The Design of the XI Shadow Mechanism

Robert S. Phillips

1 Overview

"XI" refers to an implementation within the Xen hypervisor of shadow page tables for fully virtualized guest domains. This document describes the design of XI.

In a fully virtualized environment the guest does not control the hardware page tables. It maintains guest page tables and the hypervisor maintains the hardware page table. The hypervisor reflects changes made by the guest in its page tables into the hardware page tables.

Page tables contain physical addresses. For the guest, it uses pseudo-physical addresses (PFNs) in its page tables. The hypervisor uses machine physical addresses (MFNs) in the shadow page tables. Much of the hypervisor's work in shadowing the guest page tables is to validate PFNs and translate them to MFNs.

The hypervisor detects when its shadow tables are out-of-sync with the guest tables and it resyncs them. This happens at two times: when handling page faults and when processing TLB flushes.

The guest can increase access to pages by marking 'not present' guest page table entries as 'present', or marking 'read only' entries as 'read write'. The hypervisor will only detect its shadow entries are out of sync when the guest faults while attempting to access or write those pages. At that time the hypervisor resyncs the shadow page table entries and the guest retries the access.

The guest can reduce access to pages by marking them 'not present' or 'read only' or by changing the pages to which page table entries point. Before such changes become effective, the guest must flush the affected TLBs. This causes a VMEXIT which gives the hypervisor an opportunity to resync any modified page table entries.

But how does the hypervisor know which page tables have been modified? It is not practical to scan all guest page tables. The solution is to keep all guest page tables read-only normally. Then, when the guest tries to change a guest page table, a page fault occurs. The hypervisor can add the guest page table to an "out of sync" (OOS) list and allow write access. At TLB flush time, the hypervisor need only review the pages on the OOS list for changes. Once all changes in the guest page tables have been reflected in the shadow page tables, the OOS list is emptied and all guest page tables are made read-only again.

2 Rationale

Virtual Iron decided to re-write the shadow code for the following reasons:

1. In SMP machines, the shadow code must support each virtual CPU having its own memory model; for example, one CPU might be running in 64-bit mode and another in 32-bit non-PAE mode.
2. XI provides recovery from out-of-memory problems rather than crashing the machine. The shadow pages are charged against its domain. They are not take from the hypervisor's small pool of memory.
3. XI minimizes "resyncing"; that is, bringing shadow pages back into sync with their corresponding guest page tables.
4. It avoids discarding shadow pages during context switches. When a process resumes execution, its shadow pages probably still exist.
5. It has short and simple critical paths, such as the page fault path.
6. It maintains data structures that allow operations to be done quickly; for example, making a page read-only no longer requires a serial scan of all PTEs.
7. It supports large (2MB) pages, an important performance benefit.

8. It uses fine grain locking to improve SMP performance.
9. It supports live migration by implementing dirty page logging.

The XI code is only used for fully virtualized guest domains. The initial implementation only supports the 64-bit hypervisor.

These restrictions reflect Virtual Iron's current priorities.

3 Nomenclature

In this document *guest* refers to the guest domain's view of its environment, and *shadow* refers to the hypervisor's view. Sometimes we use *hardware* rather than *shadow* to emphasize that the hypervisor is maintaining the real machine state.

A *page table* (PT) is a 4KB block of memory containing *page table entries* (PTEs).

GPT is a guest page table. GPTE is a guest page table entry. GP is a guest page. (Not all GPs are GPTs.)

SPT is a shadow page table. SPTE is a shadow page table entry. (All shadow pages are SPTs because page tables are the only things we shadow.)

Page tables form a *page table hierarchy* or, simply, *hierarchy*. Depending on the guest's memory model, its hierarchy can be 2, 3 or 4 levels. Correspondingly the shadow hierarchy is 3, 3 or 4 levels. (In XI, the shadow hierarchy is never just 2 levels.)

A page table or page table entry can be distinguished by its level; for example, L3 GPT refers to a level 3 guest page table, and L1 SPTE refers to a level 1 shadow page table entry.

At any time the top level page table is pointed to by control register 3 (CR3). The hypervisor maintains a guest CR3 value, which points to the top level guest page table, and a hardware CR3 value, which points to the top level shadow page table.

The sense in which page tables form a hierarchy is dynamic, not static. That is, starting at the top-level page table (pointed to by CR3) one can descend 2, 3, or 4 levels (depending on the memory model), traversing other page tables and reaching all the addressable guest pages. However, the page tables form a directed graph, not a tree. This is because a page table can be pointed to by multiple page table entries.

In fact, guest page tables often contain cycles, as discussed in section 4.2.7. But shadow page tables are acyclic: non-leaf entries point to other lower-level SPTs and leaf entries point to guest pages.

Some people refer to the entire hierarchy as a "page table" and the individual pages as "page table pages". This is document tries to avoid that nomenclature.

4 Data Structures

4.1 GPI – Guest Page Info

XI maintains two data structures – SPTIs and BLIs (discussed respectively in sections 4.2 and 4.3) – which it associates with guest pages. A GP has one or more SPTIs if it is a

GPT (and none if it is not), and it has one BLI for each address mapping (and no BLIs if it is not addressable).

To provide fast access from a guest page to its SPTIs and BLIs, XI creates a singly-linked list for each GP. The list links together a GP's SPTIs and BLIs.

The elements of this list are of type struct gpi. Both SPTIs and BLIs contain an embedded struct gpi.

The list's head is in the GP's page_info element.

"page_info" is the Xen page frame structure, an array with one element per page of machine memory and indexed by MFN. This structure contains a member called "gpi", a pointer which is only used when the corresponding page is a GP or a SPT. The "gpi" member is new with XI, and increases the size of a page_info element from 40 to 48 bytes.

4.2 SPTI – Shadow Page Table Info

This is the most important data structure in the XI implementation.

The purpose of the shadowing code is to keep SPTs in sync with GPTs. This (conceptually) involves walking the guest page table hierarchy from the top down looking for changes. For each GPT that the hypervisor encounters it ensures its shadow page table exists and is in-sync. Given a GPT's address, how does the hypervisor locate the corresponding SPT? The mechanism is provided by SPTIs.

4.2.1 SPTI and Related Data Structures

Each SPT has an associated SPTI. It contains metadata associated with the shadow page. Given any one of a SPT, GPT or SPTI, the data structures allow the others to be quickly accessed, as shown in the following figure.

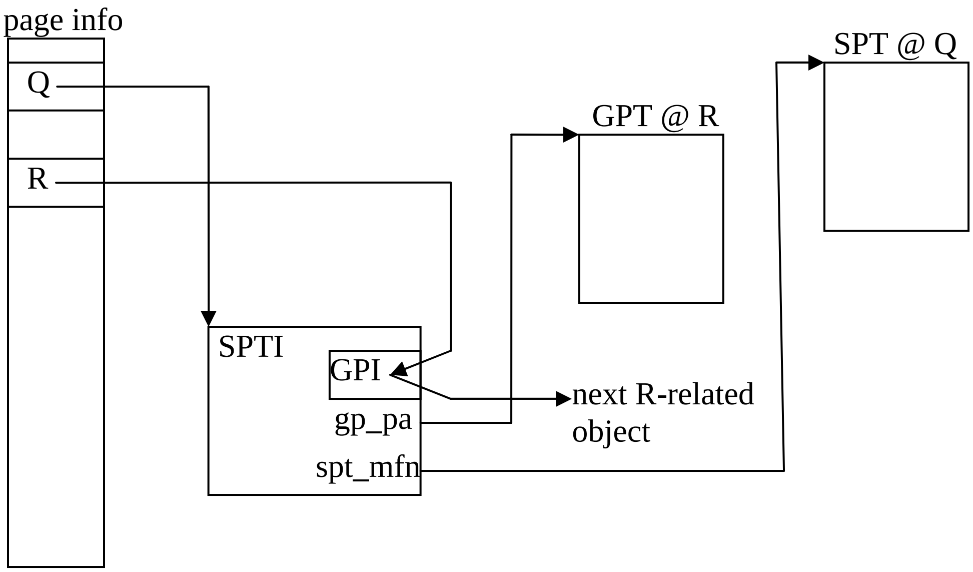

Figure 1 SPTIs and Related Data Structures

Here is how all these pointers relate:

(1) Given a SPT, to find its SPTI: for a SPT at MFN Q, page_info[Q].gpi points to the SPTI.

(2) Given a SPTI, to find the GPT it shadows: the GPT's address is spti->gp_pa.

(3) Given a SPTI, to find its SPT: the SPT's address is spti->spt_mfn.

(4) Given a GPT at MFN R, page_info[R].gpi points to a linked-list of GPI structures and SPTI is on that list. Other guest-page-related objects are also on that list.

4.2.2 SPT Allocation

A shadow page table (SPT) is allocated whenever the hypervisor encounters a new guest page table (GPT). When the hypervisor is bringing SPTs back into sync with GPTs, it walks the guest page hierarchy. In doing so it might encounter a guest page table for which it has no shadow, in which case it allocates a SPTI and a SPT.

The SPT is a real page table: it is linked into the shadow page table hierarchy which is walked by the hardware. The SPTI contains metadata that associates the GPT, SPT and SPTI together, as described above, and other information, as described below.

For guests using the 64-bit and 32-bit PAE models, there is a single 4KB SPT for each GPT. For guests using the 32-bit non-PAE model, 8KB of L1 SPT are allocated for each L1 GPT and 16KB of L2 SPT are allocated for each L2 GPT. This difference is discussed in section 4.2.11.

4.2.3 SPT Reference Counts

The life of a SPT is managed using a reference count contained in its SPTI. This count contains – for top-level SPTs – plus 1 for each ASI that points to it (as discussed in section 4.4) and – for other SPTs – plus 1 for each higher-level SPT that points to it.

When the hypervisor modifies a level N SPTE to point to a level N-1 SPT, it increments the reference count in the SPT's SPTI. Likewise when it removes a pointer to a lower-level SPT, it decrements its SPTI's reference count.

When a SPTI reference count reaches zero it means, as far as the hypervisor knows, the guest page is no longer a guest page table. The hypervisor is allowed to free the shadow page and its SPTI. However, the SPT's entries might point to subordinate SPTs. Before freeing the shadow page, the hypervisor dereferences any subordinate SPTs, which might leave them unreferenced and freeable. In this way, freeing a SPT can cascade down the hierarchy causing a whole subtree to be freed.

4.2.4 SPT Aging

It is desirable to cache shadow hierarchies for a short while even if they are not referenced. A hierarchy might be used by a guest process that is not currently running but will be rescheduled shortly. Retaining its shadow hierarchy avoids the needless overhead of tearing it down and reconstructing it. This caching is implemented through SPT aging.

When a top-level shadow page table (level 3 or 4) is used in handling a page fault, its SPTI is marked as young, timestamped, appended to an aging list and its reference count is incremented.

Periodically the aging list is checked. A SPTI is considered old if its timestamp indicates it hasn't been recently used in handling a page fault. It is marked as 'not young', removed from the aging list, and its reference count is decremented.

The aging list has two effects:

1. An address space (discussed in section 4.4) will be deleted if it is inactive and its top-level SPT is old.
2. A young level 3 SPT that is not pointed to by a level 4 SPT will be retained by virtue of its increased reference count. This is useful in older versions of Linux where the system uses a single level 4 page table (that is, it never reloads CR3 with a different value) but instead modifies L4 PTEs when switching contexts.

4.2.5 SPT Locking

There is a per-domain spinlock – the spt_lock – that protects the various pointers between SPT data structures, SPT reference counts and other SPTI state. For example before descending the shadow page table hierarchy, functions must acquire the spt_lock. This lock is held for short periods of time.

Once a function starts operating on a specific SPT, it typically releases the spt_lock and busies its SPTI. The SPTI contains a 'busy bit' which provides a thread with exclusive access to the SPT.

4.2.6 GPTs and Their Related SPTs

A guest page table (GPT) might have more than one shadow page table (SPT). This happens in several ways, briefly mentioned here and described more fully just below:

(1) when a GPT is dynamically used at multiple levels in the page hierarchy,

(2) when the guest changes memory models,

(3) when a GPT coincides with a large (2MB) page,

(4) when a 32-bit PAE guest page contains multiple level 3 GPTs.

The several SPTIs (usually only one) associated with a GPT are linked together using GPI links, as shown in Figure 1, and accessible from the GPT's page_info[mfn].gpi.

4.2.7 Linear Page Table

This section describes how a GPT containing a loop can result in multiple SPTs.

It is common practice for an operating system to create a linear page table by having one entry in the top-level PT point to that PT itself. This has the effect of making all the page tables themselves addressable like any other page. This is shown in the following figures for both the guest and shadow hierarchies.

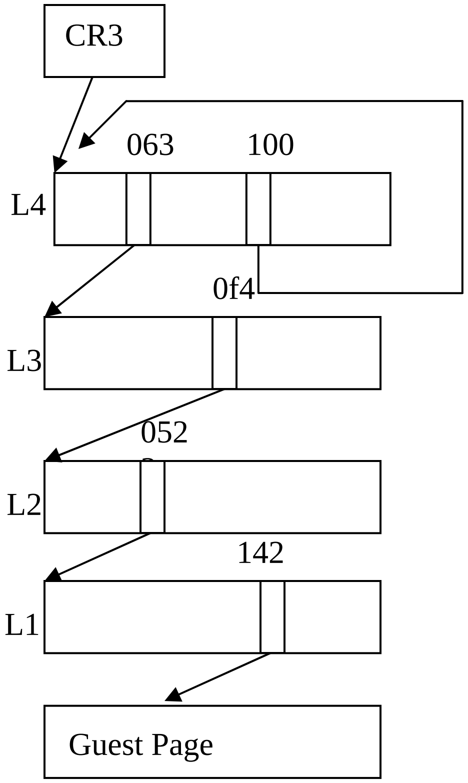

Figure 2 Guest Linear Page Table

In this example, we have a 4 level hierarchy with each page table having 512 entries. Since each page table is accessed by 9 bits of the virtual address, we show a 48 bit virtual address as four 9-bit nontets and three hex digits.

The guest page is accessed using virtual address 063.0f4.052.142.000.

The level 1 page table is accessed using virtual address 100.063.0f4.052.000.

The level 2 page table is accessed using virtual address 100.100.063.0f4.000.

The level 3 page table is accessed using virtual address 100.100.100.063.000.

The level 4 page table is accessed using virtual address 100.100.100.100.000.

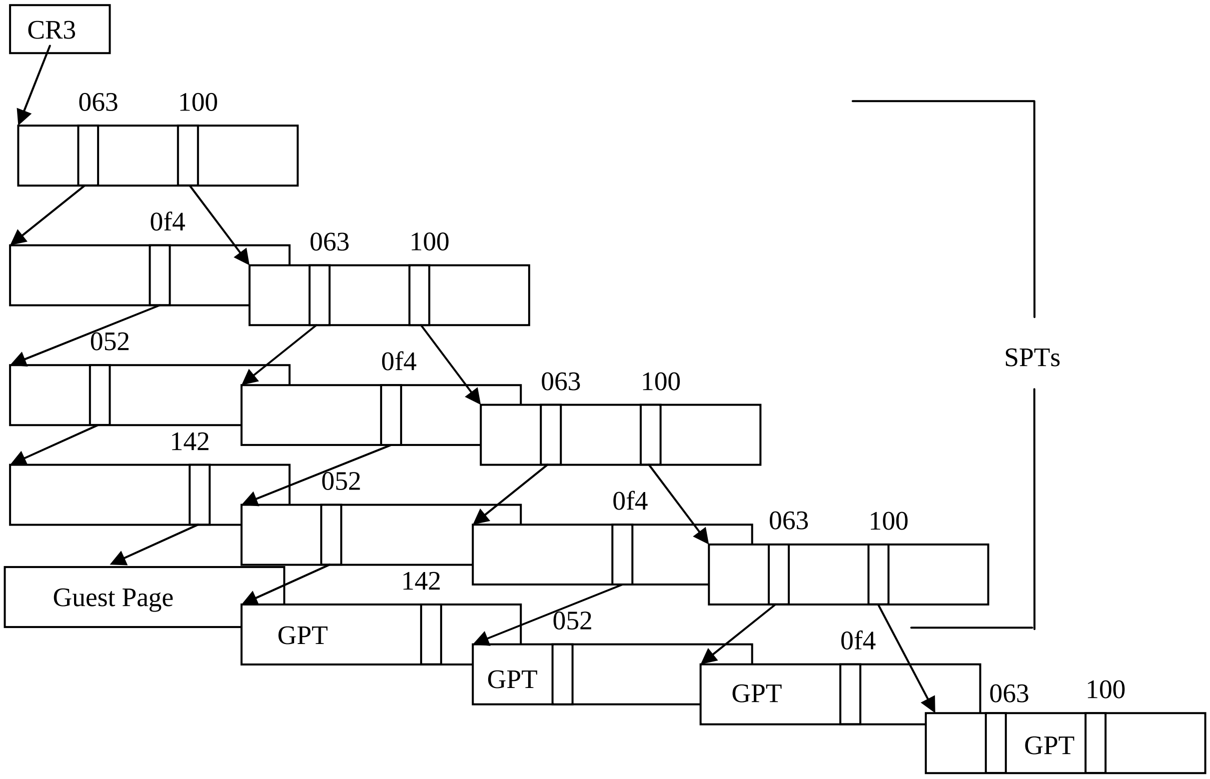

Figure 3 Shadow of Linear Page Table

However, this cyclic trick does not work with the shadow hierarchy. There, non-leaf SPTEs point to other SPTs (which are in the hypervisor's address space) and only leaf SPTEs point to guest pages.

In other words, when the hardware walks the shadow hierarchy it must "bottom out" at a guest page. If shadow pages had loops, it would bottom out at a shadow page (and allow the guest access to hypervisor memory!).

So the guest hierarchy shown in Figure 2 results in a shadow hierarchy as shown in Figure 3. Note that only the leaf pages are guest pages; all the non-leaf pages are SPTs. Also note that corresponding to the L4 GPT there are four SPTs, one at each level.

This demonstrates how a GPT cycle can result in multiple SPTs.

So in our example what really is the level of the L4 GPT? When the cycle is taken (nontet = 100), the GPT also serves as a L3, L2 and L1 GPT and as a guest page. The level of a GPT is determined by *how it is used*. In contrast, the level of a SPT is *fixed* when it is first created. When a SPT is created, its SPTI is branded with the SPT's level.

4.2.8 Changing Guest Memory Model

This section describes how switching memory models can result in multiple SPTs for a GPT.

The memory model of a guest is determined by certain control registers. A guest might start with a two level hierarchy, switch to three levels, and then to four. The memory model is processor-specific, so on a SMP machine the various virtual processors in one domain might be using different memory models!

This means that the same GPT might be interpreted simultaneously (by various processors) as being at different levels in different model hierarchies.

The shadow code accommodates this by branding each new SPTI with the number of levels currently in use by the virtual processor. When the hypervisor is looking for an SPT corresponding to some GPT, it only considers SPTs whose number of levels matches the current virtual processor's usage.

4.2.9 Large Pages

This section describes how XI supports large pages and how large pages can result in multiple SPTs for a GPT.

Level 1 page table entries are always leaf entries: they point to 4KB pages, not page tables. A level 2 page table entry can also be a leaf entry. If its PSE flag is set, the entry points to a 2MB page (or 4MB in 32-bit non-PAE).

Using large pages improves performance because only a single page table walk is needed to set a TLB which then applies to the whole large page. They also reduce the amount of storage and time consumed by the shadow code. For this reason, XI attempts to use large pages whenever possible.

When XI encounters a L2 GPTE with the PSE flag set, it normally creates a L2 SPTE with the PSE flag set. As usual, for the PFN in the L2 GPTE, it substitutes the corresponding MFN in the L2 SPTE.

However, several constraints must be met:

1. The guest large page is meant to include the 512 4KB pages starting at the specified PFN. The MFNs that correspond with those PFNs must be contiguous and the first must be 2MB-aligned. If the initial MFN is not 2MB aligned or any of the MFNs is non-contiguous or any of the MFNs is invalid, the shadow code does not treat the guest page as a large page. During domain initialization, the hypervisor can determine all the PFNs that meet this constraint. It stores the results in a bit array for later reference.
2. None of the 4KB pages can be used as a GPT. A large page is either entirely writable or not, based on the SPTE's RW flag. But, as described in section 5.3, the resync logic needs to make individual GPTs writable or read-only. So, if a large guest page would contain a GPT, the shadow code does not treat it as a large page.
3. Dirty page logging must be disabled. As described in section 5.5, when dirty page logging is enabled the hypervisor is monitoring writes to each 4KB page, so it cannot treat guest pages as large pages.

When the hypervisor cannot treat a large guest page as a large shadow, it constructs a "fake" level 1 shadow page table. Where the L2 GPTE has the PSE flag set and points to a large guest page, the L2 SPTE has the PSE flag clear and points to a fake L1 SPT. Then the fake L1 SPT entries point to the 512 individual guest pages.

The large guest page's address is the address with which the fake L1 SPT is associated (via the page_info.gpi list as shown in Figure 1).

Now the first 4KB of the guest's large page might actually contain a GPT. If so it is shadowed by a SPT, which is associated with that same guest page. In other words, we might have a fake L1 SPT and a (real) SPT associated with the same guest page. The former exists because some L2 GPTE treats that 2MB region as a large page; the latter exists because the first 4KB of the region contains a GPT.

Hence large pages can result in multiple SPTs for the same guest page.

4.2.10 32-bit PAE Mode

When the guest is using this memory model, the guest hierarchy has 3 levels and a guest virtual address is only 32 bits. CR3 points to the L3 GPT which contains only 4 8-byte entries. VA[31:30] provides the index. A L3 GPT is 32 bytes and must be 32-byte aligned. This is exceptional; all other paging structures are 4K bytes and must be page aligned.

The L2 GPT contains 512 8-byte entries and is indexed by VA[29:21]. The L1 GPT contains 512 8-byte entries and is indexed by VA[20:12]. A L1 GPTE points to a specific 4KB page and VA[11:0] indexes to a specific byte within it.

XI maintains a 3 level shadow hierarchy. The L3 SPT occupies a whole page but only the first 4 entries are used. The L2 and L1 SPTs have the same geometry as their guest counterparts.

A guest page might contains multiple (up to 128) L3 GPTs. XI would maintain a separate shadow page for each, rather than having a single shadow page containing multiple SPTs. Here is the final example where a single GP can have multiple SPTs.

L3 GPTEs are odd in another way: they contain no RW or Accessed flags. In all other hierarchies, there is a RW flag at each level and a guest page is only writable if all the RW flags are set. However, for 32-bit PAE there is no way to make read-only the 1GB VA space addressed by a single L3 GPTE. Likewise there is no way to record that a 1GB VA space has been accessed.

4.2.11 32-bit Non-PAE Mode

This mode allows the guest to address only 4GB of memory because the guest PTEs are only 32 bits. To avoid that same limitation, the shadow implementation uses 64-bit SPTEs. The unfortunate result is that the guest and shadow hierarchies have completely different geometries! This is illustrated in Figure 4.

The shadow maintains a three level PAE hierarchy into which it maps the guest's two level non-PAE hierarchy.

The following table describes how various bits within a 32-bit virtual address are used to index page tables, both guest and shadow.

For emphasis we use commas to separate dectets (which is how the guest views the bits), dots to separate nontets (which is how the shadow views the bits) and spaces to keep the guest and shadow views nicely aligned.

```
 33 22222222 2 211111111110000000000 - VA bit #
 10 98765432 1 0987654321 9876543210

 22 22222222,1 111111111bbbbbbbbbbbb - level of Guest Table indexed
 33.22222222 2.111111111bbbbbbbbbbbb - level of Shadow Table indexed
```

VA[11: 0] index to a byte within a page.

VA[20:12] index into a L1 SPT, which has 512 entries. Note that each L1 guest page table has 1024 entries, so it takes a pair of shadow pages to represent a L1 GPT. (However, each SPT entry is 64-bits and is capable of addressing all physical memory – whereas a GPT entry can only address 4GB – which is the whole reason for always using PAE in the shadow.)

VA[29:21] index into a L2 SPT. Just as a pair of L1 SPTs correspond with one L1 GPT, a pair of L2 SPT entries correspond with one L2 GPT entry.

VA[31:30] index into the L3 SPT. Its four entries are constant and are initialized to point to 4 pre-allocated L2 SPTs. (These are allocated in contiguous physical memory which simplifies indexing.)

For example:

VAs in the range 0XAE400000 .. 0XAE5FFFFF map into a single L2 GPT entry which points to a L1 GPT.

That same range maps into a pair of L2 SPTEs each of which points to a L1 SPT.

```
Guest  L2[1010111001,]             --> L1[10 10111001,0 000000000
                                      ... 10 10111001,1 111111111]

Shadow L3[10.]--> L2[10.101110010.]--> L1[10.10111001 0.000000000
                                      ... 10.10111001 0.111111111]
                  L2[10.101110011.]--> L1[10.10111001 1.000000000
                                      ... 10.10111001 1.111111111]
```

Again, commas separate dectets, dots separate nontets, and spaces provide column alignment.

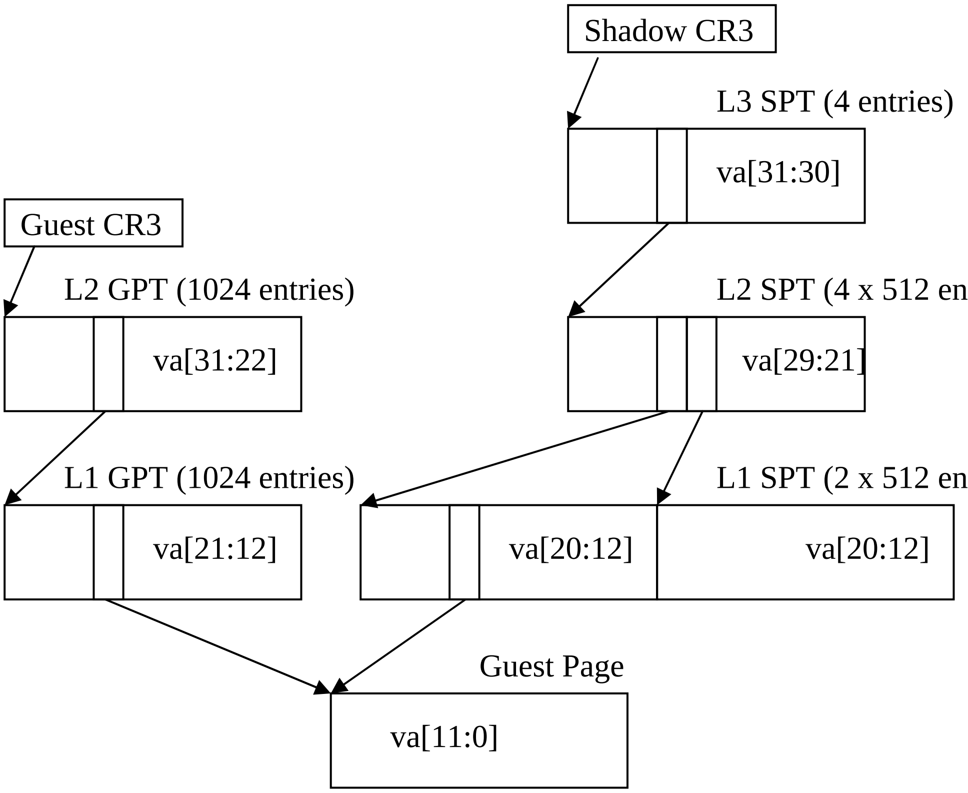

Figure 4 32-bit non-PAE Guest and Shadow Hierarchies

4.3 Backlinks

XI maintains backlinks from guest pages to the SPTEs that provide access to them. These backlinks are used in two algorithms: making pages read-only and dividing large pages.

A GPT is made read-only so the hypervisor can detect when the guest tries to modify it. This lets the hypervisor bookkeep all modified (and therefore out-of-sync) GPTs.

GPs are also made read-only so the hypervisor can bookkeep dirty pages, which is necessary for dirty page logging, a feature in support of live migrate.

The hypervisor divides large pages when it discovers they contain a GPT. As mentioned in section 4.2.9, XI supports large pages but only when they don't contain GPTs. If a GPT is found in a large page, the hypervisor modifies the L2 SPT entry, clearing the PSE flag and pointing the entry to a new fake L1 SPT.

Backlink Information (BLI) is implemented using struct bli. When a SPTI is needed, it is allocated along with its SPT and a BLI array. This array's elements are one-to-one with the SPT entries; for example, with the 64-bit memory model a BLI array would contain 512 elements. The SPTI has a pointer to its BLI array. A BLI contains a pointer to the next BLI for the same GP, and a pointer to the SPTE. These relationships are shown in the following figure:

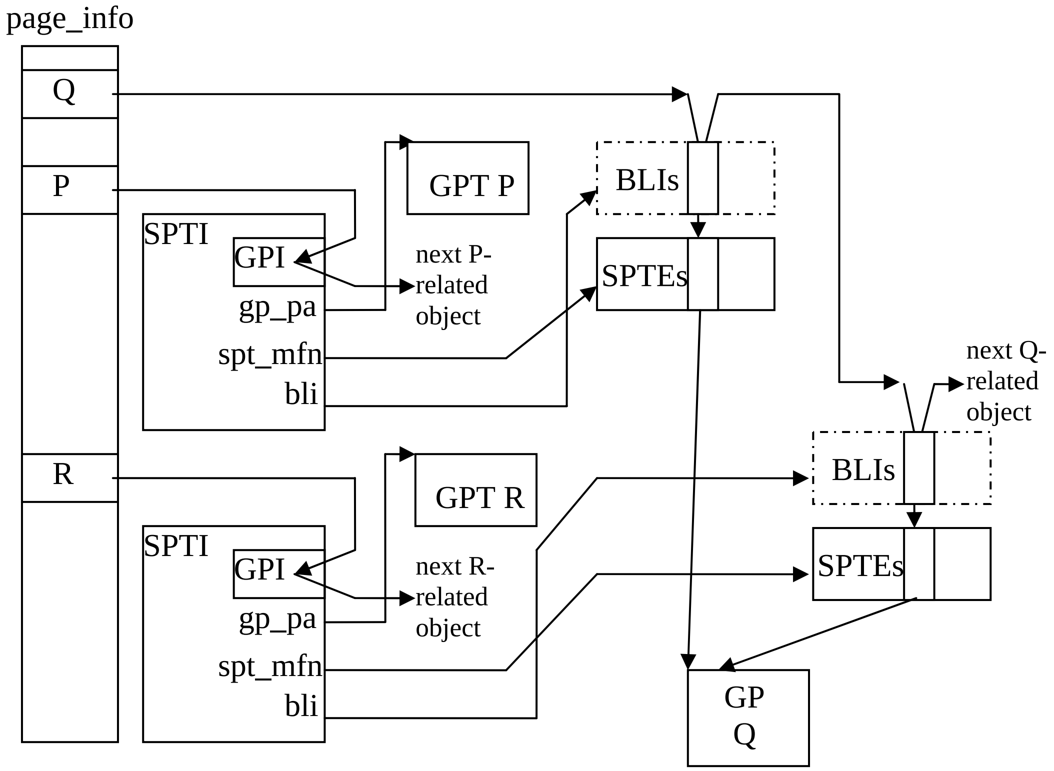

Figure 5 Backlinks and Related Data Structures

In this example, GP Q is twice-mapped. It is mapped by a SPTE in GPT P's shadow page and by a SPTE in GPT R's shadow page.

To find the SPTEs that provide access to GP Q, the hypervisor would walk the linked list with head at page_info[Q].gpi. This list contains a BLI for each SPTE which points to GP Q. (The list would also contain a SPTI if GP Q happened to be a guest page table.)

Each BLI points to its SPTE. So by following these links the hypervisor can locate every SPTE for a guest page. It can mark them all as read-only or it can locate and divide any large pages.

In much the same way, BLIs provide backlinks from each SPT to the SPTEs (in the parent SPTs) that point to it. This is used to determine whether a SPT is active. The hypervisor can follow a chain of backlinks from any SPT up to the top-level SPT. From there it can check if the owning ASI is active. XI tries to retain active SPTs and keep them insync; whereas it discards inactive SPTs if they go out of sync.

4.4 Address Spaces

To improve performance, XI caches multiple address spaces. This means when the guest reloads CR3 to switch to a previously used address space any previously created shadow pages might still be available.

The notion of an address space is implemented using an Address Space Information (ASI) block, which is of type struct asi.

An ASI is bound to a particular guest CR3 value and a top-level SPT. That, in turn, is linked to all the lower-level SPTs.

When a VCPU loads a particular CR3 value, it starts using any cached shadow pages that remain from the previous use of that CR3 value.

An ASI is discarded when it is both inactive and its top-level SPT is ‘not young’ (as discussed in section 4.2.4). Discarding an ASI dereferences its top-level SPT, which can make it freeable and thus make some or all of its subordinate SPTs freeable.

4.5 Domain Extensions

XI requires additional domain-related storage. This is placed in a structure of type struct dom_xi_ext. There is a pointer to this structure in the domain structure. (This level of indirection helps keep the XI code disentangled from the rest of the Xen code and avoids putting a storage burden on non-XI domains.)

5 Algorithms

5.1 Accessed and Dirty Flag Management

XI tries to emulate the hardware’s management of PTE Accessed and Dirty flags. The hardware sets a PTE’s Accessed flag whenever the page is accessed.

(Of course the access must be via that PTE. A page might be pointed to by several PTEs. Only the PTE actually used has its Accessed flag set.)

Similarly the hardware sets a PTE’s Dirty flag whenever the page is written.

Accessed flags exist at each level of the hierarchy and an access sets them all. The Dirty flag is only set in the leaf PTE. This is usually the L1 PTE but, for L2 PTEs with the PSE flag set, the hardware sets the L2 PTE Dirty flag.

These flags are never cleared by the hardware, only by software. Once cleared, the flag is set again when the page is next accessed (or dirtied).

XI emulates this by setting the Accessed flag in a GPTE during a read fault, and the Accessed and Dirty flags during a write fault. To ensure such faults occur when the page is first accessed or dirtied, the hypervisor is said to “provoke” them. It provokes such faults by initially clearing the Present and RW flags in the shadow PTE. Any attempt by the guest to read the page results in a fault. The hypervisor responds by setting the GPTE’s Accessed flag and, to allow the access, the SPTE’s Present flag. Similarly, any attempt by the guest to write the page results in a fault. The hypervisor responds by setting the GPTE’s Accessed and Dirty flags and, to allow the write, the SPTE’s Present and RW flags.

The hypervisor never clears the GPTE’s Accessed flag but when, during resync, it notices that the guest has cleared it, the hypervisor clears the SPTE’s Present flag, thus starting the cycle again. These state transitions are illustrated in the following figure.

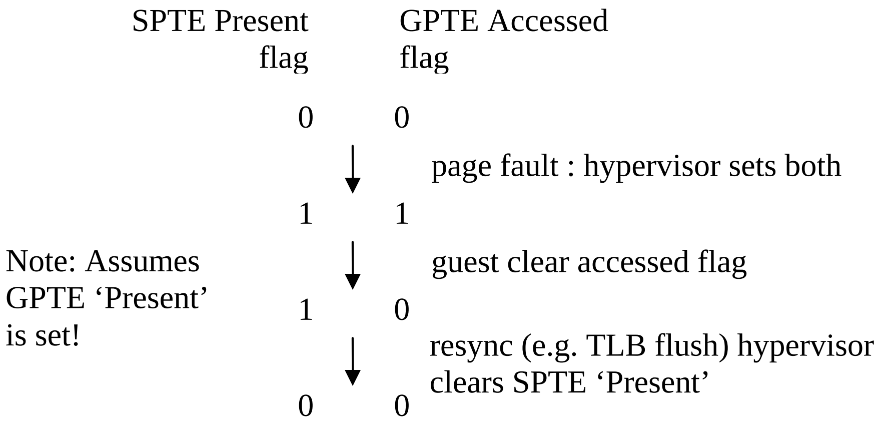
SPTE Present flag
GPTE Accessed flag
0 0
page fault : hypervisor sets both
1 1
guest clear accessed flag
1 0
resync (e.g. TLB flush) hypervisor clears SPTE 'Present'
0 0
Note: Assumes GPTE 'Present' is set!

Figure 6 Management of the GPTE Accessed flag

The Dirty and RW flags go through a similar state transition.

To facilitate fast handing of a provoked page fault, XI uses two of the available SPTE flags to record the guest PTE's Present and RW flags. The software names these G_PRESENT and G_RW.

When a read fault occurs (due to SPTE 'Present' being clear) the fault handler tests if G_PRESENT is set. If so the fault was provoked. The handler can simply set both GPTE Accessed and SPTE 'Present' and it is done. (Likewise for G_RW, RW and GPTE Dirty.)

5.2 Page Fault Handling

A page fault occurs under four conditions:

1. A read fault : the guest tried to access memory but a 'not present' SPTE entry was found at some level in the shadow hierarchy;
2. A write fault : the guest tried to write memory but a 'not RW' SPTE entry was found at some level in the shadow hierarchy;
3. A no-execute fault : the guest tried to execute code but a 'no execute' SPTE entry was found at some level in the shadow hierarchy;
4. A reserved flag fault : the hardware found a reserved bit set in some SPTE entry at some level in the shadow hierarchy.

There is a fast path through the fault handler, as described in section 5.1. If the handler detects that the fault was provoked, it takes the action described in that section, and returns.

The fast path also handles modified GPTs, as described in section 5.3.

Otherwise the handler assumes one of the SPTEs in the hierarchy is out of sync. It processes the out-of-sync list (as discussed in section 5.3), walks down the guest hierarchy, resyncs the corresponding SPTEs. If resynching has solved the problem, the handler returns with success otherwise it returns with a failure. A failure causes the page fault to be propagated to the guest.

Read faults are quite straight forward: If the guest marks a GPTE as ‘present’ and tries to access the now-present page, a read fault occurs because the shadow PTE is out-of-sync. The handler will resync the shadow PTE, determine that that solved the problem and return success.

Write faults work similarly when the guest has set the RW flag in a GPTE.

A no-execute fault occurs when the guest attempts to execute from memory that is marked ‘no execute’. Note that it is the guest itself that marked the memory as ‘no execute’. It does so by setting the EXB[1] flag (bit 63) in a guest PTE. (This flag is only available with the 64-bit and 32-bit PAE memory models.) XI faithfully copies this flag from the guest PTE into the shadow PTE. The fault handler responds to a no-execute fault by resynching the SPTEs, which might clear the EXB flag if an SPTE was out-of sync. If that fails, the handler returns with a failure and the fault is propagated to the guest.

A reserved flag fault occurs when some reserved bit in a SPTE is set. Naively this should never happen – XI would not construct an invalid SPTE, right? – but there is one case where it is expected.

XI copies the EXB flag from GPTEs into SPTEs. This flag is only permitted when the no-execute feature has been enabled in the VCPU. If a reserved flag fault occurs because the SPTE contains an out-of-sync EXB flag, resynching will clear the flag and the fault handler can return success. If resynching does not clear the flag, then the guest has incorrectly specified EXB, and the fault handler returns failure thus propagating the page fault to the guest.

5.3 Out-of-Sync Processing

Section 5.2 discussed changes by the guest to GPTEs to increase access. To recap, if the guest changes GPTEs to make memory accessible, writable or executable, the corresponding shadow PTEs are left out-of sync. When the guest actually tries to access, write or execute the memory, a page fault occurs. The fault handler resyncs the pertinent entries and returns success.

However, when the guest changes GPTEs to make memory less accessible or to change what GPTEs point to, the guest must explicitly indicate that the shadow is out of sync. It does so by flushing TLBs. This triggers XI’s out-of-sync processing.

The OOS processor examines all modified GPTs and brings their shadow PTs back into sync. The hypervisor maintains a list of all GPTs that have been modified since the last OOS processing. This is done as follows: GPTs are normally kept read-only. When the guest attempts to modify one, a page fault occurs. The page fault handler simply marks the GPT’s SPT as out-of-sync, makes the GPT writable, and returns success.

To bring a SPT into sync with its GPT, the hypervisor steps through each of the individual SPTEs and GPTEs, makes a quick decision of what actions (if any) are need to update the SPTE, and performs those actions. The decision process is described in section 5.4.

[1] Called EXB by Intel and NX by AMD.

Once a SPT has been resynched, it is marked as in-sync and its GPT is made read-only again.

5.4 SPTE Decision Table

Bringing a SPTE into sync with its GPTE can be complex. It involves many conditions. To name just a few:

- Is the current SPTE marked ‘Present’?
- Is the current GPTE marked ‘Accessed’?
- Has the GPTE’s physical address changed?

These conditions affect the content of the new SPTE, whether new subordinate SPTs should be constructed or dereferenced, etc.

The rules by which the conditions affect the requisite actions are very complex. This complexity can spread throughout the shadow code making it a jungle of tests and special cases.

XI attempts to control this complexity by using a decision table. The decision table specifies what actions are needed to bring a SPTE into sync with its GPTE. The algorithm gathers together a large number of simple (mostly binary) conditions to form an index, uses the index to access the decision table, and takes actions based on the selected decision table entry.

Here in summary are the conditions considered and the possible actions. For details, please consult the code.

Conditions
• number of guest paging levels
• current SPT level
• SPTE Present flag
• SPTE PSE flag
• SPTE Fake flag
• GPTE Accessed flag
• GPTE Dirty flag
• GPTE Present flag
• GPTE PSE flag
• Phys addr change (GPTE vs SPTE)
• GPTE phys addr valid
• GPTE phys addr is insync GPT

Actions
• attach backlink
• detach backlink
• deref subordinate SPTI
• ref subordinate SPTI
• initialize L2 SPTE large page
• keep fake L1 SPT
• retain SPTE phys addr
• use GPTE’s MFN as phys addr
• mark SPTE as Present
• mark SPTE as RW
• GPTE’s PFN is bogus so clear various SPTE flags

- GPTE phys addr being logged

5.5 Dirty Page Logging

To support live migrate, XI implements dirty page logging. When enabled all pages are made read-only. As the guest modifies pages, a bitmap records the dirtied pages. From time to time, the migrate application can read the bitmap to learn which pages must be re-sent. It can then clear the bitmap and mark all pages as read-only again.

Because this bookkeeping is done for each 4KB page, large pages are problematic: we don't want to re-send 2MB of data when we might only need to re-send 4K bytes. Accordingly, when dirty logging is enabled, XI replaces all large pages with 4KB pages. This involves changing all L2 SPTEs with PSE to point to new fake L1 SPTs.

[end]