Michael S. Harvey
Leonard S. Szubowicz

Extending OpenVMS for 64-bit Addressable Virtual Memory

The OpenVMS operating system recently extended its 32-bit virtual address space to exploit the Alpha processor's 64-bit virtual addressing capacity while ensuring binary compatibility for 32-bit nonprivileged programs. This 64-bit technology is now available both to OpenVMS users and to the operating system itself. Extending the virtual address space is a fundamental evolutionary step for the OpenVMS operating system, which has existed within the bounds of a 32-bit address space for nearly 20 years. We chose an asymmetric division of virtual address extension that allocates the majority of the address space to applications by minimizing the address space devoted to the kernel. Significant scaling issues arose with respect to the kernel that dictated a different approach to page table residency within the OpenVMS address space. The paper discusses key scaling issues, their solutions, and the resulting layout of the 64-bit virtual address space.

The OpenVMS Alpha operating system initially supported a 32-bit virtual address space that maximized compatibility for OpenVMS VAX users as they ported their applications from the VAX platform to the Alpha platform. Providing access to the 64-bit virtual memory capability defined by the Alpha architecture was always a goal for the OpenVMS operating system. An early consideration was the eventual use of this technology to enable a transition from a purely 32-bit-oriented context to a purely 64-bit-oriented native context. OpenVMS designers recognized that such a fundamental transition for the operating system, along with a 32-bit VAX compatibility mode support environment, would take a long time to implement and could seriously jeopardize the migration of applications from the VAX platform to the Alpha platform. A phased approach was called for, by which the operating system could evolve over time, allowing for quicker time-to-market for significant features and better, more timely support for binary compatibility.

In 1989, a strategy emerged that defined two fundamental phases of OpenVMS Alpha development. Phase 1 would deliver the OpenVMS Alpha operating system initially with a virtual address space that faithfully replicated address space as it was defined by the VAX architecture. This familiar 32-bit environment would ease the migration of applications from the VAX platform to the Alpha platform and would ease the port of the operating system itself. Phase 1, the OpenVMS Alpha version 1.0 product, was delivered in 1992.[1]

For Phase 2, the OpenVMS operating system would successfully exploit the 64-bit virtual address capacity of the Alpha architecture, laying the groundwork for further evolution of the OpenVMS system. In 1989, strategists predicted that Phase 2 could be delivered approximately three years after Phase 1. As planned, Phase 2 culminated in 1995 with the delivery of OpenVMS Alpha version 7.0, the first version of the OpenVMS operating system to support 64-bit virtual addressing.

This paper discusses how the OpenVMS Alpha Operating System Development group extended the OpenVMS virtual address space to 64 bits. Topics covered include compatibility for existing applications, the options for extending the address space, the strategy for page table residency, and the final layout of the OpenVMS 64-bit virtual address space. In implementing support for 64-bit virtual addresses, designers maximized privileged code compatibility; the paper presents some key measures taken to this end and provides a privileged code example. A discussion of the immediate use of 64-bit addressing by the OpenVMS kernel and a summary of the work accomplished conclude the paper.

Compatibility Constraints

Growing the virtual address space from a 32-bit to a 64-bit capacity was subject to one overarching consideration: compatibility. Specifically, any existing nonprivileged program that could execute prior to the introduction of 64-bit addressing support, even in binary form, must continue to run correctly and unmodified under a version of the OpenVMS operating system that supports a 64-bit virtual address space.

In this context, a nonprivileged program is one that is coded only to stable interfaces that are not allowed to change from one release of the operating system to another. In contrast, a privileged program is defined as one that must be linked against the OpenVMS kernel to resolve references to internal interfaces and data structures that may change as the kernel evolves.

The compatibility constraint dictates that the following characteristics of the 32-bit virtual address space environment, upon which a nonprivileged program may depend, must continue to appear unchanged.[2]

- The lower-addressed half (2 gigabytes [GB]) of virtual address space is defined to be private to a given process. This process-private space is further divided into two 1-GB spaces that grow toward each other.
  1. The lower 1-GB space is referred to as P0 space. This space is called the program region, where user programs typically reside while running.
  2. The higher 1-GB space is referred to as P1 space. This space is called the control region and contains the stacks for a given process, process-permanent code, and various process-specific control cells.
- The higher-addressed half (2 GB) of virtual address space is defined to be shared by all processes. This shared space is where the OpenVMS operating system kernel resides. Although the VAX architecture divides this space into a pair of separately named 1-GB regions (S0 space and S1 space), the OpenVMS Alpha operating system makes no material distinction between the two regions and refers to them collectively as S0/S1 space.

Figure 1 illustrates the 32-bit virtual address space layout as implemented by the OpenVMS Alpha operating system prior to version 7.0.[1] An interesting mechanism can be seen in the Alpha implementation of this address space. The Alpha architecture defines 32-bit load operations such that values (possibly pointers) are sign extended from bit 31 as they are loaded into registers.[3] This facilitates address calculations with results that are 64-bit, sign-extended forms of the original 32-bit pointer values. For all P0 or P1 space addresses, the upper 32 bits of a given pointer in a register will be written with zeros. For all S0/S1 space addresses, the upper 32 bits of a given pointer in a register will be written with ones. Hence, on the Alpha platform, the 32-bit virtual address space actually exists as the lowest 2 GB and highest 2 GB of the entire 64-bit virtual address space. From the perspective of a program using only 32-bit pointers, these regions appear to be contiguous, exactly as they appeared on the VAX platform.

Superset Address Space Options

We considered the following three general options for extending the address space beyond the current 32-bit limits. The degree to which each option would relieve the address space pressure being felt by applications and the OpenVMS kernel itself varied significantly, as did the cost of implementing each option.

1. Extension of shared space
2. Extension of process-private space
3. Extension of both shared space and process-private space

The first option considered was to extend the virtual address boundaries for shared space only. Process-private space would remain limited to its current size of 2 GB. If processes needed access to a huge amount of virtual memory, the memory would have to have been created in shared space where, by definition, all processes would have access to it. This option's chief advantage was that no changes were required in the complex memory management code that specifically supports process-private space. Choosing this option would have minimized the time-to-market for delivering some degree of virtual address extension, however limited it would be. Avoiding any impact to process-private space was also its chief disadvantage. By failing to extend process-private space, this option proved to be generally unappealing to our customers. In addition, it was viewed as a makeshift solution that we would be unable to discard once process-private space was extended at a future time.

The second option was to extend process-private space only. This option would have delivered the highly desirable 64-bit capacity to processes but would not have extended shared space beyond its current 32-bit boundaries. The option presumed to reduce the degree of change in the kernel, hence maximizing

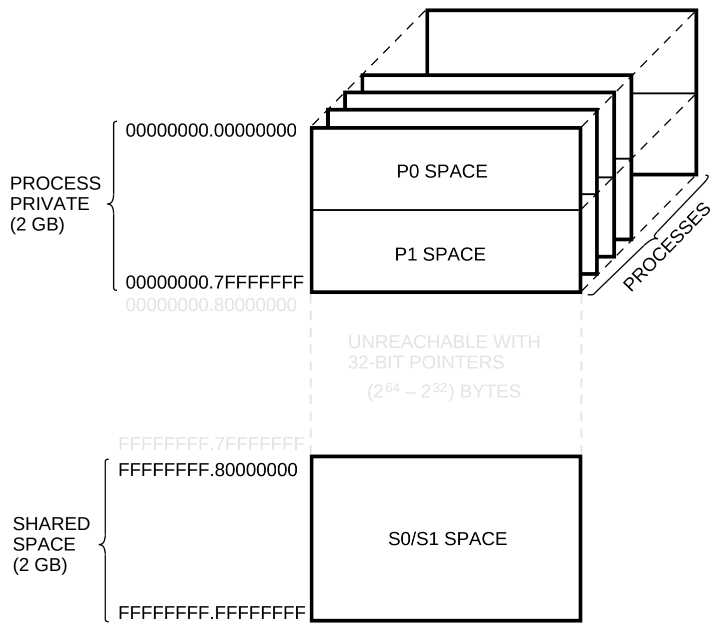


Figure 1
OpenVMS Alpha 32-bit Virtual Address Space

privileged code compatibility and ensuring faster time-to-market. However, analysis of this option showed that there were enough significant portions of the kernel requiring change that, in practice, very little additional privileged code compatibility, such as for drivers, would be achievable. Also, this option did not address certain important problems that are specific to shared space, such as limitations on the kernel's capacity to manage ever-larger, very large memory (VLM) systems in the future.

We decided to pursue the option of a flat, superset 64-bit virtual address space that provided extensions for both the shared and the process-private portions of the space that a given process could reference. The new, extended process-private space, named P2 space, is adjacent to P1 space and extends toward higher virtual addresses.[4,5] The new, extended shared space, named S2 space, is adjacent to S0/S1 space and extends toward lower virtual addresses. P2 and S2 spaces grow toward each other.

A remaining design problem was to decide where P2 and S2 would meet in the address space layout. A simple approach would split the 64-bit address space exactly in half, symmetrically scaling up the design of the 32-bit address space already in place. (The address space is split in this way by the Digital UNIX operating system.[3]) This solution is easy to explain because, on the one hand, it extends the 32-bit convention that the most significant address bit can be treated as a sign bit, indicating whether an address is private or shared. On the other hand, it allocates fully one-half the available virtual address space to the operating system kernel, whether or not this space is needed in its entirety.

The pressure to grow the address space generally stems from applications rather than from the operating system itself. In response, we implemented the 64-bit address space with a boundary that floats between the process-private and shared portions. The operating system configures at bootstrap only as much virtual address space as it needs (never more than 50 percent of the whole). At this point, the boundary becomes fixed for all processes, with the majority of the address space available for process-private use.

A floating boundary maximizes the virtual address space that is available to applications; however, using the sign bit to distinguish between process-private pointers and shared-space pointers continues to work only for 32-bit pointers. The location of the floating boundary must be used to distinguish between 64-bit process-private and shared pointers. We believed that this was a minor trade-off in return for realizing twice as much process-private address space as would otherwise have been achieved.

Page Table Residency

While pursuing the 64-bit virtual address space layout, we grappled with the issue of where the page tables that map the address space would reside within that address space. This section discusses the page table structure that supports the OpenVMS operating system, the residency issue, and the method we chose to resolve this issue.

Virtual Address–to–Physical Address Translation

The Alpha architecture allows an implementation to choose one of the following four page sizes: 8 kilobytes (KB), 16 KB, 32 KB, or 64 KB.[3] The architecture also defines a multilevel, hierarchical page table structure for virtual address-to-physical address (VA–to–PA) translations. All OpenVMS Alpha platforms have implemented a page size of 8 KB and three levels in this page table structure. Although throughout this paper we assume a page size of 8 KB and three levels in the page table hierarchy, no loss of generality is incurred by this assumption.

Figure 2 illustrates the VA–to–PA translation sequence using the multilevel page table structure.

1. The page table base register (PTBR) is a per-process pointer to the highest level (L1) of that process' page table structure. At the highest level is one 8-KB page (L1PT) that contains 1,024 page table entries (PTEs) of 8 bytes each. Each PTE at the highest page table level (that is, each L1PTE) maps a page table page at the next lower level in the translation hierarchy (the L2PTs).
2. The Segment 1 bit field of a given virtual address is an index into the L1PT that selects a particular L1PTE, hence selecting a specific L2PT for the next stage of the translation.
3. The Segment 2 bit field of the virtual address then indexes into that L2PT to select an L2PTE, hence selecting a specific L3PT for the next stage of the translation.
4. The Segment 3 bit field of the virtual address then indexes into that L3PT to select an L3PTE, hence selecting a specific 8-KB code or data page.
5. The byte-within-page bit field of the virtual address then selects a specific byte address in that page.

An Alpha implementation may increase the page size and/or number of levels in the page table hierarchy, thus mapping greater amounts of virtual space up to the full 64-bit amount. The assumed combination of 8-KB page size and three levels of page table allows the system to map up to 8 terabytes (TB) (i.e., 1,024 × 1,024 × 1,024 × 8 KB = 8 TB) of virtual memory for a single process.

To map the entire 8-TB address space available to a single process requires up to 8 GB of PTEs (i.e., 1,024 × 1,024 × 1,024 × 8 bytes = 8 GB). This fact alone presents a serious sizing issue for the OpenVMS operating system. The 32-bit page table residency model that the OpenVMS operating system ported from the VAX platform to the Alpha platform does not have the capacity to support such large page tables.

Page Tables: 32-bit Residency Model

We stated earlier that materializing a 32-bit virtual address space as it was defined by the VAX architecture would ease the effort to port the OpenVMS operating

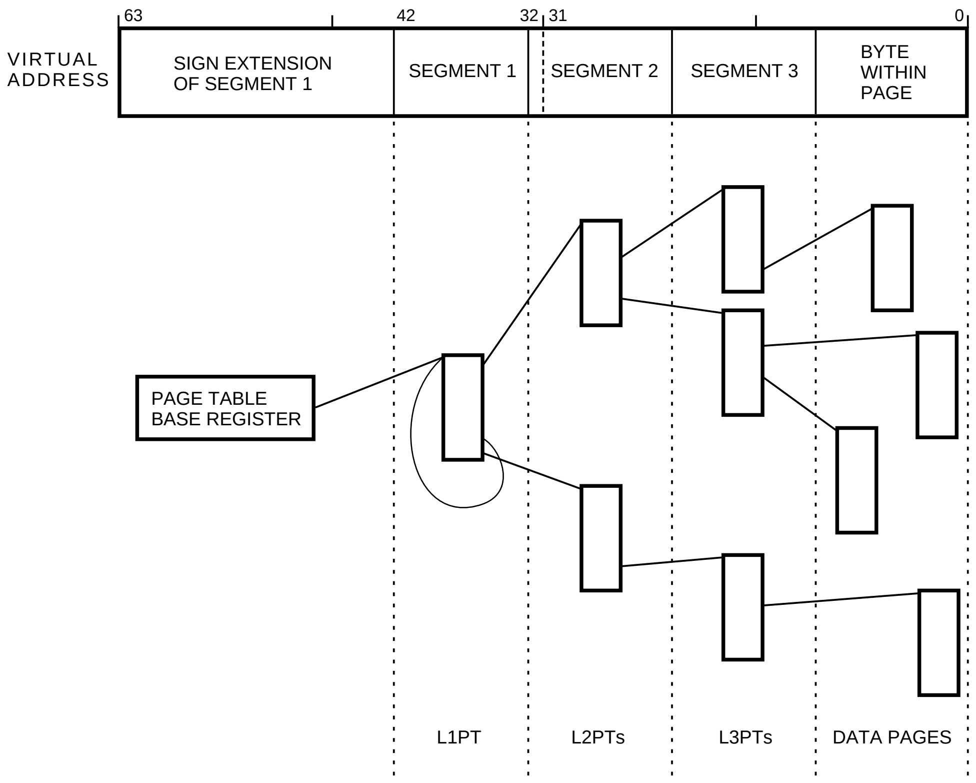


Figure 2
Virtual Address–to–Physical Address Translation

system from the VAX platform to the Alpha platform. A concrete example of this relates to page table residency in virtual memory.

The VAX architecture defines, for a given process, a P0 page table and a P1 page table that map that process' P0 and P1 spaces, respectively.[2] The architecture specifies that these page tables are to be located in S0/S1 shared virtual address space. Thus, the page tables in virtual memory are accessible regardless of which process context is currently active on the system.

The OpenVMS VAX operating system places a given process' P0 and P1 page tables, along with other per-process data, in a fixed-size data structure called a balance slot. An array of such slots exists within S0/S1 space with each memory-resident process being assigned to one of these slots.

This page table residency design was ported from the VAX platform to the Alpha platform.[1] The L3PTs needed to map P0 and P1 spaces and the one L2PT needed to map those L3PTs are all mapped into a balance slot in S0/S1 space. (To conserve virtual memory, the process' L1PT is not mapped into S0/S1 space.) The net effect is illustrated in Figure 3.

The VAX architecture defines a separate, physically resident system page table (SPT) that maps S0/S1 space. The SPT was explicitly mapped into S0/S1 space by the OpenVMS operating system on both the VAX and the Alpha platforms.

Only 2 megabytes (MB) of level 3 PT space is required to map all of a given process' P0 and P1 spaces. This balance slot design reasonably accommodates a large number of processes, all of whose P0 and P1 page tables simultaneously reside within those balance slots in S0/S1 shared space.

This design cannot scale to support a 64-bit virtual address space. Measured in terms of gigabytes per process, the page tables required to map such an enormous address space are too big for the balance slots, which are constrained to exist inside the 2-GB S0/S1 space. The designers had to find another approach for page table residency.

Self-mapping the Page Tables

Recall from earlier discussion that on today's Alpha implementations, the page size is 8 KB, three levels of translation exist within the hierarchical page table structure, and each page table page contains 1,024 PTEs. Each L1PTE maps 8 GB of virtual memory. Eight gigabytes of PT space allows all 8 TB of virtual memory that this implementation can materialize to be mapped.

An elegant approach to mapping a process' page tables into virtual memory is to self-map them. A single PTE in the highest-level page table page is set to map that page table page. That is, the selected L1PTE contains the page frame number of the level 1 page table page that contains that L1PTE.

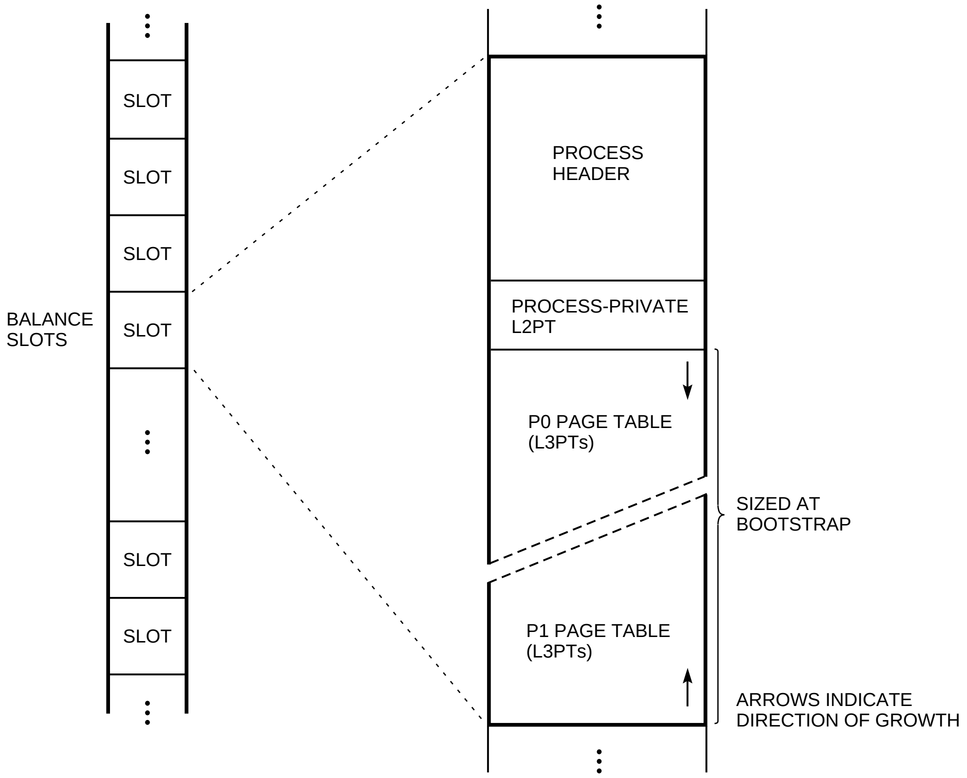


Figure 3
32-bit Page Tables in S0/S1 Space (Prior to OpenVMS Alpha Version 7.0)

The effect of this self-mapping on the VA–to–PA translation sequence (shown in Figure 2) is subtle but important.

- For those virtual addresses with a Segment 1 bit field value that selects the self-mapper L1PTE, step 2 of the VA–to–PA translation sequence reselects the L1PT as the effective L2PT (L2PT′) for the next stage of the translation.
- Step 3 indexes into L2PT′ (the L1PT) using the Segment 2 bit field value to select an L3PT′.
- Step 4 indexes into L3PT′ (an L2PT) using the Segment 3 bit field value to select a specific data page.
- Step 5 indexes into that data page (an L3PT) using the byte-within-page bit field of the virtual address to select a specific byte address within that page.

When step 5 of the VA–to–PA translation sequence is finished, the final page being accessed is itself one of the level 3 page table pages, not a page that is mapped by a level 3 page table page. The self-map operation places the entire 8-GB page table structure at the end of the VA–to–PA translation sequence for a specific 8-GB portion of the process' address space. This virtual space that contains all of a process' potential page tables is called page table space (PT space).[6]

Figure 4 depicts the effect of self-mapping the page tables. On the left is the highest-level page table page containing a fixed number of PTEs. On the right is the virtual address space that is mapped by that page table page. The mapped address space consists of a collection of identically sized, contiguous address range sections, each one mapped by a PTE in the corresponding position in the highest-level page table page. (For clarity, lower levels of the page table structure are omitted from the figure.)

Notice that L1PTE #1022 in Figure 4 was chosen to map the high-level page table page that contains that PTE. (The reason for this particular choice will be explained in the next section. Theoretically, any one

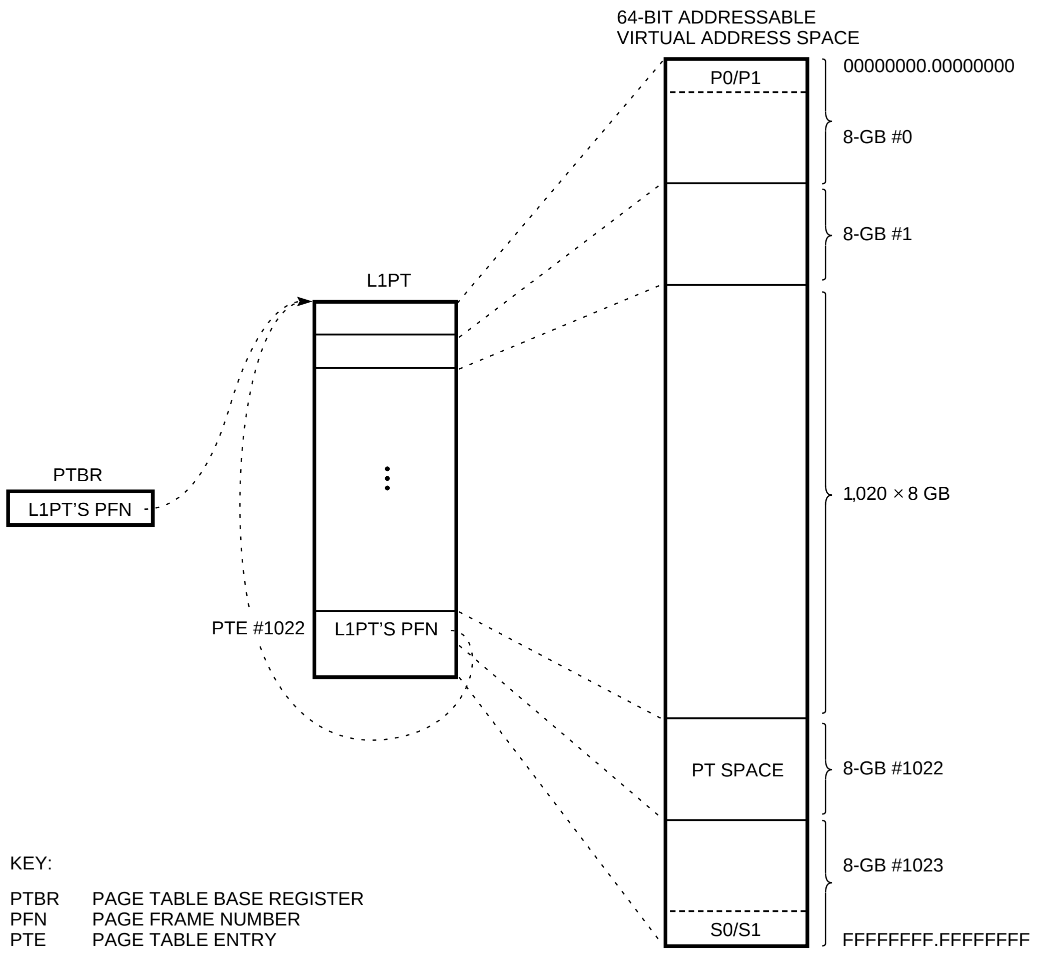


Figure 4
Effect of Page Table Self-map

of the L1PTEs could have been chosen as the self-mapper.) The section of virtual memory mapped by the chosen L1PTE contains the entire set of page tables needed to map the available address space of a given process. This section of virtual memory is PT space, which is depicted on the right side of Figure 4 in the 1,022d 8-GB section in the materialized virtual address space.

The base address for this PT space incorporates the index of the chosen self-mapper L1PTE (1,022 = 3FE(16)) as follows (see Figure 2):

Segment 1 bit field = 3FE
Segment 2 bit field = 0
Segment 3 bit field = 0
Byte within page = 0,

which result in

VA = FFFFFFFC.00000000
(also known as PT_Base).

Figure 5 illustrates the exact contents of PT space for a given process. One can observe the positional effect of choosing a particular high-level PTE to self-map the page tables even within PT space. In Figure 4, the choice of PTE for self-mapping not only places PT space as a whole in the 1,022d 8-GB section in virtual memory but also means that

- The 1,022d grouping of the lowest-level page tables (L3PTs) within PT space is actually the collection of next-higher-level PTs (L2PTs) that map the other groupings of L3PTs, beginning at

  Segment 1 bit field = 3FE
  Segment 2 bit field = 3FE
  Segment 3 bit field = 0
  Byte within page = 0,

  which result in

  VA = FFFFFFFD.FF000000
  (also known as L2_Base).
- Within that block of L2PTs, the 1,022d L2PT is actually the next-higher-level page table that maps the L2PTs, namely, the L1PT. The L1PT begins at

  Segment 1 bit field = 3FE
  Segment 2 bit field = 3FE
  Segment 3 bit field = 3FE
  Byte within page = 0,

  which result in

  VA = FFFFFFFD.FF7FC000
  (also known as L1_Base).
- Within that L1PT, the 1,022d PTE is the one used for self-mapping these page tables. The address of the self-mapper L1PTE is

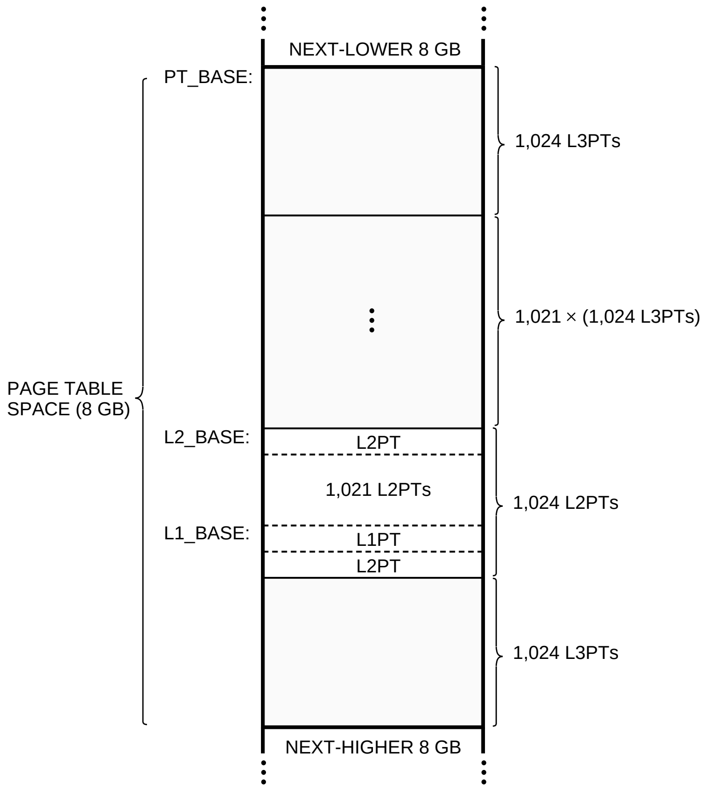


Figure 5
Page Table Space

Segment 1 bit field = 3FE
Segment 2 bit field = 3FE
Segment 3 bit field = 3FE
Byte within page = 3FE × 8

which result in

VA = FFFFFFFD.FF7FDFF0.

This positional correspondence within PT space is preserved should a different high-level PTE be chosen for self-mapping the page tables.

The properties inherent in this self-mapped page table are compelling.

- The amount of virtual memory reserved is exactly the amount required for mapping the page tables, regardless of page size or page table depth. Consider the segment-numbered bit fields of a given virtual address from Figure 2. Concatenated, these bit fields constitute the virtual page number (VPN) portion of a given virtual address.

  The total size of the PT space needed to map every VPN is the number of possible VPNs times 8 bytes, the size of a PTE. The total size of the address space mapped by that PT space is the number of possible VPNs times the page size. Factoring out the VPN multiplier, the difference between these is the page size divided by 8, which is exactly the size of the Segment 1 bit field in the virtual address. Hence, all the space mapped by a single PTE at the highest level of page table is exactly the size required for mapping all the PTEs that could ever be needed to map the process' address space.
- The mapping of PT space involves simply choosing one of the highest-level PTEs and forcing it to self-map.
- No additional system tuning or coding is required to accommodate a more widely implemented virtual address width in PT space. By definition of the self-map effect, the exact amount of virtual address space required will be available, no more and no less.
- It is easy to locate a given PTE. The address of a PTE becomes an efficient function of the address that the PTE maps. The function first clears the byte-within-page bit field of the subject virtual address and then shifts the remaining virtual address bits such that the Segments 1, 2, and 3 bit field values (Figure 2) now reside in the corresponding next-lower bit field positions. The function then writes (and sign extends if necessary) the vacated Segment 1 field with the index of the self-mapper PTE. The result is the address of the PTE that maps the original virtual address. Note that this algorithm also works for addresses within PT space, including that of the self-mapper PTE itself.
- Process page table residency in virtual memory is achieved without imposing on the capacity of shared space. That is, there is no longer a need to map the process page tables into shared space. Such a mapping would be redundant and wasteful.

OpenVMS 64-bit Virtual Address Space

With this page table residency strategy in hand, it became possible to finalize a 64-bit virtual address layout for the OpenVMS operating system. A self-mapper PTE had to be chosen. Consider again the highest level of page table in a given process' page table structure (Figure 4). The first PTE in that page table maps a section of virtual memory that includes P0 and P1 spaces. This PTE was therefore unavailable for use as a self-mapper. The last PTE in that page table maps a section of virtual memory that includes S0/S1 space. This PTE was also unavailable for self-mapping purposes.

All the intervening high-level PTEs were potential choices for self-mapping the page tables. To maximize the size of process-private space, the correct choice is the next-lower PTE than the one that maps the lowest address in shared space.

This choice is implemented as a boot-time algorithm. Bootstrap code first determines the size required for OpenVMS shared space, calculating the corresponding number of high-level PTEs. A sufficient number of PTEs to map that shared space are allocated later from the high-order end of a given process' highest-level page table page. Then the next-lower PTE is allocated for self-mapping that process' page tables. All remaining lower-ordered PTEs are left available for mapping process-private space. In practice, nearly all the PTEs are available, which means that on today's systems, almost 8 TB of process-private virtual memory is available to a given OpenVMS process.

Figure 6 presents the final 64-bit OpenVMS virtual address space layout. The portion with the lower addresses is entirely process-private. The higher-addressed portion is shared by all process address spaces. PT space is a region of virtual memory that lies between the P2 and S2 spaces for any given process and at the same virtual address for all processes.

Note that PT space itself consists of a process-private and a shared portion. Again, consider Figure 5. The highest-level page table page, L1PT, is process-private. It is pointed to by the PTBR. (When a process' context is loaded, or made active, the process' PTBR value is loaded from the process' hardware-privileged context block into the PTBR register, thereby making current the page table structure pointed to by that PTBR and the process-private address space that it maps.)

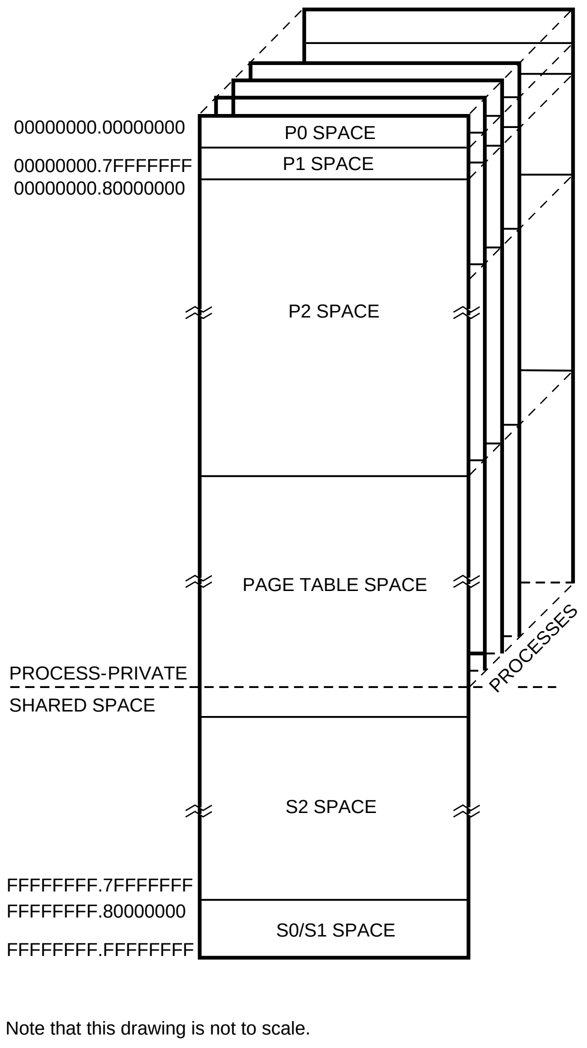


Figure 6
OpenVMS Alpha 64-bit Virtual Address Space

All higher-addressed page tables in PT space are used to map shared space and are themselves shared. They are also adjacent to the shared space that they map. All page tables in PT space that reside at addresses lower than that of the L1PT are used to map process-private space. These page tables are process-private and are adjacent to the process-private space that they map. Hence, the end of the L1PT marks a universal boundary between the process-private portion and the shared portion of the entire virtual address space, serving to separate even the PTEs that map those portions. In Figure 6, the line passing through PT space illustrates this boundary.

A direct consequence of this design is that the process page tables have been privatized. That is, the portion of PT space that is process-private is currently active in virtual memory only when the owning process itself is currently active on the processor. Fortunately, the majority of page table references occur while executing in the context of the owning process. Such references actually are enhanced by the privatization of the process page tables because the mapping function of a virtual address to its PTE is now more efficient.

Privatization does raise a hurdle for certain privileged code that previously could access a process' page tables when executing outside the context of the owning process. With the page tables resident in shared space, such references could be made regardless of which process is currently active. With privatized page tables, additional access support is needed, as presented in the next section.

A final commentary is warranted for the separately maintained system page table. The self-mapped page table approach to supplying page table residency in virtual memory includes the PTEs for any virtual addresses, whether they are process-private or shared. The shared portion of PT space could serve now as the sole location for shared-space PTEs. Being redundant, the original SPT is eminently discardable; however, discarding the SPT would create a massive compatibility problem for device drivers with their many 32-bit SPT references. This area is one in which an opportunity exists to preserve a significant degree of privileged code compatibility.

Key Measures Taken to Maximize Privileged Code Compatibility

To implement 64-bit virtual address space support, we altered central sections of the OpenVMS Alpha kernel and many of its key data structures. We expected that such changes would require compensating or corresponding source changes in surrounding privileged components within the kernel, in device drivers, and in privileged layered products.

For example, the previous discussion seems to indicate that any privileged component that reads or writes PTEs would now need to use 64-bit-wide pointers instead of 32-bit pointers. Similarly, all system fork threads and interrupt service routines could no longer count on direct access to process-private PTEs without regard to which process happens to be current at the moment.

A number of factors exacerbated the impact of such changes. Since the OpenVMS Alpha operating system originated from the OpenVMS VAX operating system, significant portions of the OpenVMS Alpha operating system and its device drivers are still written in MACRO-32 code, a compiled language on the Alpha platform.[1] Because MACRO-32 is an assembly-level style of programming language, we could not simply change the definitions and declarations of various types and rely on recompilation to handle the move from 32-bit to 64-bit pointers. Finally, there are well over 3,000 references to PTEs from MACRO-32 code modules in the OpenVMS Alpha source pool.

We were thus faced with the prospect of visiting and potentially altering each of these 3,000 references. Moreover, we would need to follow the register lifetimes that resulted from each of these references to ensure that all address calculations and memory references were done using 64-bit operations. We expected that this process would be time-consuming and error prone and that it would have a significant negative impact on our completion date.

Once OpenVMS Alpha version 7.0 was available to users, those with device drivers and privileged code of their own would need to go through a similar effort. This would further delay wide use of the release. For all these reasons, we were well motivated to minimize the impact on privileged code. The next four sections discuss techniques that we used to overcome these obstacles.

Resolving the SPT Problem

A significant number of the PTE references in privileged code are to PTEs within the SPT. Device drivers often double-map the user's I/O buffer into S0/S1 space by allocating and appropriately initializing system page table entries (SPTEs). Another situation in which a driver manipulates SPTEs is in the substitution of a system buffer for a poorly aligned or noncontiguous user I/O buffer that prevents the buffer from being directly used with a particular device. Such code relies heavily on the system data cell MMG$GL_SPTBASE, which points to the SPT.

The new page table design completely obviates the need for a separate SPT. Given an 8-KB page size and 8 bytes per PTE, the entire 2-GB S0/S1 virtual address space range can be mapped by 2 MB of PTEs within PT space. Because S0/S1 resides at the highest addressable end of the 64-bit virtual address space, it is mapped by the highest 2 MB of PT space. The arcs on the left in Figure 7 illustrate this mapping. The PTEs in PT space that map S0/S1 are fully shared by all processes, but they must be referenced with 64-bit addresses.

This incompatibility is completely hidden by the creation of a 2-MB "SPT window" over the 2 MB in PT space (level 3 PTEs) that maps S0/S1 space. The SPT window is positioned at the highest addressable end of S0/S1 space. Therefore, an access through the SPT window only requires a 32-bit S0/S1 address and can obtain any of the PTEs in PT space that map S0/S1 space. The arcs on the right in Figure 7 illustrate this access path.

The SPT window is set up at system initialization time and consumes only the 2 KB of PTEs that are needed to map 2 MB. The system data cell MMG$GL_SPTBASE now points to the base of the SPT window, and all existing references to that data cell continue to function correctly without change.[7]

Providing Cross-process PTE Access for Direct I/O

The self-mapping of the page tables is an elegant solution to the page table residency problem imposed by the preceding design. However, the self-mapped page tables present significant challenges of their own to the I/O subsystem and to many device drivers.

Typically, OpenVMS device drivers for mass storage, network, and other high-performance devices perform direct memory access (DMA) and what OpenVMS calls "direct I/O." These device drivers lock down into physical memory the virtual pages that contain the requester's I/O buffer. The I/O transfer is performed directly to those pages, after which the buffer pages are unlocked, hence the term "direct I/O."

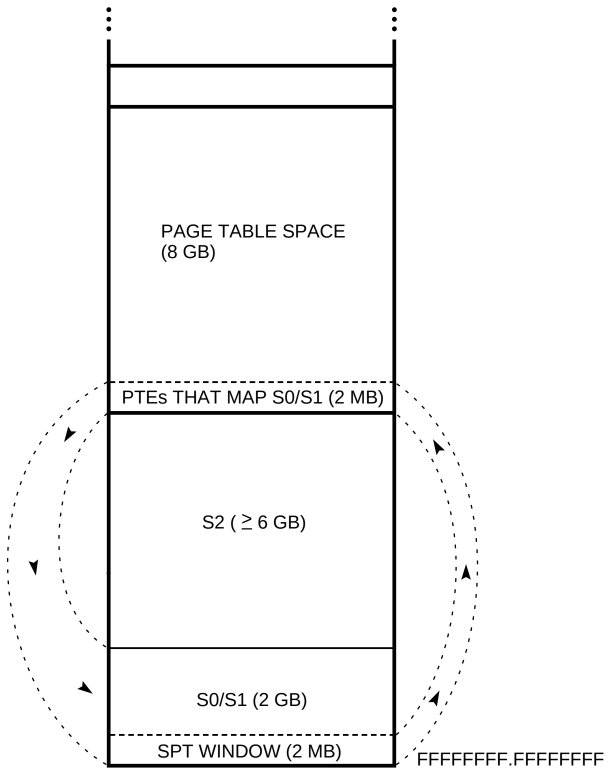


Figure 7
System Page Table Window

The virtual address of the buffer is not adequate for device drivers because much of the driver code runs in system context and not in the process context of the requester. Similarly, a process-specific virtual address is meaningless to most DMA devices, which typically can deal only with the physical addresses of the virtual pages spanned by the buffer.

For these reasons, when the I/O buffer is locked into memory, the OpenVMS I/O subsystem converts the virtual address of the requester's buffer into (1) the address of the PTE that maps the start of the buffer and (2) the byte offset within that page to the first byte of the buffer.

Once the virtual address of the I/O buffer is converted to a PTE address, all references to that buffer are made using the PTE address. This remains the case even if this I/O request and I/O buffer are handed off from one driver to another. For example, the I/O request may be passed from the shadowing virtual disk driver to the small computer systems interface (SCSI) disk class driver to a port driver for a specific SCSI host adapter. Each of these drivers will rely solely on the PTE address and the byte offset and not on the virtual address of the I/O buffer.

Therefore, the number of virtual address bits the requester originally used to specify the address of the I/O buffer is irrelevant. What really matters is the number of address bits that the driver must use to reference a PTE.

These PTE addresses were always within the page tables within the balance set slots in shared S0/S1 space. With the introduction of the self-mapped page tables, a 64-bit address is required for accessing any PTE in PT space. Furthermore, the desired PTE is not accessible using this 64-bit address when the driver is no longer executing in the context of the original requester process. This is called a cross-process PTE access problem.

In most cases, this access problem is solved for direct I/O by copying the PTEs that map the I/O buffer when the I/O buffer is locked into physical memory. The PTEs in PT space are accessible at that point because the requester process context is required in order to lock the buffer. The PTEs are copied into the kernel's heap storage and the 64-bit PT space address is replaced by the address of the PTE copies. Because the kernel's heap storage remains in S0/S1 space, the replacement address is a 32-bit address that is shared by all processes on the system.

This copy approach works because drivers do not need to modify the actual PTEs. Typically, this arrangement works well because the associated PTEs can fit into dedicated space within the I/O request packet data structure used by the OpenVMS operating system, and there is no measurable increase in CPU overhead to copy those PTEs.

If the I/O buffer is so large that its associated PTEs cannot fit within the I/O request packet, a separate kernel heap storage packet is allocated to hold the PTEs. If the I/O buffer is so large that the cost of copying all the PTEs is noticeable, a direct access path is created as follows:

- The L3PTEs that map the I/O buffer are locked into physical memory.
- Address space within S0/S1 space is allocated and mapped over the L3PTEs that were just locked down.

This establishes a 32-bit addressable shared-space window over the L3PTEs that map the I/O buffer.

The essential point is that one of these methods is selected and employed until the I/O is completed and the buffer is unlocked. Each method provides a 32-bit PTE address that the rest of the I/O subsystem can use transparently, as it has been accustomed to doing, without requiring numerous, complex source changes.

Use of Self-identifying Structures

To accommodate 64-bit user virtual addresses, a number of kernel data structures had to be expanded and changed. For example, asynchronous system trap (AST) control blocks, buffered I/O packets, and timer queue entries all contain various user-provided addresses and parameters that can now be 64-bit addresses. These structures are often embedded in other structures such that a change in one has a ripple effect to a set of other structures.

If these structures changed unconditionally, many scattered source changes would have been required. Yet, at the same time, each of these structures had consumers who had no immediate need for the 64-bit addressing–related capabilities.

Instead of simply changing each of these structures, we defined a new 64-bit-capable variant that can coexist with its traditional 32-bit counterpart. The 64-bit variant's structures are "self-identifying" because they can readily be distinguished from their 32-bit counterparts by examining a particular field within the structure itself. Typically, the 32-bit and 64-bit variants can be intermixed freely within queues and only a limited set of routines need to be aware of the variant types.

Thus, for example, components that do not need 64-bit ASTs can continue to build 32-bit AST control blocks and queue them with the SCH$QAST routine. Similarly, 64-bit AST control blocks can be queued with the same SCH$QAST routine because the AST delivery code was enhanced to support either type of AST control block.

The use of self-identifying structures is also a technique that was employed to compatibly enhance public user-mode interfaces to library routines and the OpenVMS kernel. This topic is discussed in greater detail in "The OpenVMS Mixed Pointer Size Environment."[8]

Limiting the Scope of Kernel Changes

Another key tactic that allowed us to minimize the required source code changes to the OpenVMS kernel came from the realization that full support of 64-bit virtual addressing for all processes does not imply or require exclusive use of 64-bit pointers within the kernel. The portions of the kernel that handled user addresses would for the most part need to handle 64-bit addresses; however, most kernel data structures could remain within the 32-bit addressable S0/S1 space without any limit on user functionality. For example, the kernel heap storage is still located in S0/S1 space and continues to be 32-bit addressable. The Record Management Services (RMS) supports data transfers to and from 64-bit addressable user buffers, but RMS continues to use 32-bit-wide pointers for its internal control structures. We therefore focused our effort on the parts of the kernel that could benefit from internal use of 64-bit addresses (see the section Immediate Use of 64-bit Addressing by the OpenVMS Kernel for examples) and that needed to change to support 64-bit user virtual addresses.

Privileged Code Example—The Swapper

The OpenVMS working set swapper provides an interesting example of how the 64-bit changes within the kernel may impact privileged code.

Only a subset of a process' virtual pages is mapped to physical memory at any given point in time. The OpenVMS operating system occasionally swaps this working set of pages out of memory to secondary storage as a consequence of managing the pool of available physical memory. The entity responsible for this activity is a privileged process called the working set swapper or swapper, for short. Since it is responsible for transferring the working set of a process into and out of memory when necessary, the swapper must have intimate knowledge of the virtual address space of a process including that process' page tables.

Consider the earlier discussion in the section OpenVMS 64-bit Virtual Address Space about how the process' page tables have been privatized as a way to efficiently provide page table residency in virtual memory. A consequence of this design is that while the swapper process is active, the page tables of the process being swapped are not available in virtual memory. Yet, the swapper requires access to those page tables to do its job. This is an instance of the cross-process PTE access problem mentioned earlier.

The swapper is unable to directly access the page tables of the process being swapped because the swapper's own page tables are currently active in virtual memory. We solved this access problem by revising the swapper to temporarily "adopt" the page tables of the process being swapped. The swapper accomplishes this by temporarily changing its PTBR contents to point to the page table structure for the process being swapped instead of to the swapper's own page table structure. This change forces the PT space of the process being swapped to become active in virtual memory and therefore available to the swapper as it prepares the process to be swapped. Note that the swapper can make this temporary change because the swapper resides in shared space. The swapper does not vanish from virtual memory as the PTBR value is changed. Once the process has been prepared for swapping, the swapper restores its own PTBR value, thus relinquishing access to the target process' PT space contents.

Thus, it can be seen how privileged code with intimate knowledge of OpenVMS memory management mechanisms can be affected by the changes to support 64-bit virtual memory. Also evident is that the alterations needed to accommodate the 64-bit changes are relatively straightforward. Although the swapper has a higher-than-normal awareness of memory management internal workings, extending the swapper to accommodate the 64-bit changes was not particularly difficult.

Immediate Use of 64-bit Addressing by the OpenVMS Kernel

Page table residency was certainly the most pressing issue we faced with regard to the OpenVMS kernel as it evolved from a 32-bit to a 64-bit-capable operating system. Once implemented, 64-bit virtual addressing could be harnessed as an enabling technology for solving a number of other problems as well. This section briefly discusses some prominent examples that serve to illustrate how immediately useful 64-bit addressing became to the OpenVMS kernel.

Page Frame Number Database and Very Large Memory

The OpenVMS Alpha operating system maintains a database for managing individual, physical page frames of memory, i.e., page frame numbers. This database is stored in S0/S1 space. The size of this database grows linearly as the size of the physical memory grows.

Future Alpha systems may include larger memory configurations as memory technology continues to evolve. The corresponding growth of the page frame number database for such systems could consume an unacceptably large portion of S0/S1 space, which has a maximum size of 2 GB. This design effectively restricts the maximum amount of physical memory that the OpenVMS operating system would be able to support in the future.

We chose to remove this potential restriction by relocating the page frame number database from S0/S1 to 64-bit addressable S2 space. There it can grow to support any physical memory size being considered for years to come.

Global Page Table

The OpenVMS operating system maintains a data structure in S0/S1 space called the global page table (GPT). This pseudo–page table maps memory objects called global sections. Multiple processes may map portions of their respective process-private address spaces to these global sections to achieve protected shared memory access for whatever applications they may be running.

With the advent of P2 space, one can easily anticipate a need for orders-of-magnitude-greater global section usage. This usage directly increases the size of the GPT, potentially reaching the point where the GPT consumes an unacceptably large portion of S0/S1 space. We chose to forestall this problem by relocating the GPT from S0/S1 to S2 space. This move allows the configuration of a GPT that is much larger than any that could ever be configured in S0/S1 space.

Summary

Although providing 64-bit support was a significant amount of work, the design of the OpenVMS operating system was readily scalable such that it could be achieved practically. First, we established a goal of strict binary compatibility for nonprivileged applications. We then designed a superset virtual address space that extended both process-private and shared spaces while preserving the 32-bit visible address space to ensure compatibility. To maximize the available space for process-private use, we chose an asymmetric style of address space layout. We privatized the process page tables, thereby eliminating their residency in shared space. The few page table accesses that occurred from outside the context of the owning process, which no longer worked after the privatization of the page tables, were addressed in various ways. A variety of ripple effects stemming from this design were readily solved within the kernel.

Solutions to other scaling problems related to the kernel were immediately possible with the advent of 64-bit virtual address space. Already mentioned was the complete removal of the process page tables from shared space. We also removed the global page table and the page frame number database from 32-bit addressable to 64-bit addressable shared space. The immediate net effect of these changes was significantly more room in S0/S1 space for configuring more kernel heap storage, more balance slots to be assigned to greater numbers of memory resident processes, etc. We further anticipate use of 64-bit addressable shared space to realize additional benefits of VLM, such as for caching massive amounts of file system data.

Providing 64-bit addressing capacity was a logical, evolutionary step for the OpenVMS operating system. Growing numbers of customers are demanding the additional virtual memory to help solve their problems in new ways and to achieve higher performance. This has been especially fruitful for database applications, with substantial performance improvements already proved possible by the use of 64-bit addressing on the Digital UNIX operating system. Similar results are expected on the OpenVMS system. With terabytes of virtual memory and many gigabytes of physical memory available, entire databases may be loaded into memory at once. Much of the I/O that otherwise would be necessary to access the database can be eliminated, thus allowing an application to improve performance by orders of magnitude, for example, to reduce query time from eight hours to five minutes. Such performance gains were difficult to achieve while the OpenVMS operating system was constrained to a 32-bit environment. With the advent of 64-bit addressing, OpenVMS users now have a powerful enabling technology available to solve their problems.

Acknowledgments

The work described in this paper was done by members of the OpenVMS Alpha Operating System Development group. Numerous contributors put in many long hours to ensure a well-considered design and a high-quality implementation. The authors particularly wish to acknowledge the following major contributors to this effort: Tom Benson, Richard Bishop, Walter Blaschuk, Nitin Karkhanis, Andy Kuehnel, Karen Noel, Phil Norwich, Margie Sherlock, Dave Wall, and Elinor Woods. Thanks also to members of the Alpha languages community who provided extended programming support for a 64-bit environment; to Wayne Cardoza, who helped shape the earliest notions of what could be accomplished; to Beverly Schultz, who provided strong, early encouragement for pursuing this project; and to Ron Higgins and Steve Noyes, for their spirited and unflagging support to the very end.

The following reviewers also deserve thanks for the invaluable comments they provided in helping to prepare this paper: Tom Benson, Cathy Foley, Clair Grant, Russ Green, Mark Howell, Karen Noel, Margie Sherlock, and Rod Widdowson.

References and Notes

1. N. Kronenberg, T. Benson, W. Cardoza, R. Jagannathan, and B. Thomas, "Porting OpenVMS from VAX to Alpha AXP," *Digital Technical Journal,* vol. 4, no. 4 (1992): 111–120.

2. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford, Mass.: Digital Press, 1987).

3. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual,* 2d ed. (Newton, Mass.: Digital Press, 1995).

4. Although an OpenVMS process may refer to P0 or P1 space using either 32-bit or 64-bit pointers, references to P2 space require 64-bit pointers. Applications may very well execute with mixed pointer sizes. (See reference 8 and D. Smith, "Adding 64-bit Pointer Support to a 32-bit Run-time Library," *Digital Technical Journal,* vol. 8, no. 2 [1996, this issue]: 83–95.) There is no notion of an application executing in either a 32-bit mode or a 64-bit mode.

5. Superset system services and language support were added to facilitate the manipulation of 64-bit addressable P2 space.[8]

6. This mechanism has been in place since OpenVMS Alpha version 1.0 to support virtual PTE fetches by the translation buffer miss handler in PALcode. (PALcode is the operating system–specific privileged architecture library that provides control over interrupts, exceptions, context switching, etc.[3]) In effect, this means that the OpenVMS page tables already existed in two virtual locations, namely, S0/S1 space and PT space.

7. The SPT window is more precisely only an S0/S1 PTE window. The PTEs that map S2 space are referenced using 64-bit pointers to their natural locations in PT space and are not accessible through the use of this SPT window. However, because S2 PTEs did not exist prior to the introduction of S2 space, this limitation is of no consequence to contexts that are otherwise restricted to S0/S1 space.

8. T. Benson, K. Noel, and R. Peterson, "The OpenVMS Mixed Pointer Size Environment," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 72–82.

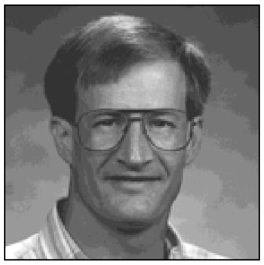
General References

R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures, Version 1.5*(Newton, Mass.: Digital Press, 1994).

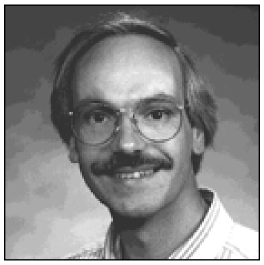
OpenVMS Alpha Guide to 64-Bit Addressing (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBCA-TE, December 1995).

OpenVMS Alpha Guide to Upgrading Privileged-Code Applications (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QSBGA-TE, December 1995).

Biographies



Michael S. Harvey
Michael Harvey joined Digital in 1978 after receiving his B.S.C.S. from the University of Vermont. In 1984, as a member of the OpenVMS Engineering group, he participated in new processor support for VAX multiprocessor systems and helped develop OpenVMS symmetric multiprocessing (SMP) support for these systems. He received a patent for this work. Mike was an original member of the RISCy-VAX task force, which conceived and developed the Alpha architecture. Mike led the project that ported the OpenVMS Executive from the VAX to the Alpha platform and subsequently led the project that designed and implemented 64-bit virtual addressing support in OpenVMS. This effort led to a number of patent applications. As a consulting software engineer, Mike is currently working in the area of infrastructure that supports the Windows NT/OpenVMS Affinity initiative.



Leonard S. Szubowicz
Leonard Szubowicz is a consulting software engineer in Digital's OpenVMS Engineering group. Currently the technical leader for the OpenVMS I/O engineering team, he joined Digital Software Services in 1983. As a member of the OpenVMS 64-bit virtual addressing project team, Lenny had primary responsibility for I/O and driver support. Prior to that, he was the architect and project leader for the OpenVMS high-level language device driver project, contributed to the port of the OpenVMS operating system to the Alpha platform, and was project leader for RMS Journaling. Lenny is a coauthor of *Writing OpenVMS Alpha Device Drivers in C,* which was recently published by Digital Press.