A High Resolution MMU for the Realization of Huge Fine-grained Address Spaces and User Level Mapping

Jochen Liedtke
German National Research Center for Computer Science (GMD) [1]
jochen.liedtke@gmd.de

GMD Technical Report No. 791

October 1993

[1]GMD SET-RS, Schloß Birlinghoven, 53757 Sankt Augustin, Germany

Abstract

This paper describes a Memory Management Unit (MMU) which can be used for implementing huge fine-grained address spaces. Granularities down to 16-byte pages seem to be possible. Furthermore, a mechanism is described whichs permits fast and secure mapping operations on user level.

Contents

Chapter 1

A high resolution MMU

Note: this chapter corresponds to patent application "Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse", Deutsches Patentamt P 43 15 567.7 (filing date May 10, 1993).

The invention relates to a procedure and equipment for transforming a virtual address into a real address. An application of the invention is the memory management unit (hardware), also called MMU transforming a.o. the address in the virtual memory into an address of the physical or real memory.

Modern operating system developments (e.g. Mach, L3), the ideas of object orientation (many small objects) and especially the emergence of processors with large address spaces (64-bit addresses) have indicated important shortcomings of the MMUs available so far. These shortcomings are in particular

- too coarse and uniform granularity;
- immense costs for sparsely occupied address spaces (2^{64}-byte spaces will always be occupied sparsely!);
- insufficient support of hierarchical structures.

Brief description of the high resolution MMU (working title):

- 64-bit-wide virtual addresses (the methods can be used in the same way for 128-bit or even wider addresses);
- various page sizes starting with 8 bytes (8, 16, 32..., 1K, 2K, 4K...), combinable in the address space;
- for any occupancy of the address space, a maximum of 16 to 32 bytes of management information per allocated page (depending on the size of the virtual and the real address space).

1.1 Conventional MMUs

MMUs (Memory Management Units) are the hardware basis for virtual memories, address spaces, paging and protection mechanisms. Consequently, they are important to all applications which process large data sets (in particular in distributed systems or supercomputers) and/or require security characteristics on objects of complex structure.

Conventional MMUs typically divide virtual address spaces into pages of 4 KB or 8 KB. Mapping virtual addresses onto real ones is done either by means of multilevel *page tables* or an *inverted page table*. Usually, address transformation is accelerated by a TLB (Translation Lookaside Buffer) or a virtual cache.

Both usual multilevel page tables and inverted page tables are suitable to a restricted extent only or they are even not suitable at all for the realization of really small (16 bytes) and/or huge sparsely occupied address spaces (2^{64} bytes or larger). Multilevel page tables need too much space (thousand times the user data seems to be quite realistic), inverted page tables disable all hierarchical operations (sharing, mapping, copy on write...) and require very complicated and varying hash functions in the situation described above.

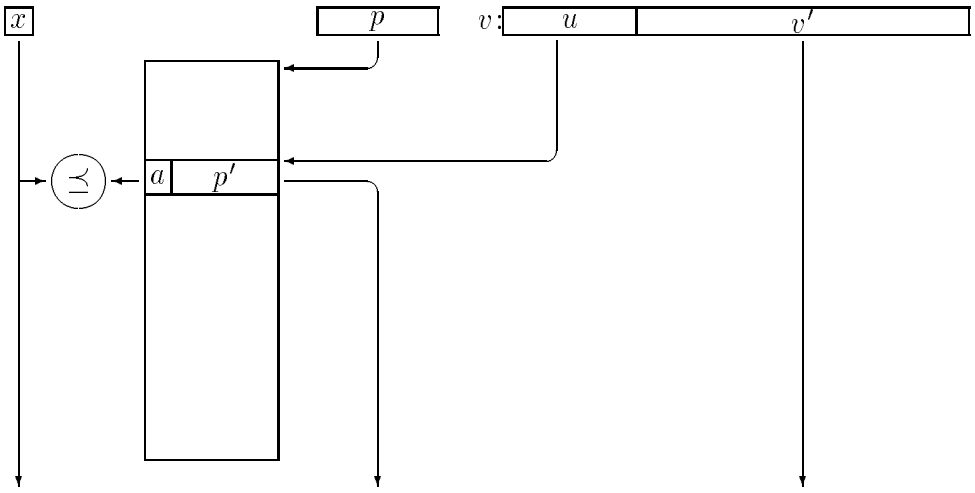
1.1.1 Conventional Page Tables

In the case of conventional page tables, a virtual address is transformed in several steps. In this case, each level works with a page table whose entries refer either to page tables of the next level or (for the last level) to data pages. In this case, page tables and data pages are usually of fixed sizes 2^i.

We now consider a transformation step of a virtual (binary) address v for an action[1] x by means of a page table with the address p. For this purpose,

[1]For many computers, actions consist of the read/write or execute operation and the user/kernel mode of operation. The access attributes which permit certain actions (in the extreme case, all actions or none) are constructed correspondingly. The set and semantics

v is split into a higher part u (consisting of a specific number of higher bits) and a lower part v' (consisting of the lower bits). Using u, we then select an entry of the page table which includes an access attribute[1] a and a new address p'.

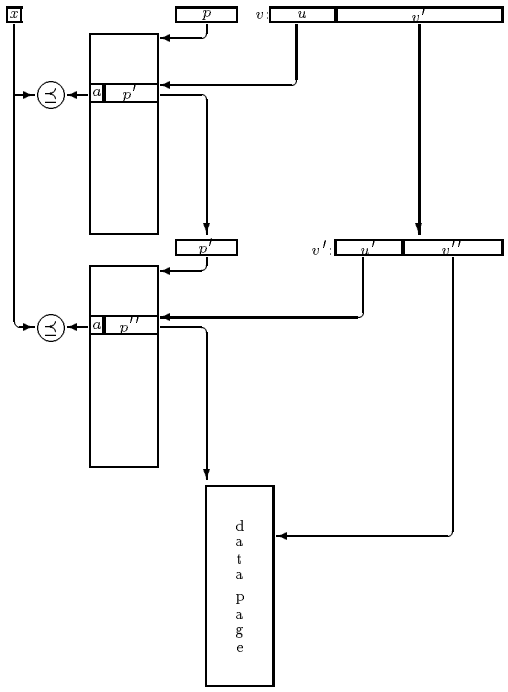


If the access attribute prohibits the action ($x \succ a$), transformation is aborted and *page fault* is signaled.

If the action is valid ($x \preceq a$), x, p' and v' are passed to the next level transformation as input parameters. It is to be noted that v' is shorter than v by the bit width of u. As soon as the last level is achieved, p' points to the beginning of the data page and v' is the offset within the page. A two-level transformation produces the following picture:

---

of concrete actions and access attributes and the method of checking action against attribute is irrelevant from the present viewpoint. The crucial point is that a circuit decides only on account of action x and access attribute a whether to allow or to abort the action.

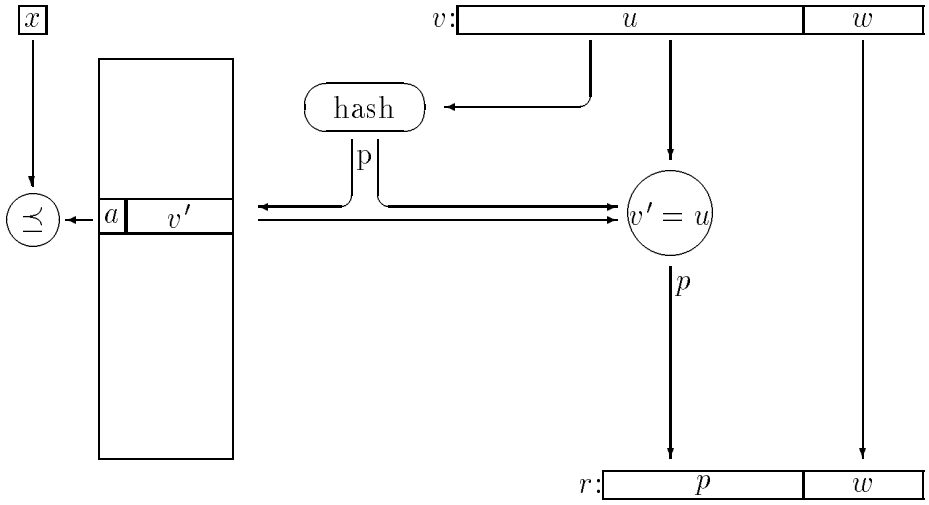


If 64 bit virtual addresses are to be transformed in this way and if the minimum page size is to be 16 bytes, this can be achieved, for example, by means of a ten-level transformation (4 KB per page table). However, sparsely occupied address spaces thus require intolerable management costs. 1024 16-byte pages can be allocated in such a way that 36 KB of management data are necessary per 16 data bytes which are 0.04% user data! By using a 60-level transformation (8 bytes per page table) management costs decrease to the minimum, namely 400 bytes, which are however still only 4% user data. In addition, a 60-level transformation process would be too time-consuming.

If we consider smaller address spaces with 32 bit addresses, for example, the corresponding values are getting better, but they are still intolerable. A 14-level transformation (16 bytes per page table) would thus produce 11% user data.

1.1.2 Conventional Inverted Page Tables

Inverted page tables consist of one entry per page frame of the real memory which contains the virtual address of the allocated page of the virtual address space. Access is performed with the aid of a hash function:



Upon transforming the virtual address v into the real address r, the lower part w is adopted directly. The higher part u is mapped to a value p by means of the hash function. This value identifies the presumable page frame in the real memory and is used for indexing the inverted page table. If the corresponding entry contains the correct virtual address u, it is a hit. Otherwise (not shown in the diagram) further page frames have to be examined by means of rehashing or linking until getting a hit or a page fault.

Since, for inverted page tables, management information depends only on the size of the real memory (and the page size), but not on the size and number of virtual address spaces, space problems do not occur.

Nevertheless, three striking disadvantages make the method useless for fine grained huge address space:

1. All pages must be of equal size[2], i.e. they must be of minimum page size. A mixture of small pages (16...256 bytes) and medium pages (2...16 KB) would however be more favourable almost in any case.

2. In the case of small pages, large real memories and huge address spaces, the used hash function must be extremely good to guarantee a sufficiently high hit rate. Procedures which change the hash function dynamically such as universal hashing are likely to be necessary. Hardware and software overhead would be immense.

[2] We could admit several page sizes by using a specific hash function and inverted page table for each size. Without a fixed division of the address space, however, several sizes (in the extreme case all sizes) would have to be tested sequentially upon each address transformation as a rule. On account of the table size, a parallel implementation seems to be hardly possible.

3. Sharing pages or entire address space regions is not possible. The hierarchical operations (lazy copying, copy on write, mapping, locking) required by modern operating systems are not feasible with acceptable efficiency.

1.1.3 Conventional TLBs

For cost reasons, the page table tree cannot be parsed upon each memory access by a program. This overhead is avoided with the aid of a specific cache for address transformation, a Translation Lookaside Buffer (TLB). Generally, more than 90% of all address transformations are done by TLB hits at no cost. Only in the case of a TLB miss, the page tables are parsed.

Conventional TLBs typically hold 32 to 128 entries each of which describing the address transformation of a page. Some of them are fully associative, but often they are only 4-way-associative.

Virtually addressed caches are sometimes used instead of or in addition to these TLBs.

1.2 Task

Construct an MMU which allows the realization of huge, sparsely occupied address spaces (2^{64} bytes or more) of an as fine as possible granularity with acceptable memory and time cost. The advantages of the tree-like page tables (sharing of subtrees, support of hierarchical operations) described in 1.1.1 are to be maintained.

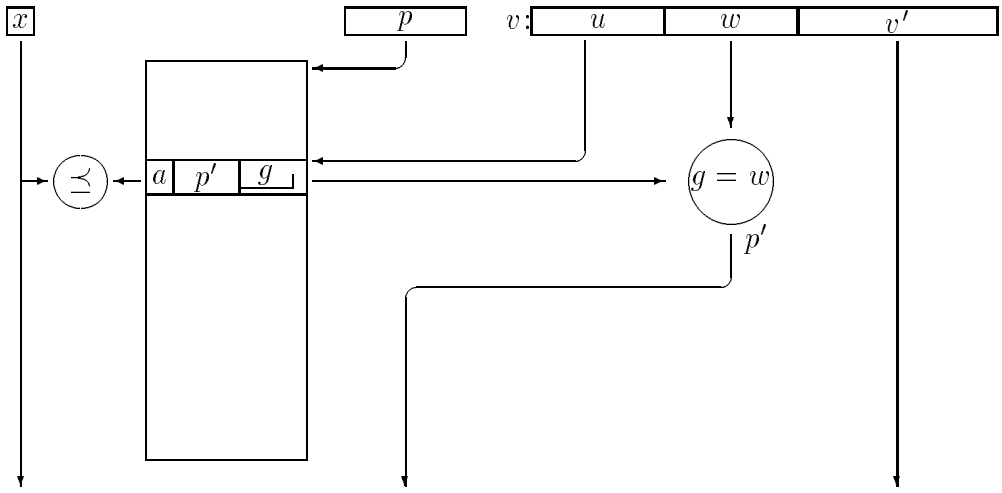
Granularity should not be uniform, i.e. the page size should potentially vary in the address space from position to position. The pages are always aligned, i.e. $v \bmod 2^i = 0$ always holds for the virtual starting address v of a page of size 2^i.

A ratio of user data to management information (page tables) of 1:1 is considered to be still acceptable for the extreme case (only pages of minimum size which are distributed randomly). The ratio should improve drastically with increasing page size. The time cost of conventional MMUs is regarded as acceptable for the new MMU.

1.3 Guarded Page Tables

1.3.1 Simple Guarded Page Tables

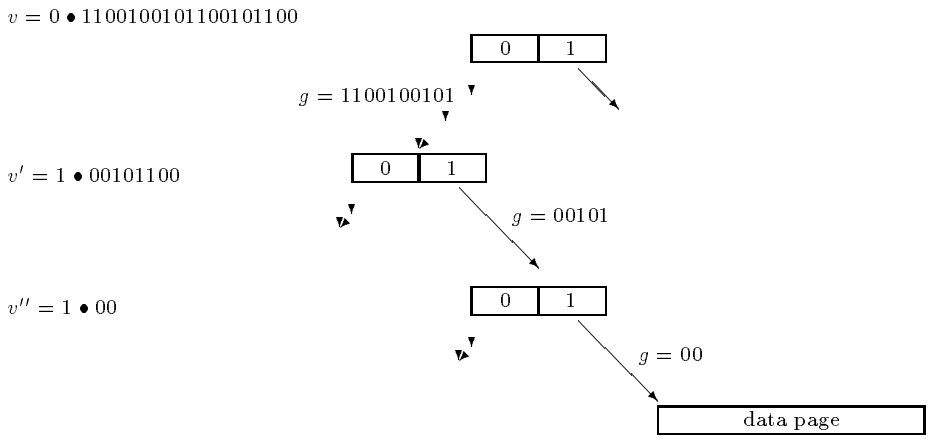
The central idea of *guarded page tables* is the supplementation of each page table entry by a bit string g of variable length which is referred to as *guard*.



First a page table entry is selected by the highest part u of the virtual address upon each transformation step in the same way as with the conventional method, and the action x is examined against the access attribute a. However, the selected entry contains not only access attribute and pointer but also the guard g. By means of the current length of g, the remaining virtual address is split into a higher part w (of equal length as g) and a lower part v'. Then we check whether $g = w$ holds. In the case of inequality, the transformation is aborted with page fault, in the case of equality, it is continued with x, p' and v' in the next level and/or $p' + v$ is delivered by the last level as a real address.

It is to be noted that the length of the guards may differ from entry to entry. Their current length is therefore contained in the page table entry and is coded as a length field or in another suitable way. For guards of length 0 ($g = \emptyset$), the procedure works just as the conventional one. But in all cases where conventional page tables with exactly one occupied entry are required, a guard can be used instead. A guard can even replace a sequence of such "one-element" page tables. This saves both memory capacity and transformation steps, i.e. guards act as a *short cut*.

As an example, we present the transformation of a 20-bit address which uses 3 binary page tables (2 entries per table):



Each page table entry contains not only the pointer to the next level table (in the part denoted by p), but in addition a size specification s for this object. In the case of page tables, s refers to the number of entries; all powers of 2, i.e. 1,2,4,8,..., are admissible. The length of u is obtained from the current page table size.

As shown in [3], on account of the thus enabled flexible tree structure, page table trees can be constructed by means of guards in such a way that a maximum of 2 page table entries is required per data page, independently of address space size and page size!

Together with the data pages varying in size, more than 50% user data should be thus attainable in almost all cases[3]

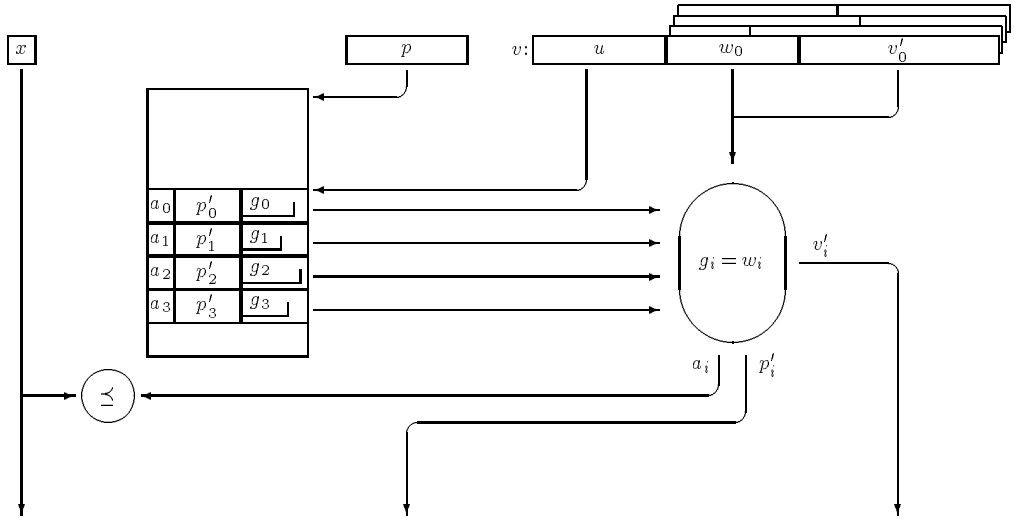
As also shown in [3], address transformation trees can be constructed such that a maximum of $n/2$ levels is necessary for an n-bit address transformation without exceeding the above storage requirements.

Consequently, a maximum of 30 levels is necessary for 64-bit addresses, for 32-bit addresses a maximum of 14 levels, to obtain 16-byte pages.

[3]With 8 bytes per page table entry, guards of a maximum length of 30 bits can be used. A maximum of 16 bytes of management information per page are thus necessary for 32-bit addresses. 64-bit addresses might need longer guards in some cases, then they are realized by an additional entry of 8 bytes. In the worst case (never more than one page per 2^{31} bytes and only 16-byte pages), 40% of the data will be user data.

1.3.2 k-associative Guarded Page Tables

In the case of *k-associative guarded page tables*, not one page table entry is selected in each step, but k entries.[4] The page table entry does not consist anymore of s simple entries, but of s/k clusters which consist of k simple entries each. u is correspondingly shorter and selects a cluster.[5] With $k = 4$, we obtain, for example:



Here, the 4 selected entries are read in parallel and analyzed (in parallel). It is to be noted that the guards g_i usually differ with regard to their lengths, i.e. for the various comparisons, the bit string reduced by u is divided into different pairs w_i, v'_i.

If no hit occurs (all $g_i \neq w_i$), transformation is aborted with page fault. In the case of exactly one hit $g_i = w_i$, the access attribute a_i is checked against the action x, and, if it is true, x, p_i and v'_i are passed to the next level and/or returned as a real address. In the case of several hits, the result is not defined.

As shown in [3], 8-associative guarded page tables enable an n-bit address transformation within a maximum of $n/4$ steps with a maximum of 2 (simple) entries per data page.

Consequently, for 64-bit addresses, a maximum of 15 levels is required, for 32-bit addresses 7 levels, to obtain 16 byte pages.

[4] For the highest possible speed, we therefore need k parallel units and parallel data paths for k page table entries.

[5] If $k > s$ holds for a page table, k is reduced for this transformation step, i.e. we work only in an s-associative way.

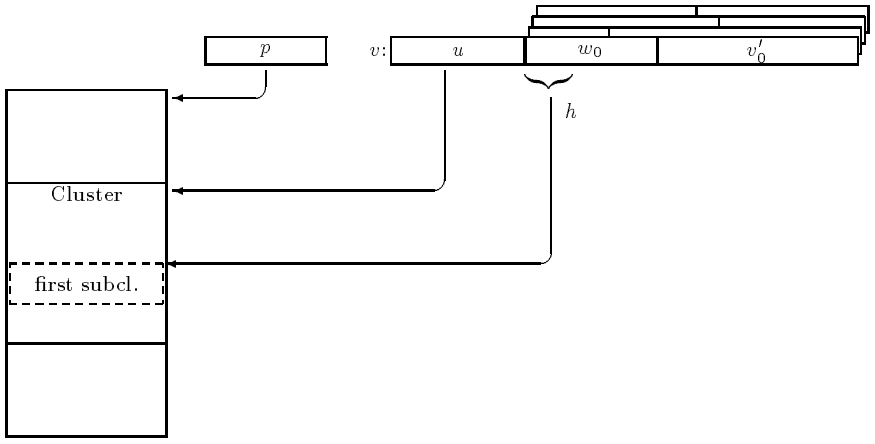
1.3.3 k/j-associative Guarded Page Tables

k/j-associative guarded page tables show the same semantics as k- associative ones. However, they require only k/j-fold parallelism for realization.[6] (In this case, k should be evenly divisible by j. In addition, these two should be powers of 2.)

The clusters comprising k entries are divided into j subclusters (which are contiguous and of equal size).[7] To maintain the semantics of k-associativity, the transformation operation is performed sequentially (k/j-parallel) on different subclusters until a hit occurs or until all j subclusters are processed. If no hit is found, address transformation is aborted with page fault, in the case of a hit, the procedure is continued as described in 1.3.2.

Obviously, this method is only efficient if possible hits are often found at the very first attempt. To obtain a hint, after removing u, the highest $\log_2(k)$ bits of the remaining virtual address are taken as a hint. (However, the hint bits are also used further for building w_i.)

In the case of *simple* k/j-associative guarded page tables, we begin with the subcluster addressed by h/j. The further sequence can be determined by incrementing h/j modulo j.[8]

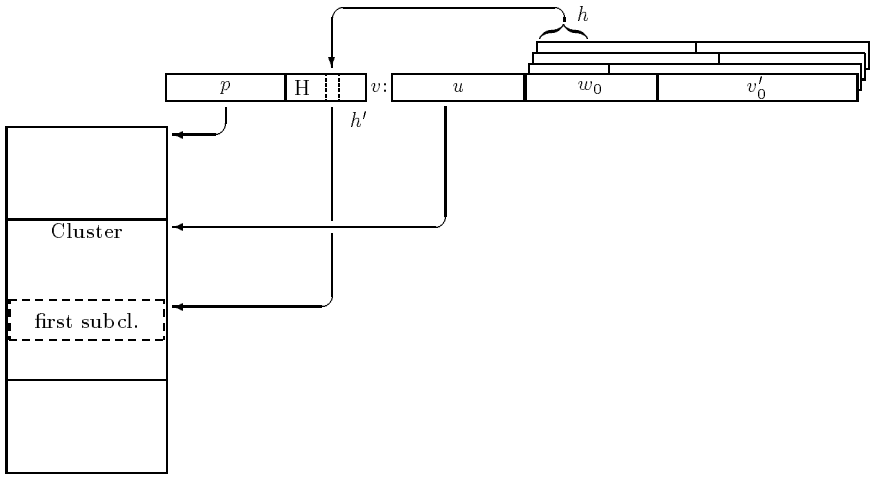


[6] For highest possible speed, we therefore only need k/j parallel units and parallel data paths for k/j entries.

[7] If $k > s$ holds for a page table, k is reduced to s for this transformation step. If $s \leq k/j$, work is s-associative, otherwise it is s/j'- associative. j' is selected such that the length of the subclusters remains unchanged, i.e. $k/j = s/j'$. This will make no problems if s,k and j are powers of 2.

[8] Other sequences are equally possible, e.g. from 0 to $j-1$ with omission of h/j. They have nearly no influence on efficiency.

In the case of k/j-associative guarded page tables *with hint*, each p (in page table entries, in the root and in the TLB's) is expanded by a field H with k hint elements. (This is relatively small with $k \log_2(j)$ bits.) One begins with the subcluster addressed by $p.H[h] = h'$ in each step.



If a hit is not achieved in the subcluster addressed by h' but in another subcluster of number h'', the hint element is reloaded: $p.H[h] := h''$. Consequently, hints adjust automatically.

As shown in [3], 8/2-associative guarded page tables with hints allow an n-bit address transformation within a maximum of $n/4$ steps for a maximum of $2\frac{1}{8}$ entries per data page. In general, they need the same time for this as 8-associative guarded page tables do, but they require only 4 parallel units instead of 8 and correspondingly narrower data paths.

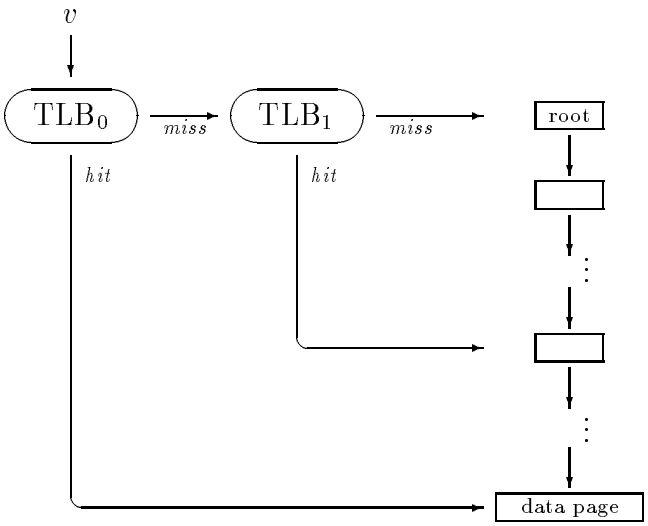
1.4 TLBs for Guarded Page Tables

To achieve a sufficiently fast address transformation, guarded page table translators have also to be supported by TLBs. The specific problems are as follows:

- different page sizes;
- larger working sets (more pages because of smaller granularity);

- deeper trees with huge address spaces (depth 15 for 60 bit address transformation), i.e. higher cost for TLB miss.

A multilevel TLB is used for solution:



TLB_0 is a more or less conventional TLB on page basis or a virtual-addressable cache; a hit delivers the corresponding real address directly. TLB_1 operates on larger regions (e.g. 16 MB) so that entering into the page table tree transversally is possible upon a near miss (TLB_0 miss and TLB_1 hit), and only a small part of the tree has to be parsed.

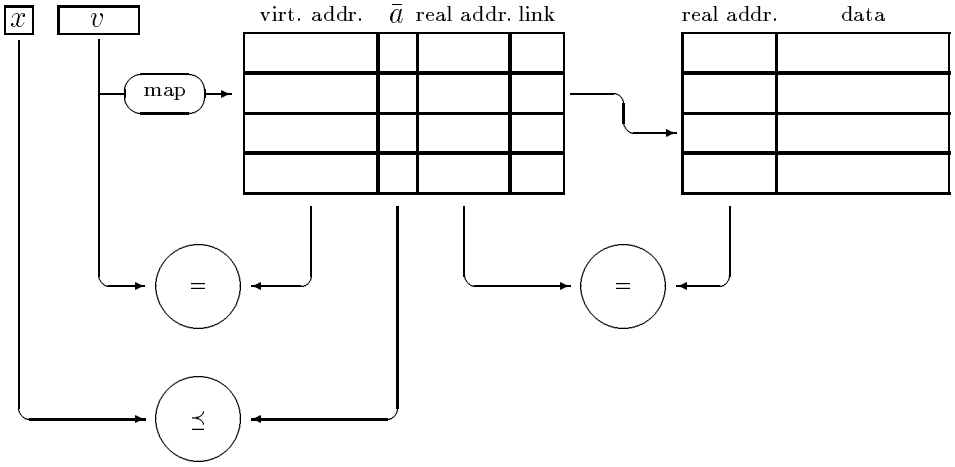
If required, this method can be extended in a natural way by further TLB_i levels.

1.4.1 TLB_0

The pages varying in size are an essential problem of TLB_0. Wellknown solutions are as follows:

- full-associative TLB (such as for MIPS R4000);
  - – high circuit cost;
- virtual-addressed cache;
  - + faster than a real-addressed cache
  - – difficulties with synonyms
  - – consistency problems with multiprocessors

A further solution is a virtual- and real-addressable cache. It combines the advantages of a virtual-addressed cache (TLB for small pages, for many pages of varying size) with those of a real-addressed cache (synonyms are possible, suitable for multiprocessor systems).

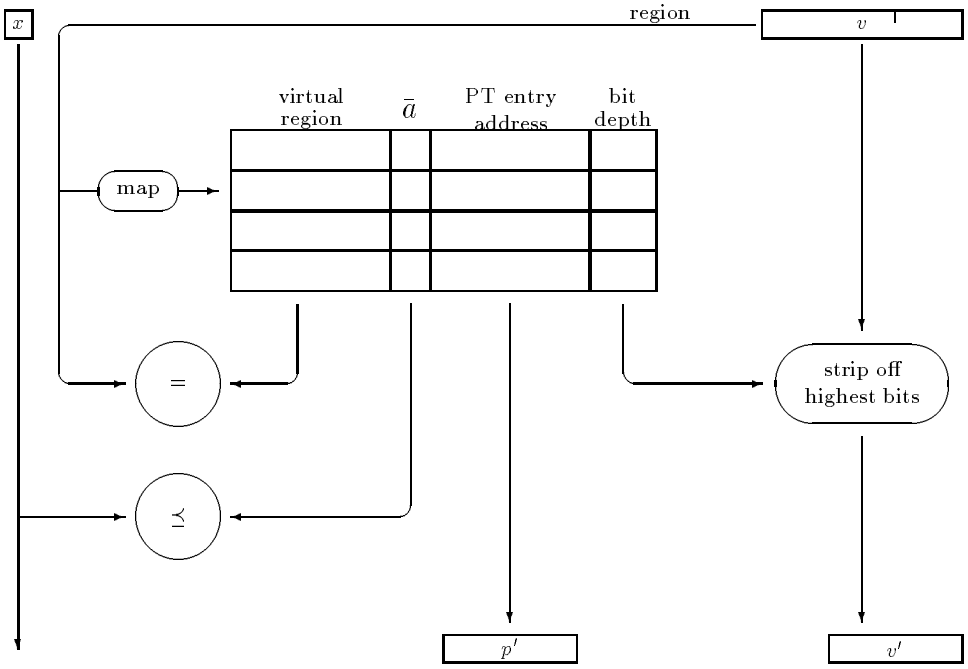


Here, field $\bar{a}$ contains the resulting access attribute which is obtained from the combination of the access attributes of the individual levels during address transformation.

1.4.2 TLB_1 (a)

Just as for TLB_0, several solutions are possible for TLB_1 (and for higher levels).

A specific cache (direct mapped or n-fold associative) can be used for the individual regions:



It is addressed with the region (e.g. $v \div 2^{24}$) and delivers the next possible entrance to the page table tree in the case of a hit. This is the address of the corresponding page table entry and its depth, i.e. the number of leading bits of the virtual address which have already been decoded up to this entry[9] and which therefore have to be removed from the virtual address v upon entering into the address transformation tree transversally. $\bar{a}$ contains the access attributes obtained by combining the page table access attributes during the address transformation up to the region.

In the case of TLB_1 hit and admissible action ($x \preceq a$), the address transformation is performed step by step according to the methods described in 1.3.1, 1.3.2 or 1.3.3. We begin with v' as a virtual address and p' as an address of a page table with only one entry (since TLB_1 does not deliver the page table but supplies the corresponding page table entry).

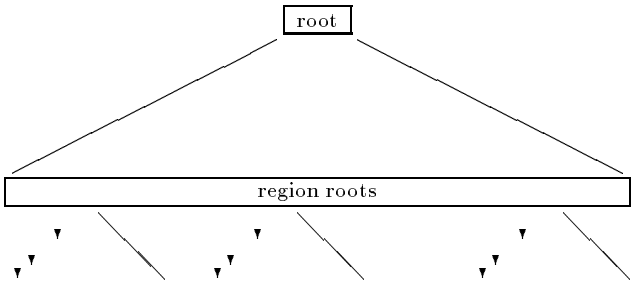
In the case of TLB_1 miss, a complete address transformation is performed as described in 1.3.1, 1.3.2 or 1.3.3. The best possible entrance to the region found in this way is then included in the TLB_1 cache.

1.4.3 TLB_1 (b)

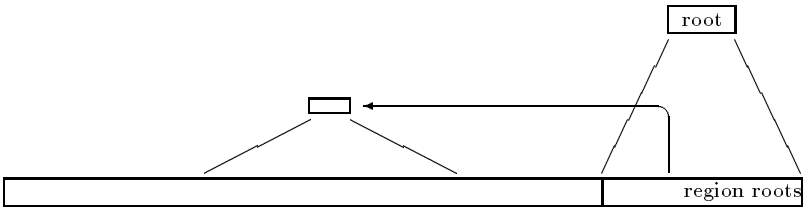
When using a sufficiently large TLB_0 respectively a virtual-addressable data cache, we can do without a specific cache for TLB_1. We can use instead a

[9] It is to be noted that the entry has not necessarily to be located on the "region depth", but it can also be positioned on a higher level due to the guards.

two- (or more) level hierarchy of address transformation trees in accordance with 1.3.1, 1.3.2 or 1.3.3.



If linearized, the address space is as follows, for example:



A specific tree is used for each region. The roots of these trees are accessible via a specific area in the virtual address space ('region roots'). In the case of a TLB_0 miss, it is now attempted to address the corresponding regional tree via its virtual address in the area 'region roots'.

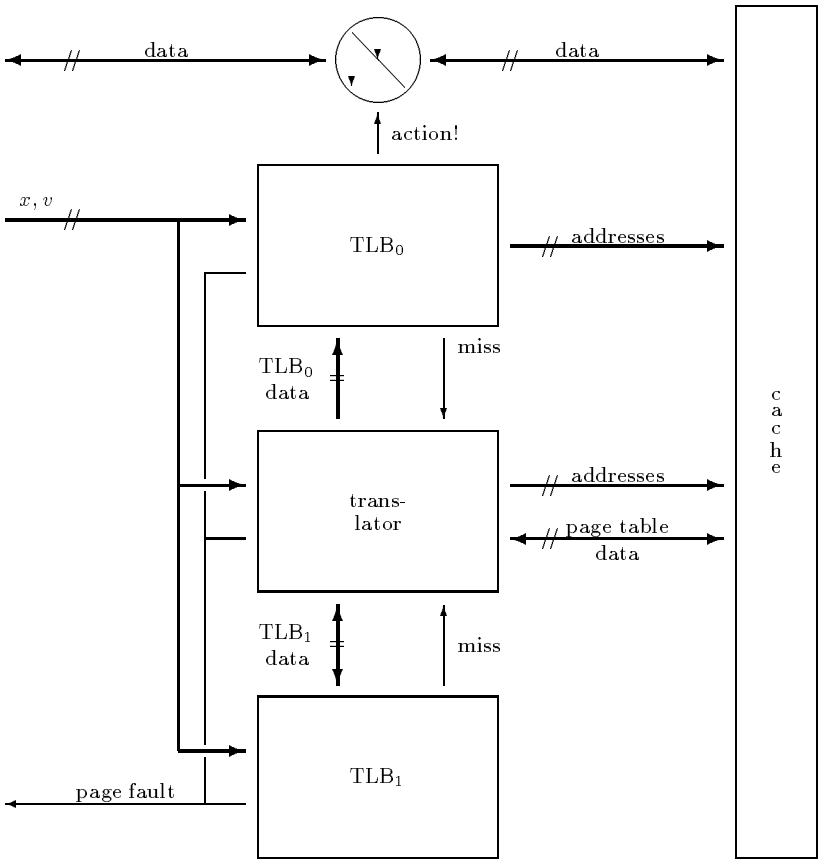
If we succeed by means of TLB_0 hit for the corresponding virtual region root address, we get a TLB_1 hit. Subsequently, we need only parse the (not very deep) regional tree. Otherwise, a complete address transformation of the virtual region root address is executed beginning with 'root', and the regional tree is parsed subsequently.

This method requires less hardware, but, in the extreme case (only one page per region), it may require one additional page table entry per accessible page[10]

1.5 Examples of hardware implementations

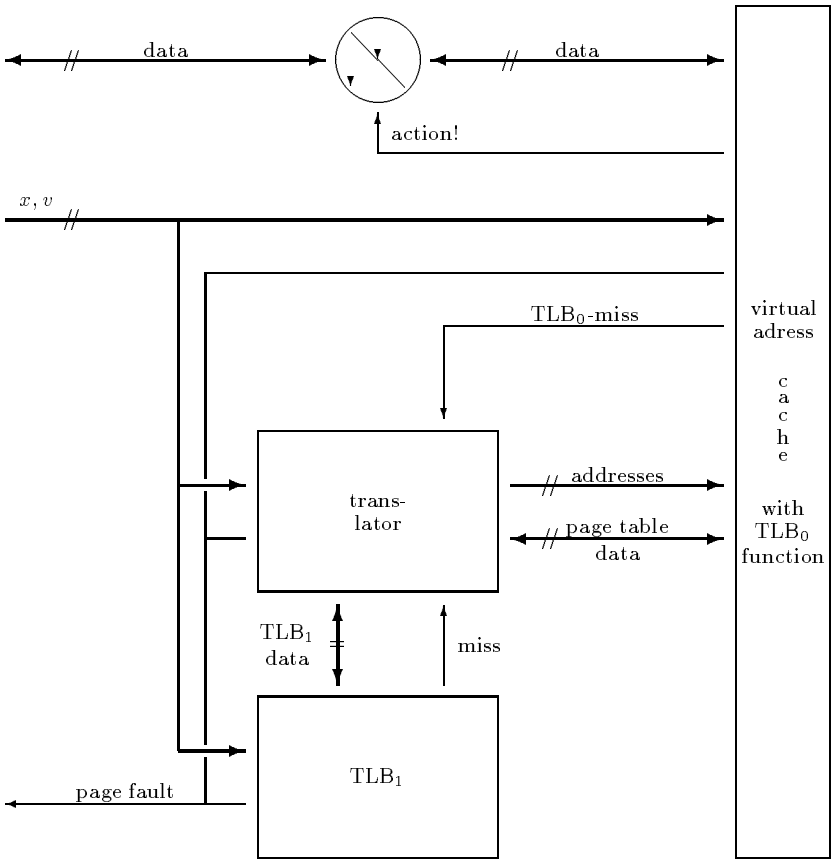
In the following, the *translator* always works according to one of the procedures described in 1.3.1, 1.3.2 or 1.3.3.

[10] If the smallest page is larger than a page table entry, the additional storage requirements may increase because of the fragmentation of the 'region root' area.

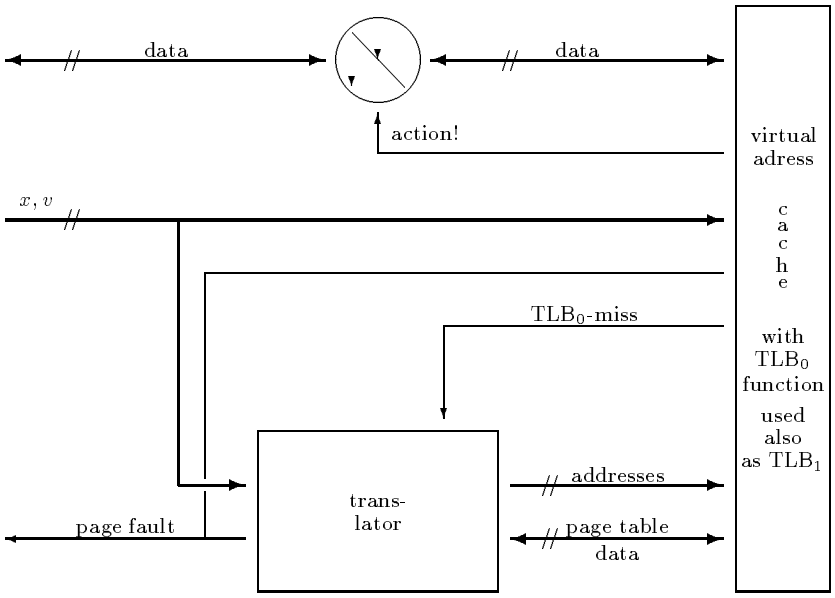


TLB_0 and TLB_1 are available here as independent hardware. Instead of the cache, a normal memory could of course be connected also directly.

If a virtual-addressable cache is used, it can adopt the functionality of the TLB_0:



The block diagram becomes even simpler if the TLB_1 is implemented by the translator with the aid of Cache+TLB_0 as described in 1.4.3:


data
data
action!
virtual adress
x, v
cache
TLB0-miss
with TLB0 function used also as TLB1
translator
addresses
page table data
page fault

Chapter 2

User Level Mapping by Hardware

Note: this chapter corresponds to patent application "Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse (Kennwort: Adressenumsetzung II)", Deutsches Patentamt P 43 19 842.2 (filing date May 27, 1993).

The invention relates to a procedure and equipment for transforming a virtual address into a real address. An application of the invention is the memory management unit (hardware), also called MMU transforming a.o. the address in the virtual memory into an address of the physical or real memory.

The invention described in the following is correlated with patent application P 43 15 567.7 (filing date May 10, 1993).

Fine grained mapping enables an access control on the level of logical storage objects, e.g. program variables. 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 proxies' or simply 'access protocol'.

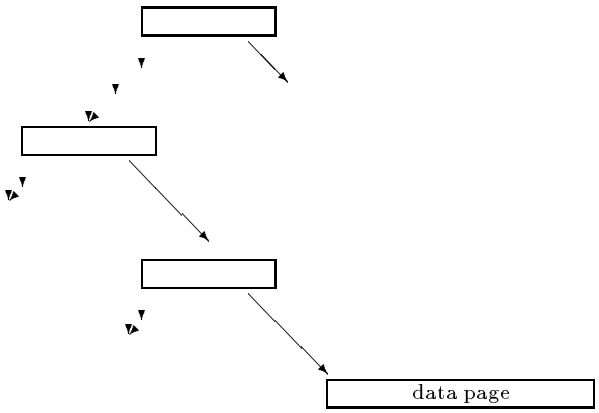
Combining the two methods allows an efficient realization of distributed memories since access to potentially remote objects is feasible by means of local object invocation. Memory accesses are performed directly in the local case and algorithmically in the remote case. Distinction is by hardware.

On the functional level, the operating system (μ-kernel) can realize all this by means of software. Since, however, mapping is likely to be modified very frequently with the mentioned methods, it is desirable to have a facility for modifying mapping directly within the user level program without using the operating system. This option should not affect the security paradigm of the operating system.

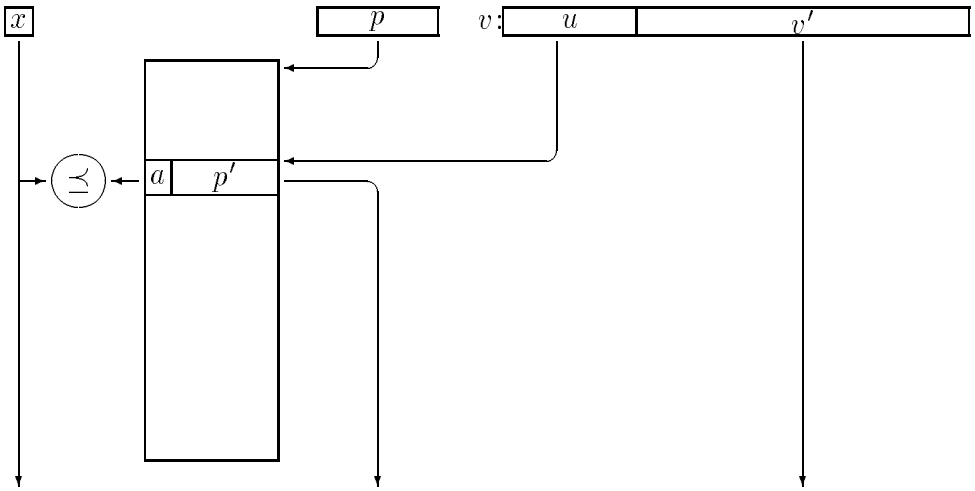
2.1 Conventional and Guarded Page Tables

The extension of MMU presented in 2.2.3 and the instruction set require the use of tree-oriented methods for transforming virtual addresses into real addresses. It is applicable to MMUs with conventional or guarded page tables, but it is inapplicable to inverted page tables.

Guarded page tables are an extension of conventional page tables and allow very small granularity in huge address spaces. Therefore, the combination of user level mapping and guarded page tables seems to be of special interest. On the other hand, user level mapping does nowhere require guards, i.e. it is equally applicable to conventional page tables. For this reason, "simple" conventional page tables are always used for describing the method in the following.



Virtual addresses are transformed step by step into real addresses by means of a page table tree. Let us consider an individual transformation step of a virtual (binary) address v for an action[1] x by means of a page table with the address p. For this purpose, v is split into a high part u (consisting of a specific number of high bits) and a low value part v' (consisting of low bits). Using u, we select an entry of the page table including an access attribute ^1a and a new address p'.



If the access attribute prohibits the action ($x \succ a$), transformation is aborted and page fault is signaled.

[1]For many computers, actions consist of the read/write or execute operation and the mode of operation user/kernel. The access attributes admitting specific actions (in the extreme case all actions or none) are constructed adequately. The set and semantics of actual actions and access attributes and the method of examining action against attribute is irrelevant from the present viewpoint. The crucial point is that a circuit decides only on account of action x and access attribute a whether to enable or to abort an action.

If the action is admissible ($x \preceq a$), x, p' and v' are passed as input parameters to the next level transformation. If the last level is reached, p' points to the beginning of the data page and v' is the offset within the page.

For security reasons, modifications of the page table tree and therefore modifications of mapping can by convention be done only by the operating system (or the μ-kernel). If compared with address transformation, they are therefore time-consuming.

2.2 User Level Mapping

As already mentioned above, *user level mapping* is of special interest to an MMU admitting fine grained address spaces like presented in chapter 1. The method presented here can also be integrated into other MMUs showing fine or coarse grained mapping *if they are based on the transformation of a virtual address into a real address by means of a page table tree.*

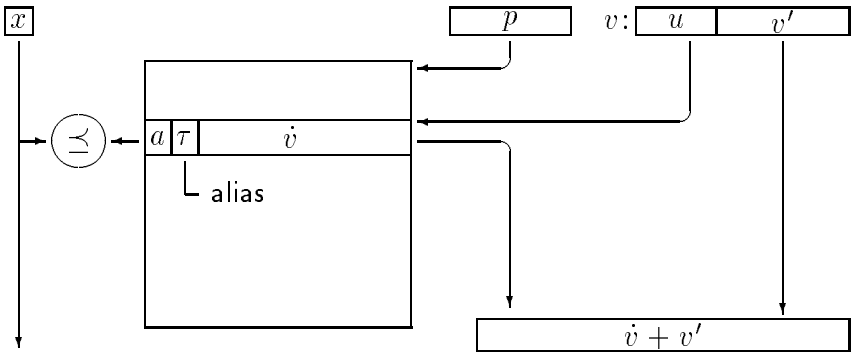
Page Table entries consist conventionally of an access attribute a which defines the applicable actions on the address space region and a pointer π which is the real address p of the next page table level or the data page in the case of normal address transformation.

In addition, each page table entry has a type τ which determines a.o. the interpretation of the pointer π. The conventional entries are of type $\tau =$ translate. For user level mapping, new types and special instructions are introduced for providing a secure modification of mapping.

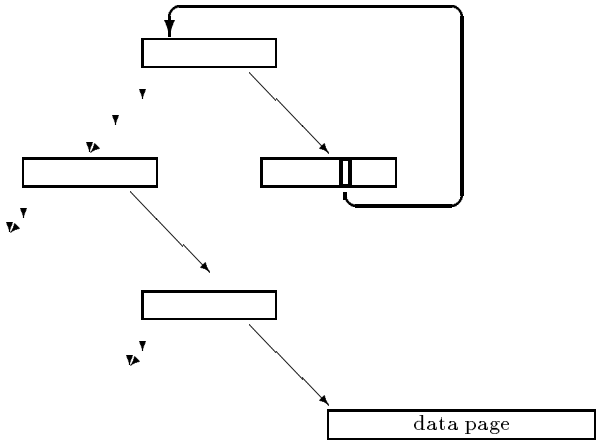
2.2.1 $\tau =$ alias

For the alias type, the pointer π is interpreted as a *virtual* alias address $\dot{v}$. If address transformation meets an alias entry[2] on any level, $\dot{v}$ is added to the remaining address v' not decoded so far.

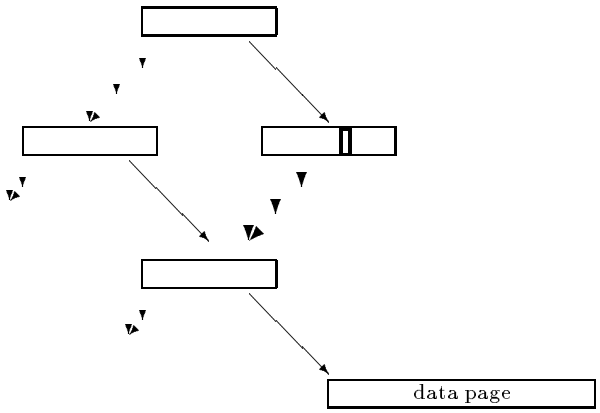
[2] whose access attribute a allows the access action x



Address transformation is restarted with the resulting $\dot{v} + v'$ which is a virtual address of full length. Consequently, the entire virtual region covered by the alias entry is mapped *virtually* to another virtual region:



This is to be distinguished from conventional *real* aliasing where several real pointers meet on a page table or data page:



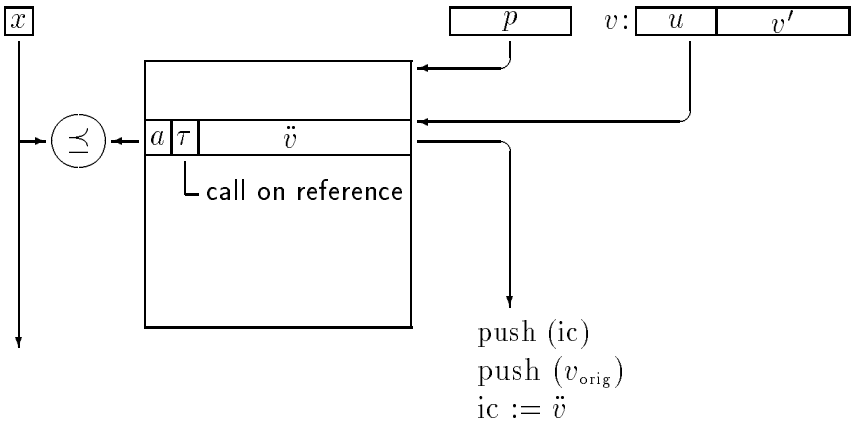
Unlike *real* aliasing, *virtual* aliasing is independent of real memory mapping. In this way, it enables modifications of mapping by means of user level software

- independently of current real memory allocation and paging;
- without being able to violate the protection boundaries of the own or foreign address spaces;
- without being able to weaken the access attributes pre-set by the operating system on pages or larger address space regions.

2.2.2 τ = call on reference

For the type call on reference, the pointer π is interpreted as a virtual address $\ddot{v}$ of a procedure. If address transformation encounters a call on reference entry[3], the accessing operation is aborted and the specified procedure is called instead. It gets the current instruction counter (ic) and the initiating virtual address v_{orig} as input parameters. Just as with a page fault, the initiating instruction can be restarted.

[3] whose access attribute a allows the access action x



Call on reference mapping assigns algorithms to address space regions. These can skip the initiating instruction, they can emulate or handle it in a similar way as with page fault, i.e. to remap the address space region in question by alias and to restart the instruction. Sometimes, skipping and emulation of individual instructions can be accelerated by special processor instructions (see 2.2.4).

Processors with strict load/store architecture allow to improve emulation by additional parameters (besides ic and v_{orig}) passed to the associated procedure:[4]

- *upon write access*
  1. 'write access' code
  2. operand size (byte, word,...)
  3. operand value
- *upon read access*
  1. 'read access' code
  2. operand size (byte, word,...)
  3. number (address) of the destination register

Applying the same idea to processors with more complex instructions leads to slightly different additional parameters:[4]

1. operation (mov, add, inc,...)
2. operand size (byte, word,...)

[4]idea: Martin F. Gergeleit, GMD

3. source register/memory address
4. destination register/memory address.

2.2.3 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.

A page table entry is denoted unambiguously by the virtual address region it covers precisely in the primary[5] address transformation.[6] The addressed entry is specified accordingly by the virtual base address b and size s of the corresponding address space region. The instruction

map $((b, s), (\tau, \pi))$

loads the addressed page table entry with τ and π provided that τ is alias or call on reference[7] *and* that the target entry

- exists[8] *and*
- is accessible from the current mode (user/kernel)[9] *and*
- it is already of the alias or call on reference type[10].

Otherwise, map will lead to page fault. By this, user level software can modify corresponding entries, i.e. switch between alias and call on reference and/or change the alias or the associated procedure address. However, this instruction cannot be used either for creating new entries or for modifying available virtual-real mappings or for weakening access attributes.

Creating and deleting corresponding entries together with the necessary modification of the page table tree should be realized by system calls in the operating system kernel.

Alias and call on reference entries can be realized by the operating system as long-living objects, since swapped-out entries lead to page fault both upon normal access to the address space region and upon map access to the entry.

[5] By *primary* address transformation, we here understand the translation process which transforms the original virtual address until a page fault is diagnosed, until an alias entry, a call on reference entry or an entry referring to a data page is found.

[6] The reverse does not hold since an entry can be responsible for several virtual address space regions on account of real sharing.

[7] Consequently, virtual-real mappings cannot be defined.

[8] Consequently, entries cannot be created secretly.

[9] Consequently, the operating system can protect itself.

[10] Overlaying a translate entry in the user part of the address space by alias or call on reference is not security-relevant, but it would lead to dead subtrees without the operating system noticing it.

2.2.4 Supplementary Instructions

Further processor instructions are not necessary, but they might be of interest for specific processors and applications:

getmap $((b, s), (\tau, a, \pi))$

reads a page table entry provided that it exists, is accessible from the current mode and is of the alias or call of reference type.

Skipping and emulation by means of a procedure associated with the address space region might get more efficient by using instructions which return the length of another instruction or which can execute it with a modified memory address:

getlength (dest, ptr)

returns the length of the instruction which is located at the virtual address 'ptr' in 'dest'.

execute (ptr, v)

executes the instruction located at the virtual address 'ptr' using however the virtual address v instead of the memory address actually used in the instruction.[11]

Both instructions are not useful for processor architectures showing only a few and simple instruction formats and addressing modes.

2.2.5 τ =call/alias

In principle, the entry types alias and call on reference are sufficient (in addition to translate). Combinations might also be of interest:

τ =call on write / alias on read
τ =call on read / alias on write

However, the page table entries must be enlarged in this case to include $\dot{v}$ and $\ddot{v}$ together, or $\ddot{v}$ has to be the same for all these entries. Then the map instruction of course works on these two types too.

Personal assessment: Both combination types are superfluous. They can be emulated efficiently enough by means of the original types.

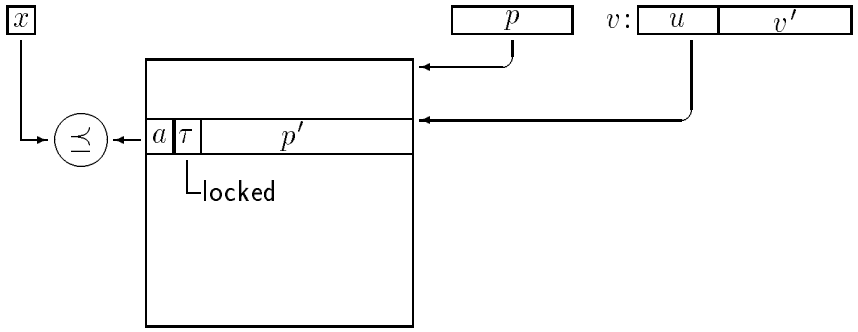
[11] For multi-address machines, more complex forms of the instruction might be necessary.

2.2.6 τ =locked

For associating specific hardware actions with address space regions, e.g. activating complex bus protocols for access to a remote memory, an alias entry can be used. It maps the access to a correspondingly sensitive hardware address (memory mapped I/O).

For multiprocessor machines with shared memory, an address space region should sometimes be locked so that accesses to it by other processors are delayed automatically [12] until release. Locking can be done by associating an empty routine to the address space region, i.e. by using a call on reference page table entry pointing directly to a ret instruction. Unlocking is done by modifying the mapping to alias.

For some architectures, a specific new type locked may be useful. In some cases, it can be implemented somewhat more efficiently since it allows lock/unlock without reinterpretations in cache and TLB:



If address transformation encounters such an entry, it is restarted completely. Only if another processor sets the entry to translate again or modifies the page table tree in such a way that the entry is no longer involved, the delay is terminated. Locked entries differ only in the type from translate entries. Changes between the two therefore only require a consistent modification of the types of the corresponding cache and the TLB entries[13], but no modification of the virtual-real mapping.

Two further instructions are used for changing between locked and translate:

[12] Of course, at least one processor should be able to access. This is done by virtual or real aliasing.

[13] For larger objects, a TLB flush is probably more efficient.

lock (b, s) unlock (b, s)

lock sets the addressed page table entry to locked and unlock sets it again to translate provided that the target entry

- exists *and*
- is accessible from the current mode (user/kernel) *and*
- is already of the type translate or locked.

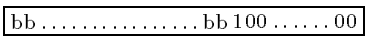
In all other cases, page fault will be triggered.

2.3 Possible Codings

Pairs (b, s) of n-bit-wide baseaddress b and size $s = 2^i$ $(s \geq 2)$ can be coded as n-bit-value

$$b + \frac{s}{2},$$

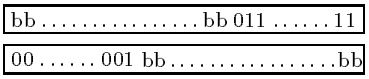
if the baseaddress is always s-aligned, i.e. $b \bmod s = 0$. Then the bitrepresentation looks as follows:



The method can be used for coding bitstrings of variable length up to maximum $n - 1$ as well. E.g. guards of Guarded Page Tables can be coded by this. A bitstring b of length $\|b\|$ is represented by the n-bit-number

$$2^{n-\|b\|}b + 2^{n-\|b\|-1}.$$

In both cases, the decoding hardware takes the lowest 1-bit as delimiter. Of course, the roles of 1-bit and 0-bits can be exchanged or the higher bits can be used for the length coding:



References

[1] J. Liedtke. Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse. *Patent application P 43 15 567.7. Deutsches Patentamt, München.*

[2] J. Liedtke. Verfahren und Vorrichtung zum Umsetzen einer virtuellen Adresse in eine reale Adresse (Kennwort: Adressenumsetzung II). *Patent application P 43 19 842.2. Deutsches Patentamt, München.*

[3] J. Liedtke. *Some Theorems About Guarded Page Tables.* Arbeitspapiere der GMD No. 792. St. Augustin, 1993.