Dealing With TLB Tags
or
I Want to Build a System, What Can L4 Do for Me?

Gernot Heiser

School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
gernot@unsw.edu.au

October 11, 2001

Abstract

This paper discusses TLB tags found on various architectures, their use in single- and multi-address-space operating systems, and the implications on the L4 API.

1 Introduction

Most modern computer architectures tag entries in the translation lookaside buffer (TLB) with one or more fields which identify the addressing or protection context to which an entry belongs. This makes it possible to minimise TLB flushes during context switches.

This paper presents the tagging scheme used by various architectures. It first presents the schemes presently in use. It then examines each scheme from the angle of how it might be used in the context of a single-address-space operating system (SASOS), as well as a more traditional multi-address-space operating system (MASOS). Finally it attempts to draw conclusions on which L4 mechanisms might be appropriate to support the use of the various schemes for the implementation of both kinds of operating systems. APIs for dealing with tags are suggested.

2 TLB Tags

The following tagging schemes are in used in contemporary architectures:

Address-space identifier (ASID): Each TLB entry (TLBE) is tagged with an ID representing the process it belongs to. On address translation the contents of an *ASID register* is combined with the page number to form the key for which the TLB is searched. TLB entries are thus only considered valid for translation if they match the value of an ASID register, which is part of the executing process's context.[1] The total number of possible ASID values is typically of the order 2^8–2^{12}. ASIDs are used on the MIPS [Hei93], Alpha [Dig92] and UltraSPARC [Sun97] architectures, as well as some PowerPC processors [IBM01].

The Pentium's segment registers can, in certain circumstances, be used like ASIDs (Liedtke's *small-address-space trick* [Lie95]), although they are conceptually more akin to region IDs.

Region identifier (RID): This represents a generalisation of the ASID scheme. Each TLB entry is tagged by a RID, and several RIDs may be active at any time. This makes it easier to share data (and makes a global bit unnecessary). RIDs are determined by the leading three bits of the virtual address, the *virtual region number* (VRN). The VRN identifies one of 8 *region registers* (RRs), which contain the RID corresponding to the virtual address. RIDs are 24 bits wide, but the full size may not be supported by the hardware. However, the architecture specifies that at least 2^{18} RIDs are supported on IA-64. RIDs are used on HP PA-RISC [Lee89] (called *space IDs* there) and IA-64 [Int00].

Protection key (PK): PKs represent an alternative to ASIDs. They are not used in the associative lookup of a TLB entry, but are used once a matching entry is found. The PK of the entry is used in a second associative lookup, this time of a set of *protection key registers* (PKRs). The matching PKR (if any) contains a second set of access rights, in addition to those in the TLBE. Both sets must allow the attempted access. By treating the PKRs as part of the process context, this scheme supports sharing

[1] Most architectures will also support a *global bit* which forces a TLB entry to match irrespective of its ASID value.

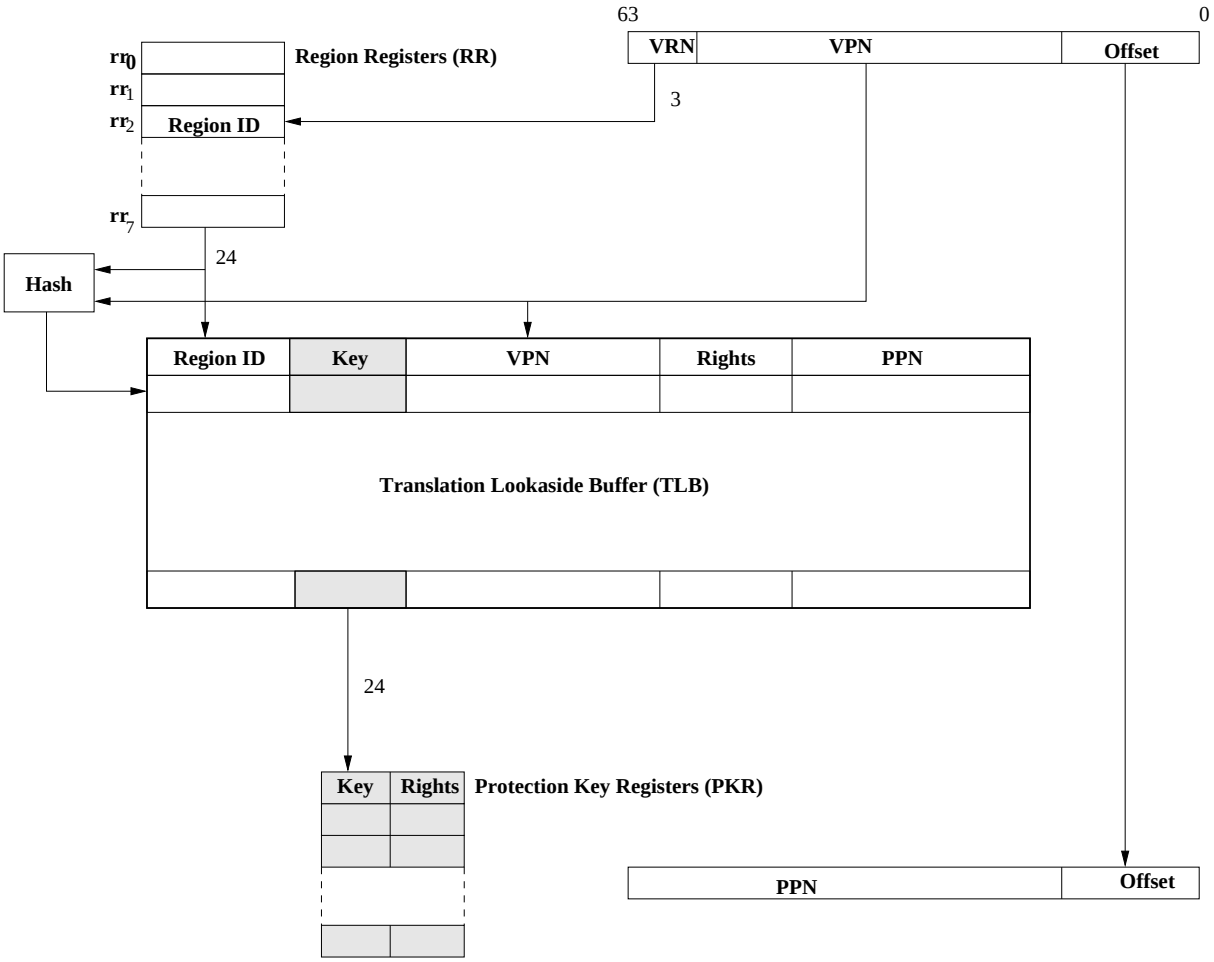

Figure 1: Address translation in IA-64, exhibiting the most general TLB tagging

a TLBE by processes sharing a page (with potentially different permissions, **but** the same address). The total number of possible PKs is large, 2^{15} on HP PA-RISC and at least 2^{18} on IA-64 (PKs are 24 bits wide on that architecture). This scheme is used in PA-RISC (where PKs are called *access IDs* or *protection IDs*) and IA-64.

Domain identifier (DID): This scheme is very similar to PKs. The main difference is that the number of possible tags is very small and the DID is used to index a *domain register* rather than using an associate lookup. Furthermore, ARM domains apply to whole 1MB *sections* of the address space, rather than individual pages. Uses are otherwise the same as for PKs. The total number of DIDs available is 16. DIDs are used in the ARM architecture [Jag95].

Figure 1 shows the address translation in IA-64, which exhibits the most general tagging scheme.

For completeness' sake, the StrongARM **process ID virtual address mapping** scheme [Int98] should be mentioned, even though it is not strictly a TLB tagging scheme. An address in the lower 32MB of the virtual address space is automatically re-mapped to another aligned 32MB region in the lower half of the virtual address space, identified by the contents of the *process ID register* (PIDR). This avoids TLB flushes when switching between small address spaces in a similar fashion to Liedtke's trick.

3 Other Architectures

How about other architectures? The only major architecture not covered so far is the PowerPC, which has a "proper" segmented memory architecture [MSSW94]. The user view of the 64-bit PowerPC's address space is that of a 64-bit *effective address* (EA), consisting of a 36-bit *effective segment ID* (ESID), a 16-bit page num-

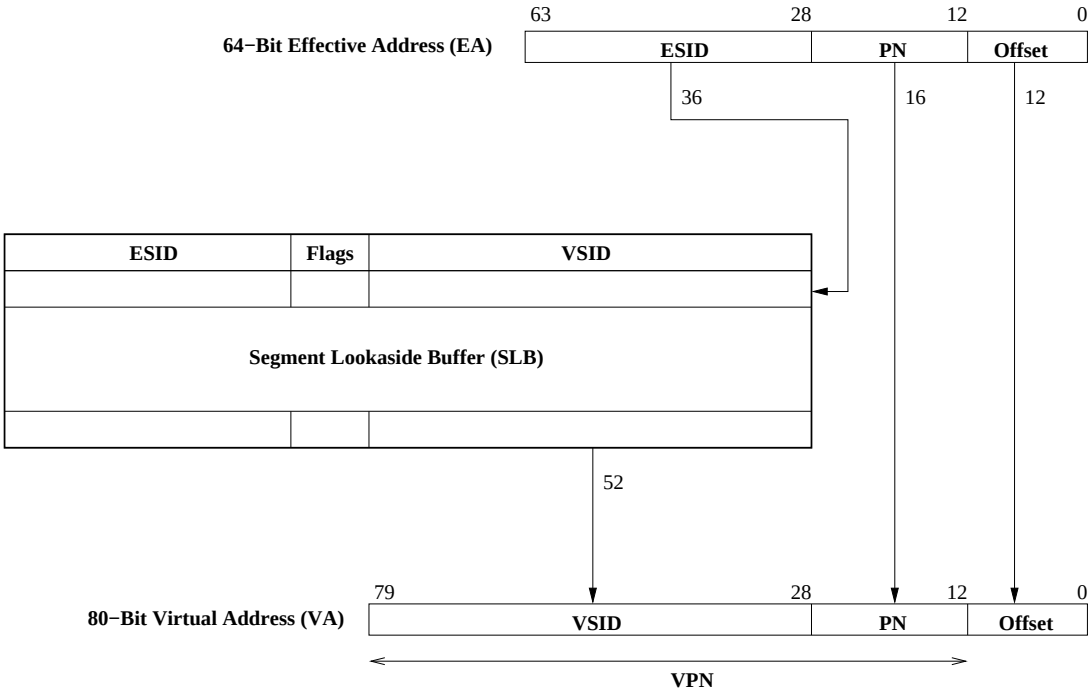

Figure 2: Translation from segmented to flat virtual addresses on the PowerPC architecture

ber (PN) and a 12-byte offset.[2] This is translated into a flat 80-bit *virtual address* (VA) consisting of a 68-bit *virtual page number* (VPN) and a 12-bit offset. The VPN is created by concatenating a 52-bit *virtual segment ID* (VSID) with the PN.

The translation from EA to VA happens by using the ESID as an index into a hashed *segment table*. Segment table entries are cached in a *segment lookaside buffer* (SLB), which is loaded from the segment table by a hardware walker. Translation from the segmented EA to the flat VA is shown in simplified form in Figure 2. The VPN is translated into a frame number in the usual fashion via a TLB (which may be hardware or software loaded).

According to the generic PowerPC documentation neither the SLB nor the TLB contains a context tag (and the SLB contains no write-protection bit). For the TLB this is no issue, as the 80-bit VA is context independent. This is not the case for the EA, though, and in a system where a process's address-space layout is determined at link time, a context switch would imply a switch of the segment table, and hence force flushing of the SLB. This is probably not a big deal, as the SLB is small and hardware-loaded. Still, context switch overhead would be reduced if the SLB was tagged, or if it was at least possible to pre-load it on a context switch (which the architecture doesn't seem to support either).

PowerPC ESIDs/VSIDs look superficially like IA-64 VRNs/RIDs. However, the vastly different number (256M ESIDs vs. 8 VRNs) implies significant differences in handling them. The set of IA-64 RIDs is contained in the 8 RRs, which are set once for a process and and can easily be reloaded on a context switch. Contrast this to the PowerPC, where the set of VSIDs make up the segment table and are cached in the SLB. The implications of this difference should become clearer later on.

However, I consider it unlikely that IBM does not support context switches without SLB flushes on their high-end servers, so I assume that the information on the architecture I have presently at hand is incomplete. Hence I suggest that further discussion of this architecture be deferred until we have specific documentation of memory management on 64-bit PowerPCs.

4 What can I do with these tags?

In this section I'll discuss the ways the tags would be used in a MASOS or SASOS scenario. Assuming that the MASOS or SASOS is to be implemented on top of L4, I'll discuss whether the tag assignment can be done by the kernel or not.

4.1 Address-space IDs

Irrespective of whether I'm building a MASOS or a SASOS, I want to associate each process with a unique

[2]The PowerPC architecture does not seem to support super pages.

ASID, and want to tag all of the process's TLBEs with that ASID value. An exception would be hardware which provides other tagging schemes as well, but I am not aware of such an architecture. ASID assignment is a fairly mechanical process, and I'd be happy to leave it to the kernel.

A complication arises from the fact that the number if ASIDs is limited, and can easily be exceeded by the number of processes in the system. This requires preemption of a process's ASID, and a (costly) invalidation of all TLBEs tagged with it. The preemption strategy introduces a significant amount of policy. As long as it can be assumed that the *process working set* is small compared to the number of ASIDs available, and ASID preemption is rare, the actual preemption policy is irrelevant and it doesn't matter whether it is done in the kernel or by the OS server. However, it is probably possible to construct for each preemption strategy a scenario where it performs badly. This makes it dubious whether ASID handling should be done by the kernel.

Things become even more complicated if the Pentium's segment registers are used to simulate ASIDs. This introduces additional policy — which address spaces are considered "small". "Small" address spaces can be switched quickly, but are limited in size, as all of them must share the Pentium's 32-bit address space with the largest possible "large" address space, plus the kernel-reserved address-space region. It seems unreasonable to leave the allocation of small-address-space IDs (SmASIDs) to the kernel.

4.2 Region IDs

The only architectures which presently provide RIDs are IA-64 and its predecessor PA-RISC, both of which also provide PKs. This combination implies that optimal TLB management differs markedly between a MASOS and a SASOS.

4.2.1 In a MASOS

For a MASOS I want to associate each process with a unique RID. RIDs are plentiful enough so that preemption is not considered an issue, so RID assignment could safely be left to the kernel, if a traditional approach to address-space management is used.

However, this makes poor use of RIDs. The architecture allows us to share TLB entries, e.g. for shared text segments of executables, by assigning a separate RID to the executable [AH01]. In general this approach requires at least two different RIDs associated with each process, in two or more RRs, one for process-private data and one for shared data and code. The kernel may want to reserve one RR for global data. The non-kernel RRs are initialised at process creation time and remain unchanged for the lifetime of the process. This scheme requires some control over RIDs by the OS server.

4.2.2 In a SASOS

When building a SASOS in IA-64 there is no need to use RIDs for tagging TLB entries. Of course, they cannot simply be turned off. Instead they would effectively be disabled by assigning a fixed unique RID value to each RR entry (e.g., just the RR number), independent of any process. A MASOS-like per-process assignment of RIDs would defeat the potential of the architecture to share TLBEs for shared pages in a SASOS.

4.3 Protection keys

Protection keys (while also supported on PA-RISC) are only really an issue on IA-64, where they would be used as an alternative to, or in combination with, region IDs. Their use differs significantly between MASOS and SASOS.

4.3.1 In a SASOS

A SASOS would express all access rights in the PKRs, and set the TLBEs permissions to maximum access. The PKR entries are used to define access rights. This is straightforward if the OS server has control over the PKs tagged to particular mappings. Alternatively the kernel could detect shared pages, assign PKs on the fly, and migrate the permission bits from the TLBEs to the PKRs. This requires probing or flushing the TLB when a page is first mapped into a second address space.

4.3.2 In a MASOS

In a traditional approach to MASOS, protection keys would be disabled (or rather, all TLBEs would be tagged with the same PK, and the corresponding PKR would be pre-loaded with a never-changing entry that allows all access).

However, if one wants to utilise the architecture's potential for sharing TLBEs on shared pages, one would use them in a SASOS-like fashion (discussed above) [AH01]. The above comments apply here as well.

4.4 Domain IDs

DIDs are very similar to PKs and can be used in pretty much the same way. The main difference is that the number of DIDs is very small, and the preemption policy becomes quite important [WH00]. If DID management is done inside the kernel, it must be ensured that the DID consumption rate is minimised. For example, the kernel would need to ensure that all pages shared by two particular processes are tagged with the same DID.

More so than in the ASID case, the rather restricted set of DIDs makes it likely that any particular preemption policy will break under some reasonably realistic scenario.

With respect to managing the PIDR address-space relocation, the comments made on SmASIDs apply correspondingly.

5 What do I want from L4?

In short, I want the kernel to get out of my way as much as possible. Efficient use of the hardware should not be obstructed by the kernel. This is another way of saying that the kernel should be policy-free.

Let's therefore revisit the various tagging schemes under the angle of whether they should be managed by the kernel or the OS server implemented on top of it.

5.1 Region IDs

As explained above, there must at least be a way to turn off RID assignment, in order to support a SASOS. This could presumably be done by a global state flag.[3]

The scheme for sharing TLB entries for shared text segments [AH01] requires that at least two different RID tags are utilised for a each process, one for private data and one for shared data. Changing the region on the fly when sharing is detected is not feasible, as this would require changing the virtual address. Hence, *some* user-level control over RID assignment is necessary.

Any limited control is likely to be a foul compromise. I think that we should bite the bullet and give the OS server *full* control over RID assignment. This can be achieved by making RRs accessible to user code.

I postulate that L4 *must* give OS servers control over RRs.

The kernel might reserve one or two RRs for its own use, the others would be set at address-space creation time. I see no need to change them later, unless RIDs need to be pre-empted. Given the large number this seems unlikely; IA-64's 2^{18} possible RID values will support about 100-200k concurrent processes.

Note that this approach is not likely to be suitable for ESIDs on 64-bit PowerPCs.

5.2 Address-space IDs

As mentioned earlier, RIDs are a generalisation of ASIDs. ASIDs do not support sharing of TLB entries (other than for truly global data accessible by everybody). Therefore, control over their *assignment* isn't necessarily required.

However, the issue of ASID *preemption* has been brought up before, and the fact that this introduces a rather uncomfortable amount of policy into the kernel. It could be argued that ASIDs have been kernel-managed on all ASID-tagged architecture where L4 has been implemented to date (MIPS and Alpha), and no-one has complained so far. However, it must then also be admitted that none of these L4 implementations has to date been used in a production setting where ASID preemption would have been an issue. We don't really have any relevant experience with ASID recycling.

Add to this the fact that the small-address-space trick, which simulates ASIDs on the Pentium, requires that this simulated ASID be user-managed. If user-level management of SmASIDs is accepted to be necessary, we might as well do the same for real ASIDs (and for the PIDR).

I postulate that L4 should leave ASID management to the user level, and the mechanism to do this should be analogous to the mechanism for managing small address spaces on the Pentium, and for the StrongARM PIDR.

The present (V2) mechanism for assigning SmASIDs on the Pentium is via the `thread_schedule()` syscall. This serves to declare an address space as "small", assign a SmASIDs to it, as well as for invalidating the SmASID (I think — this isn't really documented). The same mechanism should be applicable for ASIDs and for the PIDR. The question remains what the kernel's default ASID assignment would be, if no ASID is supplied explicitly.

Note that the way present ix86 kernels deal with SmASIDs introduces a significant amount of policy into the kernel: the size of a small address space, the number of small address spaces (and, by implication, the maximum size of a "large" address space).

5.3 Domain IDs

The amount of policy introduced into the kernel by managing the assignment and, in particular, preemption of DIDs [WH00] is outright intolerable. Whatever policy is implemented in the kernel is likely to go horribly wrong in some cases.

I postulate that DID management *must* be done at L4 user level.

The way to provide user-level control over DID assignment would be to specify a DID to each mapping IPC. There are some problems with this approach, for

[3] Global to the particular OS server, at least; different OS personalities should be able to implement different address-space management policies.

example the fact that duplicate mappings that only differ by the RID must not be inserted into the TLB. Also, the fact that regions apply to 1MB sections makes this approach messy and error-prone.

A better way to deal with this seems to be Szmajda's proposed *linking* of fpages [Szm01], which shares address-space regions by sharing page table subtrees. A mechanism must be provided to specify the DID when the fpage is linked (i.e., at linking-IPC time). The kernel would ignore the specification if it is not for a correct (1MB) size fpage.

Revoking the DID assignment would be done by unmapping all pages tagged with a particular DID. The OS server can be held responsible for remembering which pages were tagged with a particular PK.

5.4 Protection keys

As discussed in Section 4.3.1, the kernel could assign PKs on demand (i.e., when it discovers sharing). This is a reasonable thing to do, as it doesn't introduce policy as such. There are, however, a number of issues to consider:

- The limited number of PKRs (16 on the first Itanium generation) makes PKR entry caching essential. Putting the cache into the kernel is clearly fastest and is simple as far as the API is concerned. As the PKRs are totally software managed this raises the issue of the replacement policy. However, this is not a serious policy issue as long as reloads are fast.

- The kernel has no way of telling which pages shared by a particular pair of processes are part of the same logical object. If a page shared between two processes is mapped to a third process, it must be given a PK separate from any used for pages shared between the two original processes. The kernel needs to change PKs frequently (requiring a TLB flush each time) or using a unique PK for each shared page. This consumes PKs at a much higher rate than if PKs were controlled by the OS server, which knows which pages belong together (and always share their permissions). Whether or not this is a problem depends on whether the number of shared pages is likely to exceed the number of protection keys. The specified minimum of 2^{18} is sufficient for 2GB of shared memory when using an 8kB page size.

- The fact that PKs are very much like DIDs (except that they are far more plentiful) speaks in favour of treating them the same. Ideally this is done in a way that still allows PKR management to be done by the kernel.

I postulate that PK management should be done at user level, using the same mechanism as for DIDs. The PKRs should remain under kernel control.

The preferred mechanism is again Szmajda's fpage-linking mechanism, with the addition of specifying a PK when the link is set up. Contrary to domains there is no restriction on the size of the fpage to be linked.

Revocation is as with DIDs.

6 Suggested APIs for TLB Tags

According to the discussion in the preceding section, three kinds of tag manipulation are required: initialisation of RIDs, specification and invalidation of ASIDs and their equivalents, and specification and revocation of PKs and equivalent. Here we examine ways to achieve this in the V4 API [L4K01].

6.1 Specifying RIDs

RIDs only need to be explicitly specified at address-space creation time, by specifying the contents of the RRs. The obvious way to do this is via the `SpaceControl()` system call. While the architecture specifies 8 RRs, the kernel may reserve one of them for its own use. Hence the *kernel interface page* (KIP) should specify the number of available RRs (which also limits the amount of virtual address space usable by user code).

The RR values could be specified either by explicitly passing them in general purpose registers, or by adding appropriate *virtual registers*. The former approach has the drawback that the `SpaceControl()` syscall would have architecture-specific arguments. The latter approach has the drawback that it introduces virtual registers which aren't thread-specific but address-space specific. I don't have a strong opinion on this.

6.2 Specifying ASIDs, SmASIDs and PIDs

In this section the term "ASID" is meant to include "SmASID" as well as the contents of the StrongARM PIDR.

6.2.1 Setting up ASIDs

Previous versions of L4 for IA-32 used `thread_schedule()` to control the SmASID, and earlier drafts of the V4 API used the corresponding `schedule()` syscall. The draft dated 4 October 2001 changed this to use `SpaceControl()`. The logic behind this change is, presumably, that the SmASID is an address-space attribute rather than a thread attribute, and the same applies for ASIDs. Furthermore, ASIDs

are a limited resource, which must be shared between all servers running on top of L4.

I propose that the semantics of the `control` argument to `SpaceControl()`, which is used for ASID assignment, be generalised as follows:

- The ASID field in `control` is widened to at least 32 bits.
- The KIP specifies the range of ASIDs supported by the hardware. Note that this value may differ between different implementation of the same basic architecture. The kernel may reserve a small number of ASID values for its own use (typically ASID zero).
- There is one ASID value reserved for "large" address spaces. This value is used for non-relocated address spaces on ix86 and StrongARM, and is also to be used for architectures with untagged TLBs (including IA-64).
- Assignments of kernel-reserved ASID values to user address spaces is illegal.
- Assignments of an ASID value larger than the largest supported one is legal, but it implies that no thread of such an address space is runnable. This supports non-destructive revocation of ASIDs from address spaces.

The question remains what the meaning of the "large" ASID is on architectures with an ASID-tagged TLB. It could be treated like an out-of-range value.

6.2.2 Revoking ASIDs

The need to revoke ASIDs requires further consideration. Assignment of an ASID to an address space should implicitly preempt that ID from any other address space it may have been assigned to previously. This makes an explicit mechanism for ASID revocation unnecessary. The proposed semantics of `SpaceControl()` already supports non-destructive calls on an existing address space, so the specification needs to be clarified so that it allows to make use of it for ASID assignment.

6.3 Specifying PKs and DIDs

In this section the term "PK" is meant to include "DID".

6.3.1 Setting up PKs

In order to keep things simple, we will support TLBE sharing only when also sharing page tables, i.e. when regions of two or more address spaces are *linked* [Szm01]. This will greatly simplify implementation of PKs (and implicitly ensures that the same-address requirement is met).

PKs are specified via the linking `Ipc()` system call. I don't have a strong opinion as to whether this should happen via an additional argument or via a virtual register. It cannot be done via a *LinkItem*, as there is insufficient space left. As well, it makes sense to use the same PK for all linkings specified in a single `Ipc()` call.

The semantics of linking linked pages to further address spaces must be clarified. As all pages linked together must share a PK, there are three possibilities:

1. When linking pages which are already linked, any PK specification is ignored.
2. When linking pages which are already linked, any PK specification overrides any previous PK specification for those pages.
3. When linking pages which are already linked, any PK specification must be the same as any previous PK specified for those pages.

I don't have a strong opinion on which is to be preferred, and would suggest to determine this based on implementation considerations. However, the second variant might be insecure.

Remember that ARM DIDs apply to whole 1MB sections only. Linking of other fpages might still be supported, but a PK specification would be ignored unless it is for an fpage of at least 1MB in size.

6.3.2 Revoking PKs

PKs can be revoked by flushing all links to a page. The semantics are that the kernel removes the PK from a page if its link count falls below two.

In addition, flushing by PK can be considered. This could work by the `Unmap()` syscall supporting the use of a PK list as an alternative to an fpage list. The advantage of this scheme would be increased efficiency and ease of revocation. The drawback is that it is not necessary (and thus should not be supported according to the microkernel *credo*). So it's probably not a good idea.

7 Conclusions

An examination of appropriate use of TLB tags in either a MASOS or a SASOS scenario leads to the conclusion that tag management should be under the control of the OS personality implemented on top of L4. The microkernel needs to provide mechanisms that enable OS servers to perform this task.

References

[AH01] Alan Au and Gernot Heiser. TLB sharing in IA-64 Linux. In *Australian*

Linux Conference, January 2001. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[Dig92] Digital Equipment Corp., Maynard, MA, USA. *Alpha Architecture Handbook*, 1992.

[Hei93] Joseph Heinrich. *MIPS R4000 User's Manual*. Prentice Hall, 1993.

[IBM01] IBM. *PowerPC 405GP Embedded Processor User's Manual*, 9th edition, March 2001.

[Int98] Intel Corp. *SA-1100 Microprocessor Technical Reference Manual*, September 1998. Order no: 278088-001.

[Int00] Intel Corp. *IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture*, January 2000. URL http://developer.intel.com/design/ia-64/index.htm, order no 245318-001.

[Jag95] Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, July 1995.

[L4K01] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual*. University of Karlsruhe, version 4-x.2 edition, October 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[Lee89] Ruby B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report 933, GMD SET-RS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany, November 1995.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[Sun97] Sun Microsystems, Palo Alto, CA, USA. *UltraSPARC User's Manual*, July 1997.

[Szm01] Cristan Szmajda. Calypso: A portable translation layer. Subm. to L4 Workshop, August 2001.

[WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *Proceedings of the 5th Australasian Computer Architecture Conference (ACAC)*, pages 97–104, Canberra, Australia, January 2000. IEEE CS Press.