Address Space Sparsity and Fine Granularity

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

Abstract

To fully exploit the potential of large address spaces, e.g. 2^{64}-byte, the sparsity problem has to be solved in an efficient manner. Current address translation schemes either cause enormous space overhead (page table trees) or do not support address space structuring, object grouping and mixed page sizes (inverted page tables). Furthermore, an essential handicap of current virtual address spaces is their coarse granularity. It restricts the concept's relevance to low level OS technology. Without this constraint, mapping could be a vertically integrating paradigm, useful on all levels from hardware up to application programming.

Guarded page tables help solving both problems. They permit significant extensions of the current programming model without performance degradation: sparse occupation and coarse-grain (4K) pages can be handled by purely conventional hardware; fine-grain (down to 16-byte) pages without fine-grain aliasing become also possible using conventional cache and TLB technology combined with stochastically colored allocation. Unrestricted aliasing and unlimited user level mapping without performance degradation may become possible by hardware innovation.

1 Motivation

Upcoming 64-bit processors can lead to a quantum leap in the virtual address space paradigm. 2^{32}-sized spaces have proved to be a convenient and sufficiently powerful *local* tool for most problems. 2^{64}-sized spaces do not only permit to handle larger problems, but they offer new possibilities by means of *flat global* address spaces. This is a promising paradigm for distributed systems – especially for object oriented and/or persistent ones – and perhaps also for supercomputers. There are strong reasons that "unlike the move from 16- to 32-bit addressing, a 64-bit address space will be revolutionary instead of evolutionary with respect to the way operating systems and applications can use virtual memory." (*Koldinger, Chase, Eggers [8]*). Experimental single address space operating systems are for example Opal [4, 3] and Mungi [6]; a similar design is described in [2] and [15]. The latter system especially relies on rich *per page protection* facilities.

One serious problem coming up in this context is sparsity. A 2^{64}-byte address space will ever be sparsely occupied. (Otherwise, a $2^{\approx 32}$-space would do as well.) And after all what we have learned from history and from Murphy, we should expect that systems and applications in real life will tend to unlimited sparsity. Imagine, for example, the complete usenet or other huge distributed databases mapped into each single address space. Sparsity is also essential for architectures relying on anonymity as protection mechanism [17].

The critical point in supporting sparsely occupied huge spaces is the efficiency of the mapping mechanism. Currently used address translation schemes either explode in space requirements or do not support structuring. We need a scheme which is *tree-based, efficient (independent of the degree of sparsity)* and *as efficient as existing ones.*

Conventional schemes meet only two of these requirements: page table trees (also called multi-level page tables) do not support sparsity, whereas inverted page tables support solely the 1-page abstraction and

*GMD I5.RS, 53754 Sankt Augustin, Germany obstruct grouping of pages. In addition, the latter scheme restricts the OS architecture by high page sharing costs, and it does not permit mixed page sizes, what is essential for fine granularity.

Sparsity is strongly related to granularity: from the point of view of a 2^{64}-space, a $2^{12} = 4\text{K}$ page and a $2^4 = 16$ byte page do not look very different. Therefore, besides sparsity, we address granularity as well. Fine-grained address spaces potentially can have strong architectural impacts not only in obvious fields like object orientation, but also, for example, in code generation, data bases, distributed systems, supercomputers, file systems and multi media applications.

But whereas the sparsity problem can be solved solely by a better address translation mechanism without affecting other hardware components, fine granularity strongly interferes with the TLB and cache system.

2 Guarded Page Tables

The central idea of *guarded page tables* is to supplement each page table entry by a *guard*, a bit string g of variable length.

First, a page table entry is selected by the highest part of the virtual address upon each transformation step in the same way as with the conventional multi-level page table method. The selected entry however contains not only a pointer (and perhaps an access attribute) but also the guard g. If g is a prefix of the remaining virtual address, the translation process either continues with the remaining postfix or terminates with the postfix as page offset.

As an example, figure 1 presents the transformation of 20 address bits by 3 page tables. Note that the

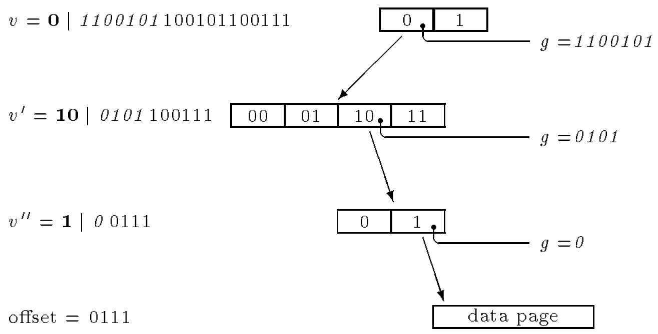


Figure 1: *Guarded Page Table Tree*

length of the guards may vary from entry to entry. Furthermore, page tables sizes can be mixed; all powers of 2 are admissible. The same holds for data pages, i.e., a mixture of 2-, 4-, . . . 1024-, . . . entry page tables and pages can be used.

Guarded page tables contain conventional tables as a special case: if a guard has length zero, a translation step works exactly like in the conventional mechanism. But in all cases conventionally requiring a table with only one valid entry, a guard can be used instead. It can even replace a sequence of such "single-entry" page tables. This saves both memory capacity and transformation steps, i.e., guards act as a *shortcut*.

The guarded page table algorithms can be implemented purely by software. However, the "non regular" operations of guard checking and stripping can be substantially accelerated by dedicated hardware. Parallel execution can further speed up the algorithm: checking 4 adjacent page table entries in parallel doubles the walking speed of nearly minimal trees [9, 12].

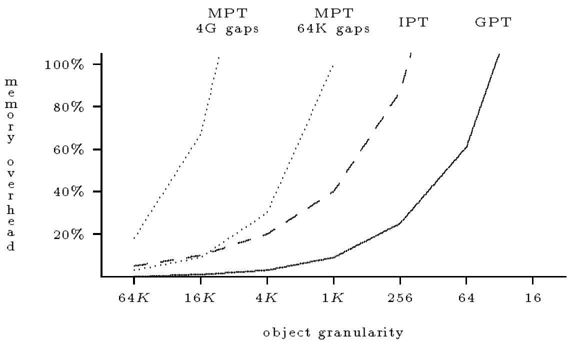
3 Sparsity

Guarded page tables have the advantages of conventional page table trees (support of hierarchical operations, sharing of subtrees) and outperform inverted page tables in most cases (see [10]). They require only a maximum of 2 page table entries per mapped page, regardless of how the pages are allocated and how wide the virtual addresses are. Without restricting the operating system architecture, guarded page tables solve the sparsity problem of 64-bit systems: the page table overhead is less than 1% of user data for any chosen mapping of 4 K pages. In contrast, conventional page table trees require overheads between 6% (compact) and 400% (very sparse) for 100 pages.

If only the sparsity problem is addressed, a unique page size and conventionally coarse-grained pages are sufficient. In this case, a guarded page table MMU can be integrated into a conventional processor without changing the TLB and cache architecture.

4 Fine Granularity

Figure 2 shows the worst case memory overhead depending on the object size for multi-level (MPT)[1], inverted (IPT) and guarded (GPT) page tables. Note that in practice the overhead might be lower. Nevertheless, guarded page tables seem to support fine granularity better than the other mechanisms. A more detailed cost analysis of the worst case and some other models is contained in [11].



100% overhead means that 1 M of user data requires one additional megabyte for page table data and fragmentation. The x-axis denotes the minimal object size. If it is e.g. 256, 257-byte objects are most expensive due to fragmentation. The page size is always chosen in such a way that the sum of page table data and fragmentation is minimal. Note that this usually leads to pages which are smaller than the minimal object size. MPT-entries are 4-bytes, page tables 4K; IPT entries are 16-bytes and a 50% occupied hash table is assumed; GPT entries are 8-bytes.

Figure 2: *Worst Case Memory Overhead*

Fine granularity does not only rely on the page table mechanism but also on the TLB and cache architecture. For example, it is difficult to handle small pages with todays favorite caches (virtually indexed + physically tagged). Therefore you may suspect performance degradation and/or higher hardware costs due to fine granularity.

Combined with stochastically colored allocation [10], more or less conventional caches (virtually indexed + physically tagged) and TLBs can fortunately be constructed which perform as well as conventional ones, in both the fine-granular and the coarse-granular case. Surprisingly, the combination permits larger caches than conventional technology, i.e., it will perhaps not only preserve system performance but increase it.

5 Visions

Fine-grained aliasing makes higher demands on the cache system. It either increases cache latency or requires additional hardware, e.g. base address caches [5] or a virtually tagged cache supporting synonyms [13, 14]. Therefore, this section is highly speculative: imagine that unrestricted aliasing is sufficiently cheap. What can we do with this mechanism?

[1] The MPT overhead strongly depends on sparsity. Therefore two scenarios are shown: in '4G gaps', the objects are located at consecutive 4G-aligned addresses, in '64K gaps' at consecutive 64K-aligned ones.

Experiments with user-level mapping are described in [1, 7]. But since, with guarded page tables, the obtainable granularity is in the magnitude of a program variable, we should even explore techniques to replace references by mapping, perhaps even parameter passing by mapping. For example, a compiler could assemble statically determinable sets of parameters and construct alias regions for them. Objects could be synthesized by mapping. Database queries could result in mapping the found objects instead of copying them. This could be extremely useful in distributed databases since it easily permits lazy evaluation of queries.

Extending the guarded page table mechanism permits user programs to manipulate mapping directly: efficient user controlled aliasing (for object synthesis, constructing alternate views or parameter passing) and call on reference become possible. The latter facility enables user programs to associate specific access semantics with address space regions (or pages), for example, 'delay upon read access' (variable value has not yet been computed), 'signal upon write access', 'remote object invocation', 'access by proxies' [16] or simply 'access protocol'.

The costs of unrestricted fine-grained aliasing are not yet clear, but the possibilities are amazing. Further work is required for exploration, including the construction of a really usable virtual machine, an appropriate kernel and experiments with code generators and applications.

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] J. S. Chase, H. M. Levy, M. J. Freely, and E. D. Lazowska. Sharing and protection in a single address space operating system. Technical Report 93-04-02, Univ. of Washington, Dept. of Computer Science, Seattle, WA, 1993.

[4] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Shared objects in a 64-bit operating system. In *OOPSLA*, pages 397–413, 1989.

[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] 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.

[7] 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.

[8] 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.

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

[10] J. Liedtke. Guarded page tables. unpublished, March 1994.

[11] J. Liedtke. Mmu impacts on system architecture. unpublished, June 1994.

[12] 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.

[13] 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.

[14] 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.

[15] T. Okamoto, H. Segawa, S. H. Shin, H. Nozue, K. Maeda, and M.Saito. A micro-kernel architecture for next generation processors. In *Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 83–94, Seattle, WA, April 1992.

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

[17] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous rpc: Low-latency protection in a 64-bit address space. In *Summer Usenix Conference*, pages 175–186, Cincinnati, OH, June 1993.