessimistic.

6 Related Work

64-bit systems: Opal [Chase et al. 1994], Mungi [Heiser et al. 1993], Monads [Rosenberg et al. 1989] and [Carter et al. 1992]. Page table mechanisms: [Organick 1972; Cocke 1981; Huck and Hays 1993]. TLBs and caches: [Koldinger et al. 1992; Kaiser and Czaja 1992; Nagle et al. 1993; Chiueh and Katz 1992]. User level mapping: [Appel and Li 1991; Hosking and Moss 1993]. OO systems: [Jul et al. 1988; Krakowiak et al. 1990].

7 Conclusions

Besides that we never can be sure to consider all relevant factors in such an examination, there are remarkable factors of uncertainty: research is required to find out whether appropriate TLBs and caches can be built and whether OS book-keeping can be made efficient.

Nevertheless, there might be good chances to achieve high sparsity, fine granularity, fast hierarchical operations, inexpensive sharing and lightweight address spaces.

While MPTs and IPTs support either *fine granularity, hierarchy and sharing* or *fine granularity and sparsity*, GPT supports all dimensions and seems even to perform at least as well as the other mechanisms in all dimensions so that no 'either–or' might restrict future innovative architectures.

Acknowledgements

Discussions with various people helped to discover the critical points concerning fine granularity. In particular, discussions with John Crawford and Intel's P7 group, Fujitsu's computer network systems and Hitachi's ultra-high-speed group were helpful. Also many thanks to Richard Uhlig for proofreading this paper.

References

Appel, A. W. and Li, K. 1991. Virtual memory primitives for user programs. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, pp. 73–80.

Carter, J. B., Cox, A. L., Johnson, D. B., and Zwanepoel, W. 1992. Distributed operating systems based on a protected global virtual address space. In *3rd IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL, pp. 75–79.

Chan, A. and Mergen, M. F. 1988. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems 6,* 1 (Feb.), 28–50.

Chase, J. S., Levy, H. M., Freely, M. J., and Lazowska, E. D. 1994. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems.*

Chiueh, T. and Katz, R. H. 1992. Eliminating the address translation bottleneck for physical address cache. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, pp. 137–148.

Cocke, J. 1981. Virtual to real address translation using hashing. *IBM Technical Disclosure Bulletin 24,* 6 (Nov.).

Heiser, G., Elphinstone, K., Russell, S., and Hellestrand, G. R. 1993. A distributed single address-space operating system supporting persistence. SCS&E Report 9302 (March), Univ. of New South Wales, School of Computer Science, Kensington, Australia.

Hewlett Packard 1990. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (1st ed.). Cupertino, CA: Hewlett Packard.

Hosking, A. L. and Moss, J. E. B. 1993. Protection traps and alternatives for memory management of an object-oriented language. In *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, pp. 106–109.

Huck, J. and Hays, J. 1993. Architectural support for translation table management in large address space machines. In *20th Annual International Symposium on Computer Architecture (ISCA)*, San Diego, CA, pp. 39–50.

Intel Corp. 1993. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual.* Intel Corp.

Jouppi, N. P. 1988. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *15th Annual International Symposium on Computer Architecture (ISCA)*, Honolulu, HA, pp. 281–289.

Jul, E., Levy, H., Hutchinson, N., and Black, A. 1988. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems 6,* 1 (Feb.), 109–133.

Kagimasa, T., Takahashi, K., Mori, T., and Yoshizumi, S. 1991. Adaptive storage management for very large virtual/real storage systems. In *18th Annual International Symposium on Computer Architecture (ISCA)*, Toronto, pp. 372–379.

Kaiser, J. and Czaja, K. 1992. ACOM – an access control monitor providing protection in persistent object-oriented systems. In *5th International Workshop on Persistent Object Systems*, Minato (Pisa), pp. 339–351. Springer.

Kane, G. and Heinrich, J. 1992. *MIPS Risc Architecture.* Prentice Hall.

Koldinger, E. J., Chase, J. S., and Eggers, S. J. 1992. Architectural support for single address space operating systems. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, pp. 175–186.

KRAKOWIAK, S., VAN MEYSEMBOURG, RIVEILL, M., ROISIN, C., AND DE PINA, R. 1990. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming 3,* 3 (Sept.), 11–22.

LIEDTKE, J. 1993a. A high resolution MMU for the realization of huge fine-grained address spaces and user level mapping. Arbeitspapiere der GMD No. 791, GMD — German National Research Center for Information Technology, Sankt Augustin.

LIEDTKE, J. 1993b. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, pp. 175–188.

LIEDTKE, J. 1993c. Some theorems about guarded page tables. Arbeitspapiere der GMD No. 792, GMD — German National Research Center for Information Technology, Sankt Augustin.

LIEDTKE, J. 1994a. Some theorems about restricted guarded page tables. Arbeitspapiere der GMD No. 834, GMD — German National Research Center for Information Technology, Sankt Augustin.

LIEDTKE, J. 1994b. *Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse.* München: Deutsches Patentamt. Patent application P 44 05 845.4, granted June 1995.

MASSALIN, H. 1992. Efficient implementation of fundamental operating system services. Ph. D. thesis, Columbia University, New York.

Motorola Inc. 1993. *PowerPC 601 RISC Microprocessor User's Manual.* Motorola Inc.

NAGLE, D., UHLIG, R., STANLEY, T., SECHREST, S., MUDGE, T., AND BROWN, R. 1993. Design tradeoffs for software managed TLBs. In *20th Annual International Symposium on Computer Architecture (ISCA)*, San Diego, CA, pp. 27–38.

ORGANICK, E. 1972. *The Multics System: An Examination of its Structure.* M.I.T. Press, Cambridge, MA.

PINNOW, K. W., RANWEILER, J. G., AND MILLER, J. F. 1982. The IBM system/38: Object oriented architecture. In D. P. SIEWIOREK, C. G. BELL, AND A. NEWELL (Eds.), *Computer Structures: Principles and Examples*, pp. 537–540. McGraw-Hill.

ROSENBERG, J., KOCH, D., AND KEEDY, J. L. 1989. A massive memory supercomputer. In *22nd Hawaii International Conference*, pp. 338–345.

SHAPIRO, M. 1986. Structure and encapsulation in distributed systems: The proxy principle. In *6th International Conference on Distributed Computing Systems*, Cambridge, MA.

SITES, R. L. (Ed.) 1992. *Alpha Architecture Reference Manual.* Digital Equipment Corp.

SPARC INTERNATIONAL, I. (Ed.) 1992. *The Sparc Architecture Manual, Version 8.* Prentice Hall.

TALLURI, M., KONG, S., HILL, M. D., AND PATTERSON, D. A. 1992. Tradeoffs in supporting two page sizes. In *19th Annual International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia, pp. 415–424.