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

MMU Impacts On System Architecture

Extended Abstract

Jochen Liedtke
German National Research Center for Computer Science (GMD) *
jochen.liedtke@gmd.de

Abstract

Virtual address spaces are based on MMUs and form a basic element of most system architectures. Scientific progress in this field is important, especially due to upcoming 64-bit processors, increasing object orientation and distribution. Since system architecture innovation and MMU research mutually influence each other, it is of strategic relevance which medium- and long-term MMU developments should be encouraged by OS research. To get a solid basis for this discussion, we first have to explore our potential design space.

This paper examines 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. It turns out that sparsity, fine granularity, inexpensive sharing and fast operations on large address space regions seem achievable, in case of guarded page tables even altogether. Some related problems, especially TLBs, caches and OS book-keeping are pointed out but not discussed in detail.

1 Motivation

Scientific and technical progress will sooner or later lead to system architectures which differ substantially from today's ones. Currently, 64-bit processors and object orientation enforce innovation, but it is still unclear what architectures will be appropriate. Future, still unknown ideas may do as well. On the one hand, we have no certain knowledge about the medium-term development directions; on the other hand, we determine them by concentrating research on some selected architectural paradigms. So the selection is critical.

The first step in this discussion has to be the exploration of the potential design space: which improvements of virtual address space mechanisms seem to be possible, which are probably impossible? Since this question will also be influenced by future ideas, each discussion incorporates a remarkable part of uncertainty. Nevertheless, it is necessary. We need to know what *probably* is possible in the hardware and basic software area. In a second step, we can discuss which of these potential features will be relevant for future OS and application architectures. The third step should include research efforts to build and explore both the promising system- and the requested hardware-architectures. (And of course, this cycle has to be repeated.)

This paper is part of step one.

2 Page Table Mechanisms

Most conventional processors operate with fixed sized 4K or 8K pages, e.g. Intel x86 [11], PowerPC [29], MIPS [28], PA-Risc [8], Sparc [37] and Alpha [36] (which also permits larger pages). Others (e.g. [12, 14]) experimented with larger pages and mixed page sizes. To the author's knowledge, there is actually no modern processor which really 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

*GMD I5.RS, 53754 Sankt Augustin, Germany

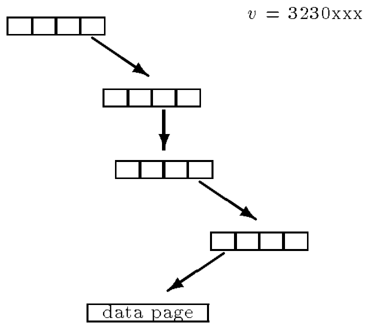

Figure 1: *Multi Level Page Table.*

of the address. The figure shows the translation process of the virtual address 3230xxx. 3230 is the page number and xxx the offset within the page. The leftmost digit "3" is taken for indexing 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 takes 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. Then 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 [32, 6], also used in IBM 801 [3], PA-Risc [8] and Monads [33]) 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.

In contrast to simple inverted page tables, *hashed page tables* [10] additionally contain the physical page address per table entry, thus permitting page aliasing.

2.3 Guarded Page Tables (GPT)

The main problem of multi-level page tables is sparsity: we need huge amounts of page table entries for

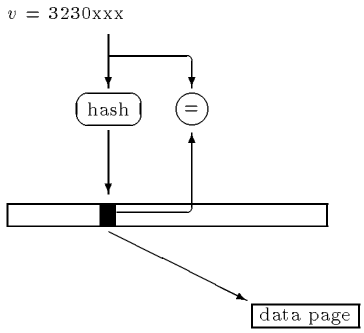

Figure 2: *Inverted Page Table.*

non mapped pages. Recall the example from figure 1, where the mapping of page 3230 was shown. Now we additionally 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

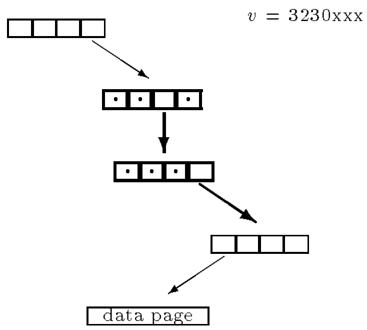

Figure 3: *Sparse Multi Level Page Tables.*

and third level page table are extremely sparse page tables: each one 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 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 prefix of the remaining address. If so, this prefix has to be stripped off and the translation process can directly 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 in addition a size specification. It defines the size of either the next level's

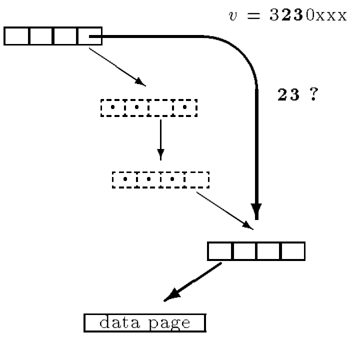

Figure 4: *Guarded Page Tables.*

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.

Detailed description of the GPT mechanism and of GPT performance can be found in [25, 18, 20, 24].

At most $n/2$ translation steps are needed for translating n address bits (using a tree with at most 2 entries per page).

The steps can further be reduced to $n/4$ by means of parallel translators.

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 to 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 equally sized regions and cache the optimal entry points of accessed regions (see figure 5).

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

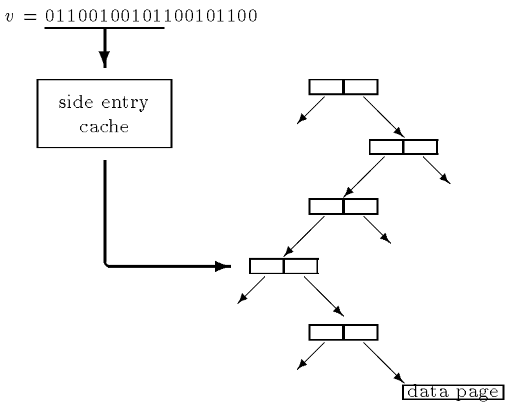

Figure 5: *Side Entry Translation.*

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' [35] or simply 'access protocol'. Remote object access can be done 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 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 performance may be acceptable and where it becomes certainly unusable.

Using benchmarks in this context is nonsense, since we deal with not yet existing architectures.

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

(The limited space of this extended abstract permits either to describe the models in detail or to present the results. I decided in favor of the results, although now it is easier to attack them.)

3.1 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.

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. Therefore, as cost measure we use the *memory overhead* depending on the gap between two mapped pages. Models with page sizes of 4K, 1K, 256 and 64 bytes are examined.

Sparsity is no problem of the IPT and GPT mechanisms, since IPTs 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.

Sparsity is very expensive in conventional MPTs. Figure 6 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 so 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

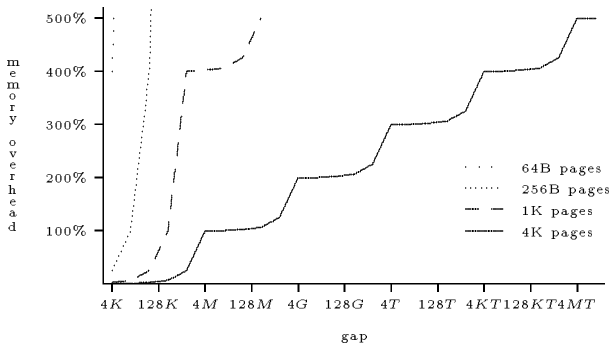

Figure 6: *MPT (4K page tables) Sparsity*

section 2.3). Then 64-byte (16-entry) page tables would be possible and dynamically perform like GPTs.

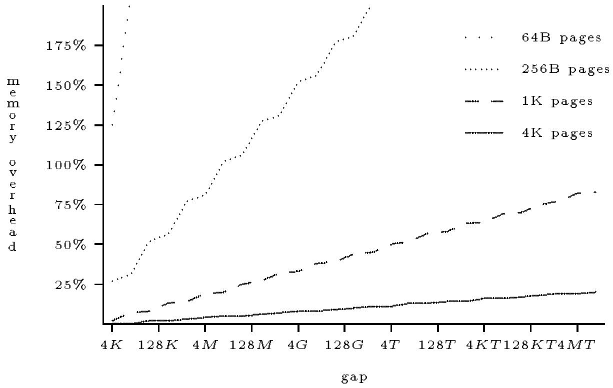

Figure 7: *MPT (64-byte page tables) Sparsity*

Figure 7 shows the sparsity overhead in the case of MPTs with 64-byte page tables. (Note the different scale of the y-axis.) Sparsity becomes possible with 4K pages, perhaps also with 1K pages. Finer granularity cannot be combined with sparsity.

3.2 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 ration of 64K-, 16K- and 4K-objects is 1:2:4; in '1/3', the corresponding ratio is $1 : \frac{4}{3} : \frac{16}{9}$.

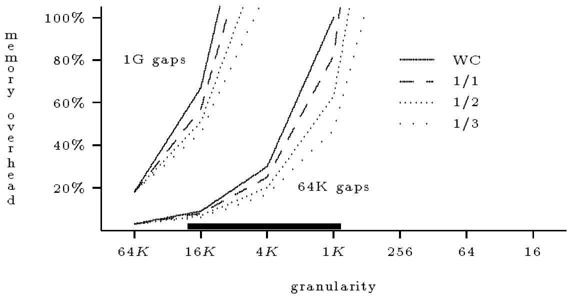

Figure 8: *MPT (4K page tables) Granularity*

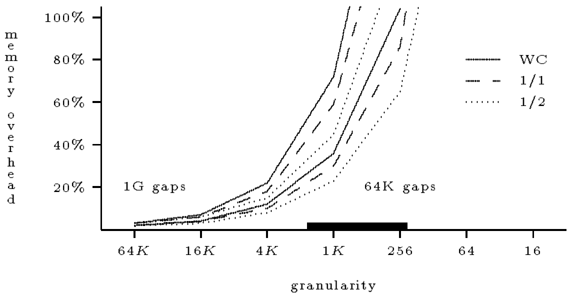

Figure 9: *MPT (64-byte page tables) Granularity*

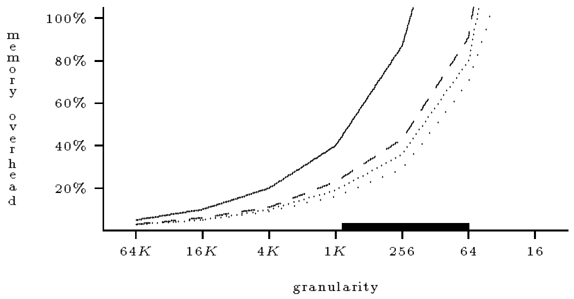

Figure 10: *IPT Granularity*

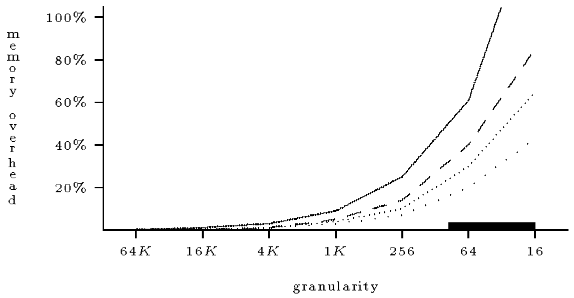

Figure 11: *GPT Granularity*

The results (figure 8, 9, 10, 11) 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.
- 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 each distribution, GPT requires the lowest overhead.

Although static overhead seems to permit fine granularity, dynamic overhead may restrict it. If a TLB of n entries is appropriate for 4K pages, we must expect that 1K pages will require at least a $4n$-entry TLB. This might even be too optimistic, since the increase of TLB misses is ignored. Nevertheless, for a single page size, this simple model shows (figure 12) 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 than smaller ones.

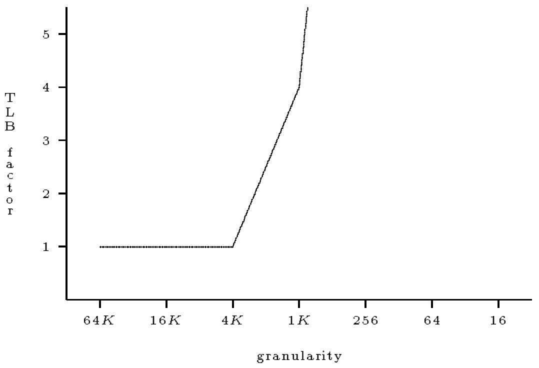

Figure 12: *TLB factor, single page size*

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, larger objects even better. (Not only in terms of required TLB entries, but also in TLB misses.) Figure 13 shows the

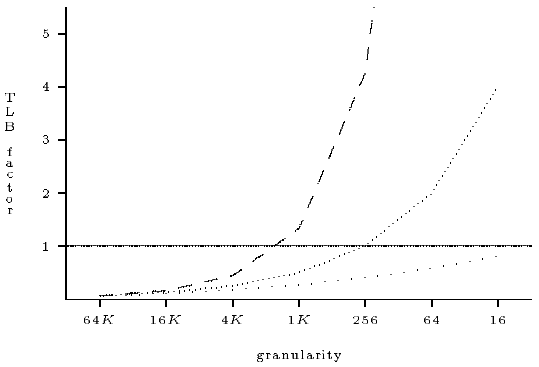

Figure 13: *GPT TLB factor*

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.

Unfortunately but not surprisingly, TLB utilization strongly depends on the actual distribution. In the '1/3' it is always even better than in the conventional case, whereas '1/1' restricts granularity to approx. 256 bytes. So programs in fine granular architecture will have to avoid working sets with "too much" 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.

Remark: The mixed page size TLB is *not* restricted to GPTs. (More about this in the full paper. Currently, we check patenting.)

(TLB miss discussion in the full paper.)

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

3.3 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. (And an appropriate TLB flush, full paper [23].) 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.

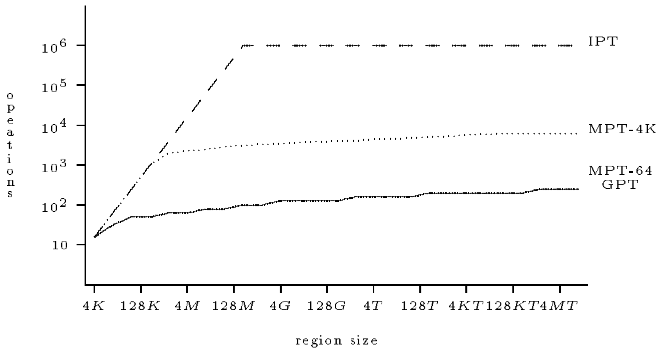

Figure 14: *Region operations*

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 [27] and L3 [19] show.)

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

3.4 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' [10] IPT variant also permits sharing, but needs a constant overhead of 1 additional entry per sharing occurrence and per page.

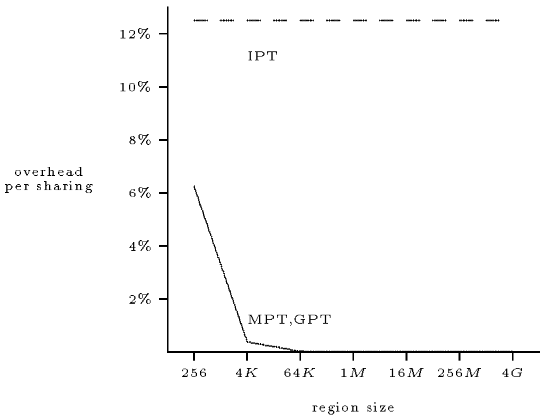

Figure 15: *Region sharing, 256-byte pages*

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. (Technical details and problems in full paper.[16, 7])

3.5 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. TLB and 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.

4 Open Questions

Small and differently sized pages impose serious problems on the TLB and cache architecture. Crucial points are

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

Ongoing research [26, 22, 21, 23] deals with these questions. It is yet too early for definite answers, but we are not pessimistic.

5 Related Work

64-bit systems: Opal [4], Mungi [34, 7], Monads [33] and [2]. Page table mechanisms: [31, 6, 10]. TLBs and caches: [16, 15, 30, 5]. User level mapping: [1, 9]. OO systems: [13, 17].

6 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 build and whether OS book-keeping can be made efficiently.

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

Discussion with various people helped to find out the critical points concerning fine granularity, especially with John Crawford and Intel's P7 group, Fujitsu's computer network systems and Hitachi's ultra-high-speed group.

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73–80, Santa Clara, CA, April 1991.

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

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

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

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

[6] J. Cocke et al. Virtual to real address translation using hashing. *IBM Technical Disclosure Bulletin*, 24(6), November 1981.

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

[8] Hewlett Packard, Cupertino, CA. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1st edition, 1990.

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

[10] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *20th Annual International Symposium on Computer Architecture*, pages 39–50, San Diego, CA, May 1993.

[11] Intel Corporation, Santa Clara, CA. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, 1993.

[12] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *15th Annual International Symposium on Computer Architecture*, pages 281–289, Honolulu, HA, June 1988.

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

[14] T. Kagimasa, K. Takahashi, T. Mori, and S. Yoshizumi. Adaptive storage management for very large virtual/real storage systems. In *18th Annual International Symposium on Computer Architecture*, pages 372–379, Toronto, May 1991.

[15] J. Kaiser and K. Czaja. ACOM – an access control monitor providing protection in persistent object-oriented systems. In *5th International Workshop on Persistent Object Systems*, pages 339–351, Minato (Pisa), September 1992. Springer Verlag Berlin Heidelberg New York.

[16] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston, MA, October 1992.

[17] S. Krakowiak, van Meysembourg, M. Riveill, C. Roisin, and R. de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11–22, September 1990.

[18] J. Liedtke. A high resolution MMU for the realization of huge fine-grained address spaces and user level mapping. Arbeitspapiere der GMD No. 791, German National Research Center for Computer Science (GMD), Sankt Augustin, 1993.

[19] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles*, pages 175–188, Asheville, NC, December 1993.

[20] J. Liedtke. Some theorems about guarded page tables. Arbeitspapiere der GMD No. 792, German National Research Center for Computer Science (GMD), Sankt Augustin, 1993.

[21] J. Liedtke. *Ein Cache/TLB-System ohne Färbungsrestriktion mit virtuell indiziertem und physisch markiertem Cache*. Deutsches Patentamt, München, April 1994. Patent application P 44 14 116.5.

[22] J. Liedtke. *Ein virtuell indizierter und physisch markierter Falschfarben-Cache*. Deutsches Patentamt, München, April 15 1994. Patent application P 44 12 929.7.

[23] J. Liedtke. *Flush Lookaside Buffer*. Deutsches Patentamt, München, May 1994. Patent application P 44 16 658.3.

[24] J. Liedtke. Some theorems about restricted guarded page tables. Arbeitspapiere der GMD No. 834, German National Research Center for Computer Science (GMD), Sankt Augustin, 1994.

[25] J. Liedtke. *Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse*. Deutsches Patentamt, München, February 1994. Patent application P 44 05 845.4.

[26] J. Liedtke. A virtually indexed cache with efficient synonym handling. Arbeitspapiere der GMD No. 829, German National Research Center for Computer Science (GMD), Sankt Augustin, 1994.

[27] H. Massalin. *Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, New York, 1992.

[28] MIPS Computer Systems, Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, 1st edition, 1991.

[29] Motorola Literature Distribution, Phoenix, AR. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[30] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software managed TLBs. In *20th Annual International Symposium on Computer Architecture*, pages 27–38, San Diego, CA, May 1993.

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

[32] K. W. Pinnow, J. G. Ranweiler, and J. F. Miller. The IBM system/38: Object oriented architecture. In D. P. Siewiorek, C. G. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, pages 537–540. McGraw-Hill, New York, 1982.

[33] J. Rosenberg, D. Koch, and J. L. Keedy. A massive memory supercomputer. In *22nd Hawaii International Conference*, pages 338–345, 1989.

[34] S. Russel, K. Elphinstone, J. Vochteloo, J. Ouyang, and G. Heiser. Page table management in single address-space operating systems. submitted to OSDI.

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

[36] R. L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, Maynard, MA, 1992.

[37] Inc. Sparc International, editor. *The Sparc Architecture Manual, Version 8.* Prentice Hall, Englewood Cliffs, NJ, 1992.