Page Table Structures For Fine-Grain Virtual Memory

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

GMD Technical Report No. 872

October 1994

*GMD I5.RS, 53754 Sankt Augustin, Germany

Abstract

Guarded page tables are tree-structured like conventional multilevel tables and support mixed page sizes, even down to 16-byte pages. They support huge address spaces, and perform well in time and space for arbitrary sparsity. Also, a mechanism for secure and fast modification of mappings by user-level software is described. This paper concentrates on page-table mechanisms; impacts on TLB and cache architecture are not discussed here.

Contents

1 Rationale

1.1 Motivation

From a low-level point of view, an object can be regarded as a piece of memory (holding object data and sometimes also access methods) combined with some sort of access control. This control is usually based on *virtual address spaces* and *access attributes* associated with pages or segments. The reader is assumed to be familiar with how to use the basic mechanisms for implementing higher level access control.

Earlier architectures, e.g., the Burroughs B5000 [1], Intel 432 [6] and IBM system /38 [18], try to support objects by using fine-grain *segments*, i.e. freely specifyable contiguous parts of the virtual address space. Since each segment is defined by a segment-table entry containing base address, size and access attributes, segments can be small and of different size. From an o-o point of view, this fine granularity is a convincing advantage of segments over conventional coarse-grained page-based virtual address spaces.

Most modern processor architectures avoid segments and implement virtual address spaces that are solely page based.

Examples are the SPARC [22] and MIPS [7] processors, DEC's Alpha [21] and the PowerPC [16] (which uses the term "segment," but in a different way than mentioned above, for large fixed-sized aligned regions of 256 MB, which are not suitable for implementing objects). A purely page-based architecture has its benefits: it avoids the segment-unit hardware, omits one step in the instruction pipeline, has no problems with huge segment descriptor tables, and supports grouping of pages and hierarchies of them.

Therefore the goal of this work is to combine the advantages of pages and segments:

> To construct a purely page-based architecture which efficiently supports fine granularity.

Describing a completely new architecture and comparing it with existing page- and segment-based architectures is far beyond the scope of this paper. Here, we concentrate solely on the page-table mechanisms.

1.2 Requirements

For adequate support of o-o systems, we need fine-grained address spaces. This means—

1. *Small pages must be supported.*
   "Small" in this context means something in the range between 16 and 64 bytes, that is, substantially less than 4 or 8K.

2. *Various page sizes must be possible, ideally all powers of 2.*
   Why not one unique small page size? This would be far too inefficient. Assume you have an equal distribution of 1024-, 256-, 64- and 16-byte objects. Further assume that you need 16 bytes of page-table data per page (independent of its size). When using only 16-byte pages, 50% of main memory is always occupied by page tables; otherwise only 5% is. For a working set containing one object per class, in the case of unique 16-byte pages, 85 TLB entries would be required; otherwise only 4.

3. *Different-sized pages must be able to be mixed freely in the address space.*
   Why not associate fixed-pages sizes to fixed address-space regions? If we did, we would have to use a 256-byte page for implementing a 150-byte object. Otherwise, we could compose the object out of one 128-byte and one 32-byte page. Furthermore, in many cases it is nice to allocate different-sized objects within a given region.

4. *Operations on a hierarchy of small and large regions should be supported efficiently.*
   This helps when dealing with multi-page objects or sets of objects, e.g., files, directories or databases. Loosely speaking, operations like lock/unlock or transfer into another address space should be fast, independent of the region size.

1.3 Conventional Page Tables

In the case of conventional *multilevel page tables*, a virtual address is translated in several steps. Each one works with a page table whose entries refer either to page tables at 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 four entries per page table and denote the address in the 4-ary system so that each page table decodes one digit of the address.

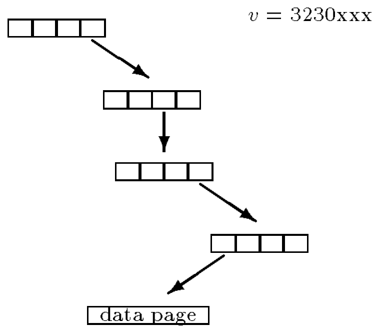


Figure 1: *Multilevel Page Table.*

The figure shows the process of translating the virtual address 3230`xxx`. The page number is 3230 and `xxx` is the offset within the page. The leftmost digit "3" is used 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 two address bits per step, 4-entry tables are unrealistic. In practice, page tables hold, for example, 1024 entries and have a size of 4K. Then 64-bit virtual addresses and 16-byte pages require a 6-level page-table tree. Comparison with our requirements shows serious problems:

- The space requirements for page tables can only be tolerated as long as only large numbers of adjacent pages are mapped. Even limited sparsity leads to very bad utilization (user data / (page tables + user data)):

pages	mapped at	tables	user	utilization
100,000	0,16...	412 K	1563 K	79 %
1,000	0,16...	24 K	16 K	40 %
1,000	0K,1K...	288 K	16 K	5 %
100	0,16...	24 K	1600 b	6 %
100	0K,1K...	48 K	1600 b	3 %
1	irrelevant	24 K	16 b	0 %

- Each address space requires at least 24K page tables. Therefore small address spaces are too expensive.

- Mixed page sizes are not really supported. The scheme allows only 16-byte, 16K, 16M etc. pages. This is far too coarse.

- Only 16-byte, 16K, 16M etc. regions can be manipulated efficiently. This is also far too coarse.

Inverted page tables, pioneered in the IBM /38 [18, 3] and also used in IBM 801 [2], PA-Risc [4] and Monads [19], consist of one entry per physical page frame. Each entry contains the virtual address of the mapped page and sometimes a pointer to the corresponding frame. A table entry is chosen by

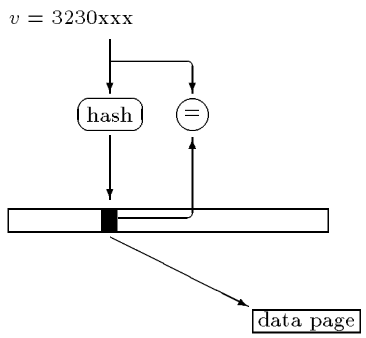


Figure 2: *Inverted Page Table.*

means of a hash function which uses all bits but the offset part of the virtual address v (see figure 2). If the virtual address stored in the table entry matches v, the physical page is located. Otherwise rehashing is required or a page fault is signalled.

In contrast to simple inverted page tables, *hashed page tables* [5] additionally contain a physical page address in each table entry, thus permitting page aliasing.

Since the size of inverted or hashed page tables depends only on the physical memory size, sparsity in the virtual address space does not increase the page table overhead. Nevertheless, comparison with our requirements shows serious problems:

- Mixed page sizes are impossible. Sixteen-byte granularity would lead to a fixed memory utilization of only 50% (16-byte pages, 16-byte IPT entries) and a tremendous number of TLB entries. Furthermore, operations on larger logical pages are too expensive: 256 IPT entries would have to be modified for changing the access attribute of only one logical 4K page.
- Due to the lack of tree structure, operations on regions are extremely expensive.
- Small address spaces are too expensive.

We conclude that the chance of overcoming the quantitative problems related to multilevel page tables is higher than that of eliminating the structural restrictions of inverted or hashed page tables.

2 Guarded Page Tables

The main problem with multilevel page tables is sparsity: we need huge amounts of page table entries for 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- 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 lead to page faults. As shown in figure 4, we can omit the two page tables and also skip the associated translation steps. Instead, whenever entry 3 of the top-level page table is reached, we have to check whether "23" is a prefix

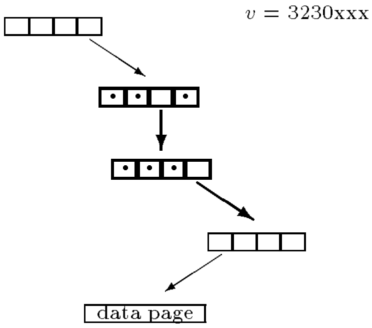


Figure 3: *Sparse Multilevel Page Tables.*

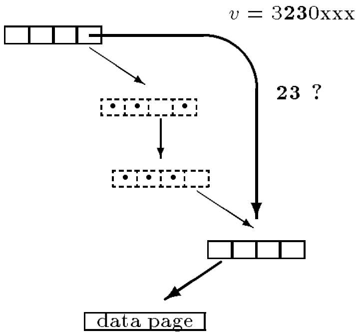


Figure 4: *Guarded Page Tables.*

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 augmented by a bit string g of variable length, which is referred to as a *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 page table or the data page. All powers of 2 are admissible; e.g., 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 multilevel tables as a special case: if the actual guard has length zero ($g = \emptyset$), a translation step works exactly like in the conventional mechanism.

2.1 Simple Guarded Translation

This section more precisely describes guarded entries and the translation process which maps virtual to physical addresses. Detailed descriptions can also be found in [14, 8, 9, 13].

In case of conventional multilevel tables, the translation process regards the virtual address v as

u	v'

Figure 5 shows one translation step: u is used for indexing the page table and v' is either the remaining virtual address for the next-level translation or the offset part of the address.

A guarded translation step is described in figure 6. Again, u is used for indexing the page table. The addressed entry specifies a guard g of variable size, e.g., possibly empty. It is checked whether g is a prefix of the remaining virtual address bits ($w = g$). When matching, the remaining v' is either used for the next-level translation or as the offset part. Guards work as shortcuts, since not only u, but both u and w are stripped off the virtual address in one step, and no table is necessary to decode w.

Note that the width of u (determined by the page table size), may vary from step to step. The width of w may also differ from entry to entry and is defined by the width of the entry's guard g.

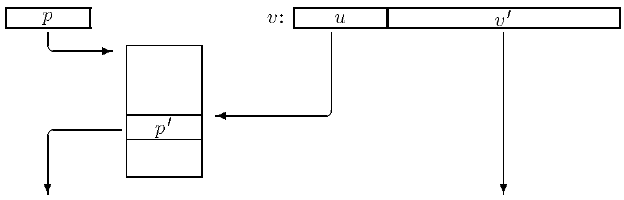


Figure 5: *Conventional Translation Step.*

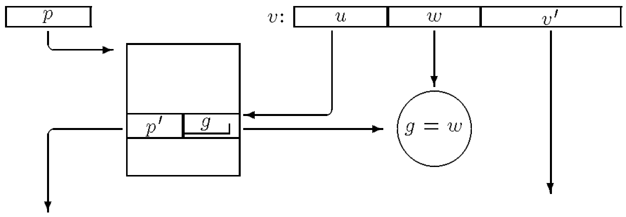
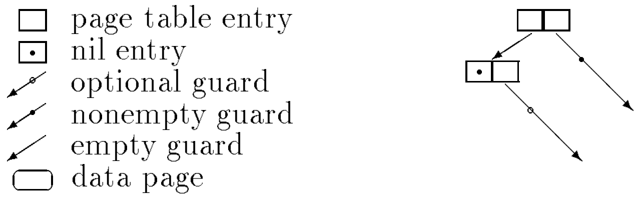


Figure 6: *Guarded Translation Step.*

When using only standard ALU operations for decomposing the virtual address, checking the guard prefix and synthesizing the physical address, a translation step would require multiple cycles. It seems reasonable to speed up this by special hardware, e.g., hardwired bit extraction and **xor** masked by guard length.

2.2 Time and Space

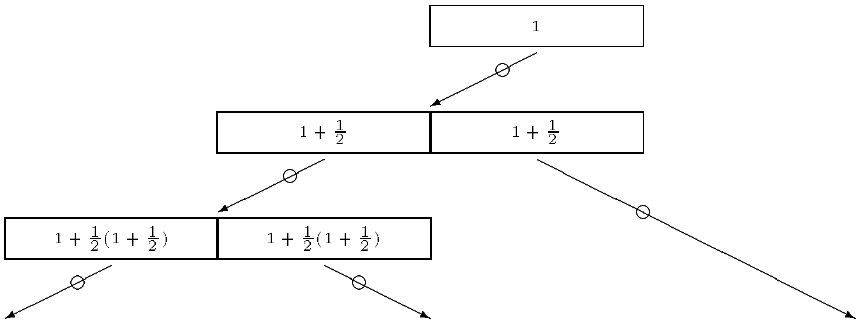
This section informally presents some results concerning worst-case space and time costs of guarded page tables. Detailed mathematical discussion, definitions and proofs can be found in [9, 13].
For page table tree diagrams we use the following notation:



2.2.1 Small Trees

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—

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

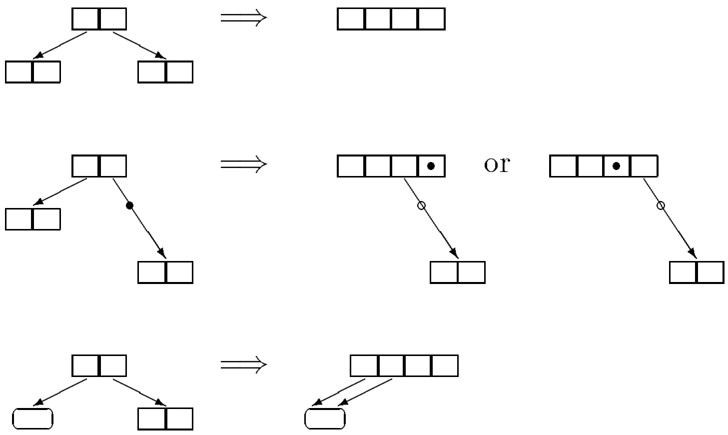
Therefore, $2k$ entries are enough for k reachable pages. This upper bound of the tree size is independent of the address-space size and of the set of reachable pages. We call this property *small* and try to hold all trees small in the following.

2.2.2 Small Trees with $n/2$-Depth

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

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 $n/2$, rounded up. We simply take the normal tree and transform all nodes which do not already decode two or more bits, i.e., which have one or two empty guards. We transform these binary nodes with their direct successors into quaternary nodes which decode 2 or more bits:



No other cases require transformations. Since none of the specified transformations increases the tree size, the resulting tree is also small.

2.2.3 k-associative Translation

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

Unfortunately, simply applying the above transformation idea once more, i.e., transforming 4-entry to 16-entry nodes, can make the tree up to four times larger. This problem can be overcome by *k-associative guarded translation* which essentially means to check a cluster of k entries in parallel. Figure 7 illustrates the 4-associative case.

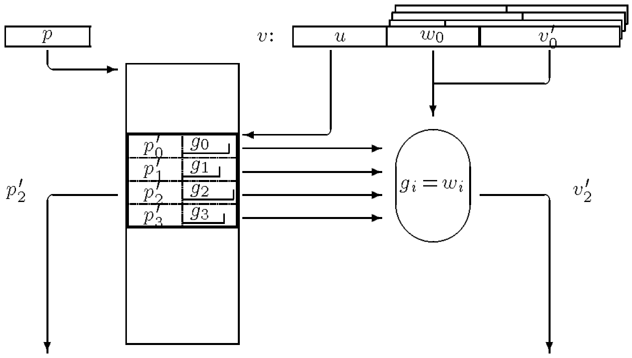
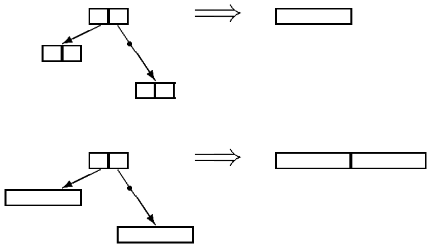


Figure 7: *4-associative Guarded Translation Step.*

The basic hardware support for k-associative translation is a k-fold guard check unit and a cache data path which can deliver k entries in parallel. For $k = 4$ and 8-byte entries, a 32-byte path is required (as in the PowerPC 601).

The key point of k-associative translation is that it permits "weakly asymmetric" distributions of entries in a page table and therefore "flattening" of subtrees, in case of 4-associativity transformations like



(k-long boxes are used to denote quadruples of entries which must be handled 4-associative.)

> A detailed description of k-associative translation mechanisms and the possibilities they offer is not quite simple and therefore omitted here. The interested reader can find it in [9].

It is important that k-associative translation is sufficient for further reducing the maximum depth of a tree without enlarging it. We can always construct a small translation tree with maximum depth $\lceil \frac{n}{m} \rceil$, if 2^{m-1}-associative translation is available. More generally, a 2^{m-1}-associative translator instead of a simple one permits shortening all paths in a given tree by a factor m.

For example, small 8-associative guarded page-table trees allow decoding n bits in about $n/4$ steps or less. In practice, even a 4-associative translator is sufficient. In this case, the tree may increase in size by up to 14%.

2.3 Optimizations

For practical reasons, we consider 64-bit virtual addresses. Sixteen-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 that a 4-associative translator is used; but 15 are still too many steps.

2.3.1 Side Entry-Point Caching

Fortunately, we can drastically reduce the average number of steps by caching side entry points into the translation tree. We divide the address space into equal-sized regions and cache the optimal entry points of accessed regions (see figure 8).

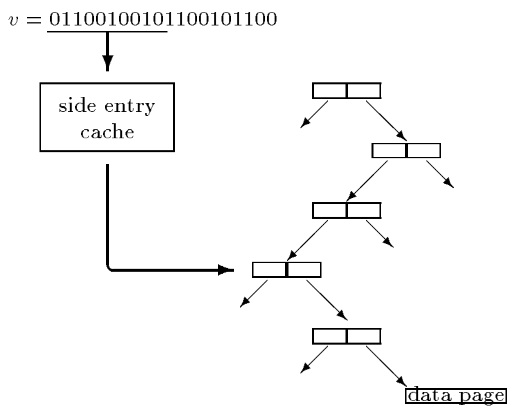


Figure 8: *Side Entry Translation.*

Whenever a complete translation process comes to a point where for the next step fewer 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 do a lookup in the side entry cache using the same region number and find the cached entry.

Of course, the side-entry cache can be built in hardware as a second level TLB. But there is a different solution operating without specific hardware: simply store the side entries in a special area of the virtual address space. Then the data cache automatically works as a side-entry cache.

If we use, for example, 16MB regions, a side-entry cache hit reduces translation to at most 3 steps for 4K 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. When a program accesses data in 100 different regions, about 2.5% of a 64K cache would be required for side entries.

2.3.2 Fast Trees

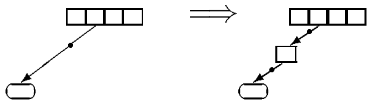
Some system architectures suggest more specific optimizations. The simplest technique is *isolation*. For example, one could use a 256-entry guarded page table at the highest level and choose some of these 2^{64-8} byte regions to hold only one page. The corresponding address translation then takes only one step. Multi-page objects can be reached by two steps, i.e., one additional page table.

A further method is to use two trees, one covering all mapped pages and another one holding only pages which need to be reached very fast.

Some systems use special address translation (fewer levels) for accessing kernel data [17]. Guarded page tables permit this more generally.

2.3.3 Restricted Guards

The maximum guard length together with the physical address width determine the space requirements per page-table entry. Sixteen bytes (64 + 64 bits) are certainly enough, but in case of limited physical addresses ($\approx$ 32-bit), 8 bytes per entry may be sufficient in practice. Since this restricts the guards to approximately 32 bits, it sometimes requires additional page-table entries as elongations:



Nevertheless, even in the worst case only $3 \cdot 8 = 24$ instead of $2 \cdot 16 = 32$ bytes per page are required.

Surprisingly, the maximum depth of the tree will *not* increase. The reason is that long guards can occur only in short paths. A more general and more detailed discussion can be found in [13].

Note that reducing guard length and entry size directly reduces the required hardware costs, or enables higher associativity at the same costs. Even 15-bit guards look reasonable for 64-bit addresses, and 7-bit guards for 32-bit addresses.

3 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 as follows:

- *Aliasing*
  Mapping a virtual storage object onto another virtual storage object. This is used, for example, for object synthesis, but also for constructing alternative views or simply for parameter passing.

- *Call-on-Reference*
  Calling a user-defined procedure upon access. This is to associate 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 proxy" [20] or simply "access protocol."

Combining the two methods allows an efficient realization of distributed memories since access to potentially remote objects is feasible by means of local object invocation. Memory accesses are performed directly in the local case and algorithmically in the remote case. Distinction is by hardware.

Of course, the OS can realize this by software. Since, however, mapping is likely to be modified very frequently with the mentioned methods, it is desirable to have a facility for modifying mappings directly from user level without system calls. Obviously, this option must not affect security paradigms of the OS.

Therefore, we add a type τ to each page-table entry. Traditional entries have τ = translate. For secure user-level mapping, new types and a special instruction are introduced.

3.1 τ = alias

Figure 9 shows a translation step for an alias typed page-table entry. The

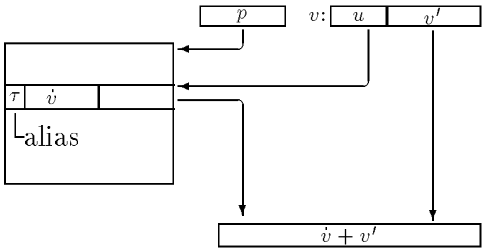


Figure 9: *Alias Translation Step.*

pointer is interpreted as a *virtual* alias address $\dot{v}$. If address transformation encounters an alias entry on any level, $\dot{v}$ is added to the remaining address v' not decoded so far.

Address transformation is restarted with the resulting $\dot{v} + v'$ which is a virtual address of full width. Consequently, the entire virtual region covered by the alias entry is mapped *virtually* to another virtual region (see figure 10). This has to be distinguished from conventional *physical* aliasing (see figure 11), where several physical pointers meet on a page table or data page. Unlike *physical* aliasing, *virtual* aliasing is independent of physical-memory

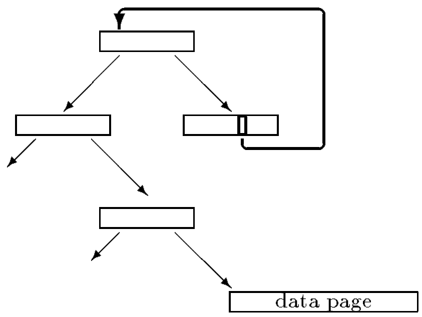


Figure 10: *Virtual Aliasing.*

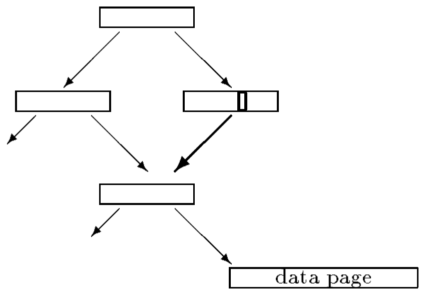


Figure 11: *Physical Aliasing.*

mapping. It enables modifications of mapping by user-level software, independent of current physical-memory allocation and paging; it remains impossible to penetrate protection boundaries and already-set access attributes cannot be weakened.

3.2 τ = call-on-reference

For the type call-on-reference, the pointer is interpreted as a virtual address $\ddot{v}$ of a procedure (see figure 12). If address transformation encounters a call-on-reference entry, the accessing operation is aborted and the specified procedure is called instead. It gets the current instruction counter and the initiating virtual address v as input parameters. Just as with a page fault, the initiating instruction can be restarted.

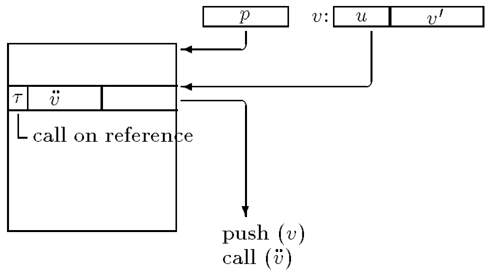


Figure 12: *Call-on-Reference Translation Step.*

Call-on-reference mapping assigns algorithms to address-space regions. They can skip the initiating instruction or emulate it. They may also operate like a page-fault handler, e.g., remapping the address space region in question by alias and restarting the instruction.

Processors with a strict load/store architecture can improve emulation by additionally passing access type (read or write), operand size (byte, word,...) and operand value or destination register ID as parameters to the associated handler.

A further extension is this: Place an additional user-defined value in each call-on-reference entry which is also passed as a parameter to the invoked procedure. This extra argument can be used for reducing the number of handlers and/or the costs for analyzing the event. If the entry fields are wide enough, we can use the guard field (obsolete in call-on-reference entries) for storing the extra argument. This avoids increasing the entry size.

3.3 The `map` Instruction

The instruction set of the processor is extended by the *non-privileged* instruction `map`. This enables user-level software to modify alias and call-on-reference entries directly. The instruction

map (base, size), (type, pointer)

selects a page-table entry as destination by specifying base and size of an address-space region. Provided that the destination entry exists, is accessible from the current mode (user or kernel), already has type alias or call-on-reference, and the new type is also alias or call-on-reference, the new type and pointer values are loaded into the destination entry. Otherwise, a page fault is signaled.

In this way, user-level software can modify corresponding entries, switch between alias and call-on-reference and change the alias or handler address. However, the abovementioned preconditions ensure that virtual-to-physical mappings cannot be modified, that protection boundaries cannot be penetrated and that entries cannot be generated by circumventing the OS.

Creation and deletion of corresponding entries together with the necessary modification of the page table tree is done by the kernel.

Note that alias and call-on-reference entries can be implemented as long-lived objects, since swapped-out entries lead to page faults both upon normal access to the address-space region and upon `map` access to the entry.

4 Related Problems

Small and different-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 region attributes without flushing the complete TLB?

Ongoing research [15, 11, 10, 12] deals with these questions. It is yet too early for definite answers, but we are not pessimistic.

Acknowledgements

Martin Gergeleit proposed improving instruction emulation in strict load/store architectures, as described in section 3.2; Marc Shapiro proposed helping call-on-reference handlers by using an additional user-defined parameter, also discussed in section 3.2.

References

[1] Burroughs Corporation, Detroit, MI. *The Descriptor — a Definition of the B5000 Information Processing System*, 1961.

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

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

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

[5] 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.

[6] Intel Corporation, Santa Clara, CA. *iAPX 432 General Data Processor Architecture Reference Manual*, revision 3 (advance partial issue) edition edition, 1982.

[7] G. Kane and J Heinrich. *Mips Risc Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[8] 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.

[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. *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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

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