OpenVMS Integrity Server VHPT Sizing for Large Applications Performance

Bruce Ellis
BRUDEN-OSSG

Overview

This paper describes the Virtual Hash Page Table (VHPT) on OpenVMS integrity Servers and how certain types of applications may yield a 5% to 9% performance gain in CPU time by adjusting the size of the VHPT.

The paper idea started long ago when I sat in a session given by Andy Kuehnel, of OpenVMS Engineering, discussing memory management changes for Integrity Server systems. In the session there was a brief discussion of the VHPT and a note that it was sized by the system parameter VHPT_SIZE. The parameter has a default setting of 1. With this setting, the VHPT is sized as the value that "OpenVMS considers optimal". That value is fixed at 32K bytes. A value of 0 disables the VHPT. A value larger than 1 is rounded to the next power of 2.

It seemed that there might be cases where it might be changed for tuning purposes. At the time, I just let it go.

For the purposes of this paper, we will assume that the reader has a fundamental understanding of the translation buffer (TLB), and how it acts as an address translation cache. For more background on the TLB for both Alpha and Integrity Server systems, please refer to the ***Intel® Itanium® Architecture Software Developer's Manual Volume 2: System Architecture***.

When the CPU attempts to translate a virtual address, it uses a hash to look in the TLB, which is made up of on-CPU (or on-Core) memory. If the translation information is found in the TLB and the address is associated with the current process or kernel thread, it is completed and the CPU continues execution. TLBs exist for instructions (ITLB) and data (DTLB).

If the translation is not found in the cache, assuming there is no VHPT (which will happen if VHPT_SIZE is set to 0), an interruption is generated and control is passed to a small piece of code in a system table called the Interruption Vector Table (IVT). The IVT code will use normal three-level address translation to look up the page table entry for the page. If the page is valid (in memory and owned by the process), the location of the page, or its Page Frame Number (PFN), is mapped into the TLB and the miss is resolved. If while examining the page table entry it is determined that the page is not valid, a page fault is generated. If the page table entry identifies that the executing code does not have read, write, or execute permission, an access violation is generated.

In the lookup process of locating the page table entry, three memory fetches can be generated. This will slow down the CPU performance of the application. On a well behaved, relatively small application, most address translations are completed using the TLB, minimizing this overhead.

Integrity Server systems provide an additional level of translation assist that works between the TLB and physical address translation. This mechanism is through the VHPT. The VHPT is a linear array of 32-byte entries (although IA64 supports 8-byte entries), similar in nature to the TLB. The VHPT entry contains a tag that uniquely identifies a virtual address and a region identifier. The region identifier uniquely associates the address with a specific process or kernel thread. If the tag generated on the lookup matches the tag value in the VHPT, there is a match. The VHPT entries also contain the PFN of the location of the page in physical memory. Additional contents of a VHPT entry include a tag invalid bit and page protection information, among other fields. If the tag invalid bit is set, the entry is stale and will not be used in the translation.

The VHPT is allocated from physically contiguous memory at boot time. It is mapped through a "pinned" translation that cannot be displaced from the TLB.

The VHPT is accessed through a component of the CPU called the VHPT walker. So, no software intervention is required to access the VHPT. Although, there is support for operating systems to directly access the VHPT and maintain collision lists on common tags. OpenVMS does not currently use this feature.

In summary, the VHPT provides an additional, relatively high performance mechanism to resolve TLB misses without software intervention. It is slower than using the TLB, but faster than performing a three-level address translation.

Sizing the VHPT

One would assume that the VHPT would, in general, benefit performance. In some cases a larger VHPT should improve performance. On well behaved applications, that are relatively small, the default setting for the VHPT is more than sufficient. As mentioned earlier, the default size of the VHPT is 32K bytes. This size allows for mapping 1024 addresses. With an 8192-byte page size, an application that is 8MB in size, or smaller, will not exceed this space. You should see significant benefits with the default setting.

The notion of a "well behaved" application is not intended to be a qualitative viewpoint of how the program was written. What we mean by "well behaved" is that the application tends to touch pages in close proximity in memory. An example of this type of application might be a program that accesses a large array iteratively from the lowest index to the highest. If you had a large array of longwords and accessed it in this fashion, you would touch the first longword in a page and likely get a TLB miss. From that point on, you would get 2047 hits (with the default 8192-byte page). This would be followed by a miss and 2047 more hits. In this type of application, the size of the VHPT would not matter, unless you accessed

The reality is that most applications use large arrays specifically so that they can access them randomly, leading to poorer locality. Additionally, large applications may organize data in trees that may also have poorer locality. There are design strategies that can lead to better locality, but most large applications will have some degree of poor locality.

Problems with the default size come into play when you have a large application with poor temporal locality. Poor temporal locality means that the application tends to frequently touch a memory location then, touch another that is not in close proximity. In this scenario, many virtual addresses are touched over a short period of time, causing the VHPT to fill and increase the likelihood of tag collisions. When a tag collision (an address hashes to a tag that is already in use) occurs, we loose the old VHPT entry and it is replaced with the new translation. This behavior minimizes the effectiveness of the VHPT and begs for tuning the size of the VHPT.

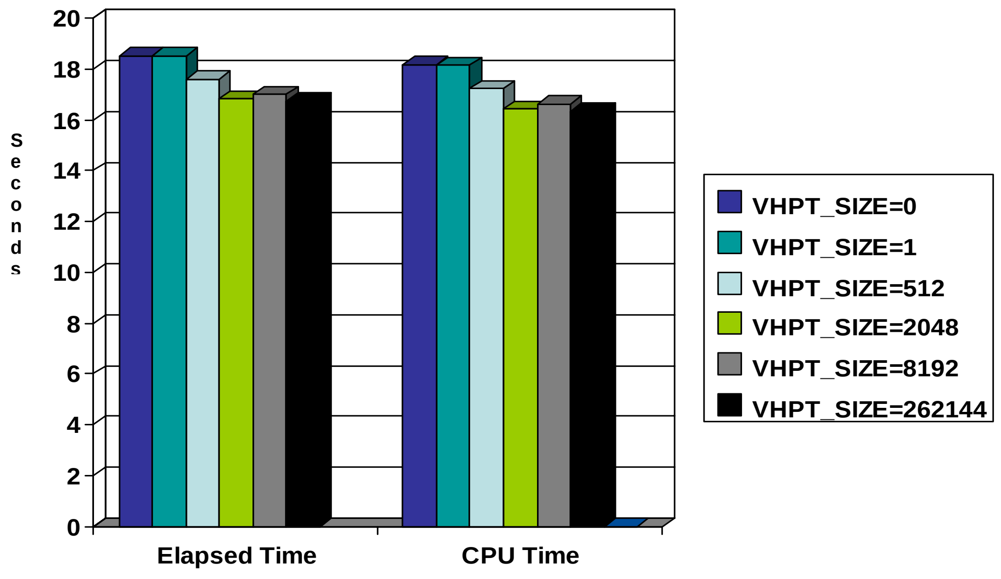
Large scientific applications, database applications, among others are large programs that frequently operate with poor locality.

To test sizing the VHPT, I have a program that I wrote many years ago that computes frequency and connectivity of lottery numbers, based on history. This factor is calculated for all possible combinations of the Colorado Lottery (5 million combinations). After computing the factor, an array of pointers to each combination is generated and the pointers are sorted from highest factor to lowest. The program has not increased the chances of picking a winning set of numbers, which is why I write this article. It does have the characteristic that it touches a lot of memory with very poor temporal locality. I have used the program to test memory interleaving performance, the performance of hyperthreads, as well as in these tests. Interestingly, the benefits achieved through the use of hyperthreading were mirrored in Oracle applications.

In my initial tests, I ran the program with a hodgepodge of settings to try to find where the sweet spot in performance would be achieved. I later ran tests with another size and also tested performance with the VHPT disabled. The tests reported are single runs, although I have repeated runs to show that the numbers stay close. The tests were run on a rx2600 with 2 CPUs (1.4 GHz). The results are reported in Table 1.

VHPT_SIZE (cost per CPU)	**Delta VMS Memory Cost in Pages From Default (w/ 2 CPUs)**	**Elapsed Time**	**Elapsed CPU**	**Page Faults**
0 (Disabled)	+32KB	**18.53**	**18.17**	30936
1 (32K)	**0**	**18.51**	**18.15**	30899
512K	120	17.61	17.23	30894
2048K	**384**	**16.81**	**16.42**	30786
8192K	2041	17.00	16.61	30932
262144K	65591	16.74	16.34	30894

Table 1: Single Thread Performance with Varying VHPT Sizes



Single Thread Performance with Varying VHPT Sizes

The most interesting results may be that disabling the VHPT had minimal performance degradation (less than .01%). This is likely due to the fact that the poor locality caused a lot of displacement of the default 1024 entries.

The setting of 2048 seemed to be the sweet spot setting for this application. Note, CPU time and elapsed time were reduced close to 9%. This setting allowed for 128896 VHPT entries. This number of VHPT entries was likely equal to or greater than the size required to map the entire virtual address space of the application.

Over-committing the size costs additional memory and does not yield significant improvement in performance. In fact, the 8192K size showed slight degradation. This behavior can be due to the hash algorithm itself maintaining VHPT entries that due not provide significantly more hits on translation.

This begs the question, what is the cost of over-sizing the VHPT? The larger the VHPT, the more memory is taken away from the system. On modern systems, 384 pages is a drop in the bucket. This is not the case when the VHPT eats up 65591 pages. Additionally, in rare cases where data is allocated, touched once or twice and never again, the VHPT support is wasted. Again, when an image runs down, VHPT entries must be invalidated, which incurs a slight CPU cost.

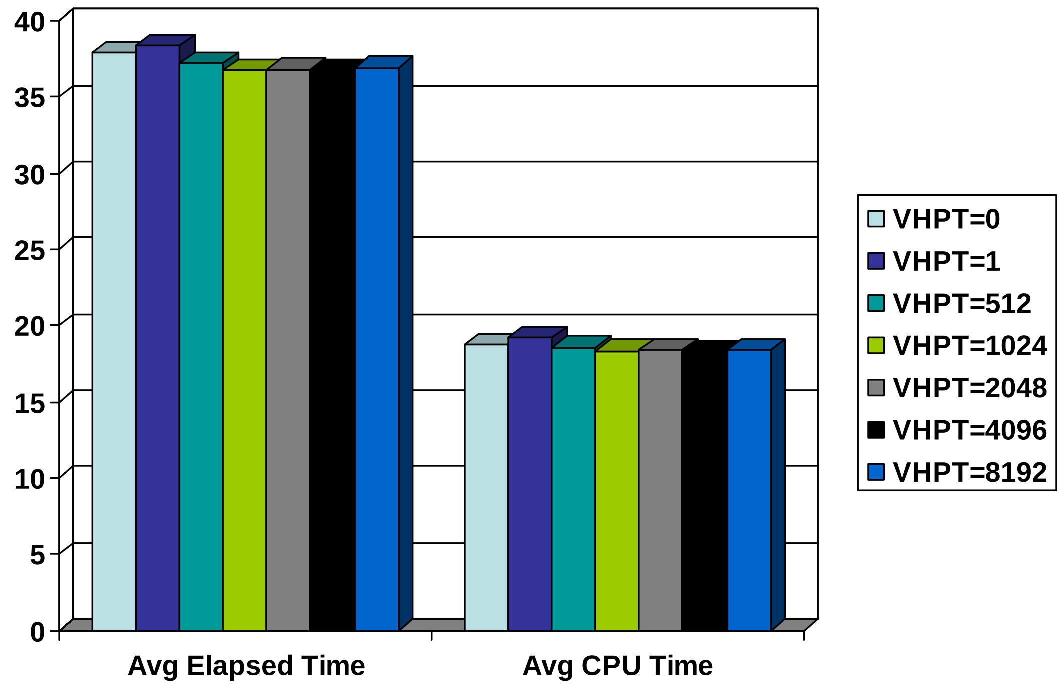
So, how do you determine the appropriate VHPT size for large applications? One method is trial and error or experimenting with different sizes. A little more scientific method is to determine the peak virtual address space size for a typical large application. If you size the VHPT large enough to hold the maximum number of page table entries, you should always have entries to resolve the corresponding TLB misses. The size can be calculated using (F$GETJPI("","VIRTPEAK")*16 (pagelets to pages))*32 (VHPT entry size)/1024 rounded up to the next power of two.

Multiple Process Sizing Tests

It is rare that you own a system that will run a single application. Tests were performed to calculate the performance of 4 programs running concurrently on a single 2 CPU rx2600. In this case, the multiple processes would trade off the use of the CPUs. These results are listed in table 2.

VHPT_ SIZE	JOB 1 Elapsed	JOB 2 Elapsed	JOB 3 Elapsed	JOB 4 Elapsed	Avg Elapsed	JOB 1 CPU	JOB 2 CPU	JOB 3 CPU	JOB 4 CPU	Avg CPU
0 (off)	37.82	37.88	38.18	37.63	**37.88**	18.94	19.01	18.94	19.00	**18.75**
1 (32K)	38.31	38.46	38.32	38.44	**38.35**	19.11	19.22	19.13	19.24	**19.18**
512	37.10	37.25	37.22	37.30	**37.22**	18.46	18.57	18.70	18.54	**18.57**
1024	36.85	36.85	36.63	36.69	**36.76**	18.36	18.30	18.22	18.35	**18.31**
2048	36.89	36.60	36.75	37.03	**36.82**	18.29	18.31	18.33	18.67	**18.40**
4096	36.95	36.78	36.74	36.41	**36.72**	18.22	18.34	18.36	18.22	**18.29**
8192	36.97	36.82	36.63	37.16	**36.90**	18.26	18.46	18.41	18.47	**18.40**

Table 2: 4 Computable Threads with 2 CPUs



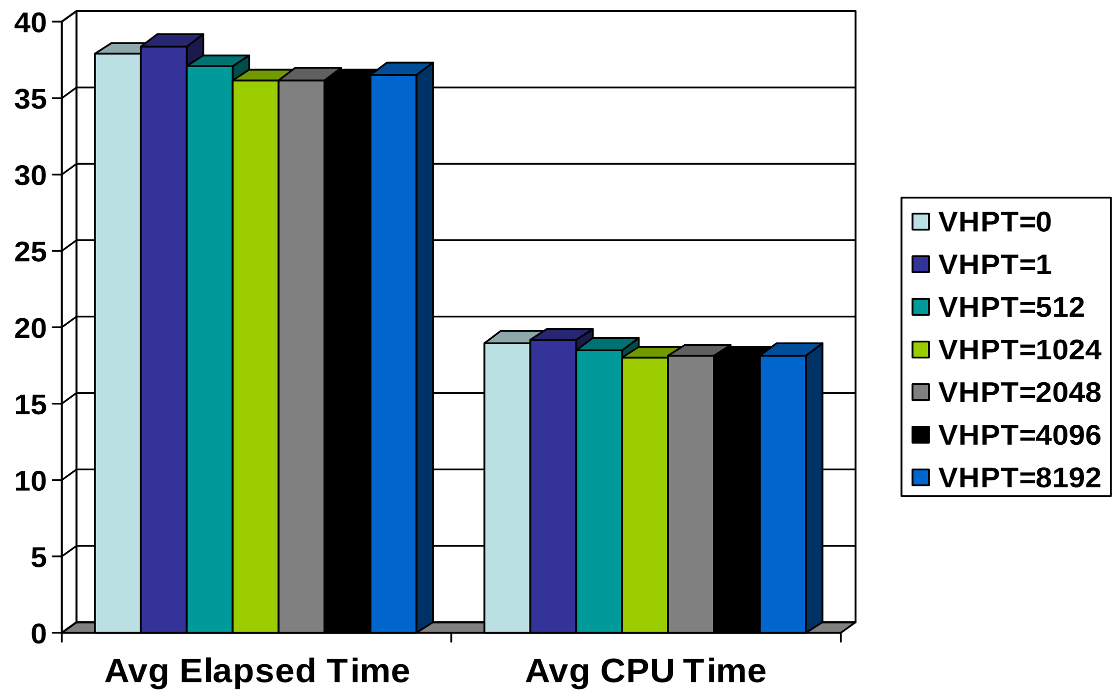
Computable Threads with 2 CPUs

Table 2 shows that almost any value for VHPT_SIZE is better than the default setting. In the case of disabling the VHPT, you are eliminating an extra step to look in the VHPT for a translation that you usually do not find. In the best case scenarios, you are seeing about a 4.5% improvement in performance. The competing processes are sharing the VHPT and losing some of the benefits of almost exclusive access to the cache. The setting of 4096 yields only slightly better (not statistically significant) improvement over 1024. The setting of 1024 costs 4 times less memory than 4096, and would probably be the way to go.

OpenVMS tries to keep processes and kernel threads scheduled on the same CPU. It is not always effective. Processes can drift from CPU to CPU over time. It is probably at least worth a test, where the processes are locked into the same CPU, to prevent restarting the build of the VHPT on a new CPU. We implement the next set of tests by setting affinity for Jobs 1 and 2 to CPU 0. Jobs 3 and 4 have affinity set to CPU 1. This method is not always practical, but is at least worth testing. Table 3 shows the results.

VHPT_ SIZE	JOB 1 Elapsed	JOB 2 Elapsed	JOB 3 Elapsed	JOB 4 Elapsed	Avg Elapsed	JOB 1 CPU	JOB 2 CPU	JOB 3 CPU	JOB 4 CPU	Avg CPU
0 (off)	37.91	38.38	37.77	37.82	**37.95**	19.04	19.28	18.87	18.89	**19.02**
1 (32K)	38.89	38.79	38.08	38.14	**38.48**	19.46	19.22	19.01	19.08	**19.19**
512	37.49	37.48	36.76	36.83	**37.14**	18.63	18.84	18.38	18.41	**18.57**
1024	36.51	36.40	35.88	35.93	**36.18**	18.18	18.23	17.94	17.80	**18.03**
2048	36.52	36.57	35.97	35.95	**36.25**	18.30	18.22	17.98	17.99	**18.12**
4096	36.34	36.38	35.85	35.89	**36.12**	18.05	18.26	17.92	17.89	**18.03**
8192	37.05	36.90	36.23	36.25	**36.61**	18.30	18.42	18.03	18.02	**18.19**

Table 3: 4 Computable Threads with 2 CPUs and Affinity Split Between CPUs



Computable Threads with 2 CPUs and Affinity Split Between CPUs

Table 3 shows that, when we can use affinity to force processes to the same CPU, we get much closer to 6% performance improvement. Again, these tests tend to be more of a proof of the theory that reusing the same CPUs provides the benefits of reusing the VHPT cache over the cost of rebuilding, than really providing a practical solution. You could implement this method using batch logins to alternatively select affinity from available CPUs. In a system with a large number of processes, this could lead to occasional CPU starvation.

To come up with a starting value for VHPT_SIZE, it is not practical to run applications and continually reset the VHPT_SIZE. As mentioned earlier, you can get a good starting value by viewing the Virtual Address Space Peak through F$GETJPI or the accounting record. The following display shows the virtual peak size from a batch accounting record. The virtual peak number is in pagelet units. If you multiply the number by 512 and divide by the page size on OpenVMS IA64 (8192 bytes), then multiply by 32 (the size of a VHPT entry), you will get the size in bytes of the VHPT that would cache all virtual addresses for a given application. Then divide by 1024, as the VHPT_SIZE in "K" units. You can then round up or down to the next power of 2 to get a starting value for VHPT_SIZE. In the example below, you would start at either 1024 or

2048 for VHPT_SIZE. The heuristic evidence shows that either of these values is very close to the sweet spot for the performance of the application.

```
   Accounting information:
   Buffered I/O count:                 59        Peak working set size:     497440
   Direct I/O count:                   95        Peak virtual size:         676624
   Page faults:                     31351        Mounted volumes:                0
   Charged CPU time:        0 00:00:19.04        Elapsed time:       0 00:00:37.91
$ write sys$output (676624*512)/8192*32
1353248
$
$ write sys$output (((676624*512)/8192*32)*2/2)/1024
1321
$
```

It should be noted that these calculations work best when using large applications with poor locality. A system that runs small applications or applications that have good temporal locality may require no change to VHPT_SIZE, whatsoever.

What about the Real World?

These tests were all performed using a single application. How does changing VHPT_SIZE affect other applications?

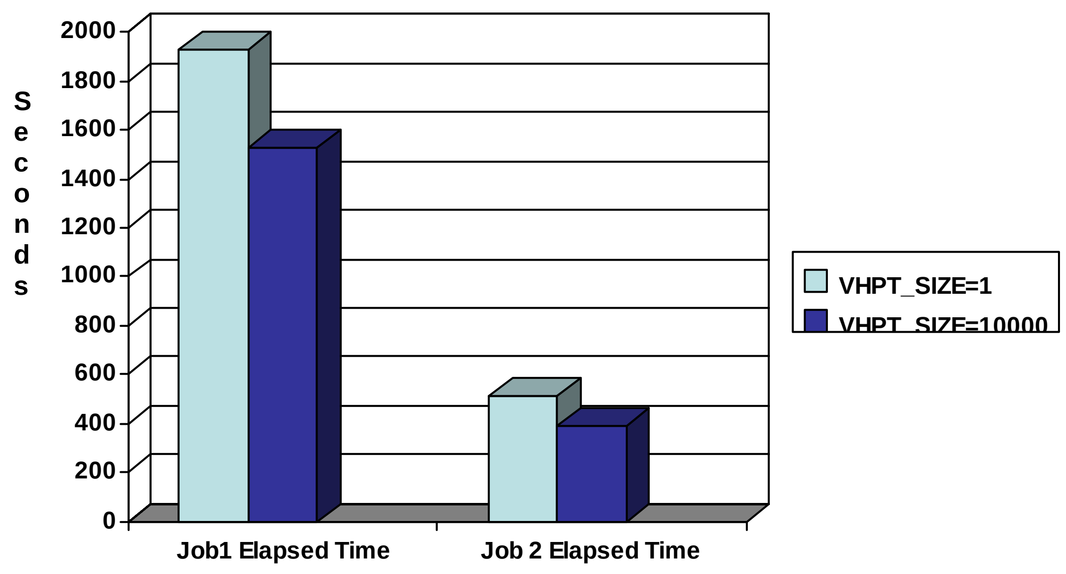
In a Java application that wrote 30,000 records to a file, run time went from 1:40.918 elapsed time to 1:35.042 by setting VHPT_SIZE from 1 to 2048, about a 6% improvement.

In an Oracle 10g test, elapsed time went from 405 seconds (VHPT_SIZE == 1) to 380 seconds (VHPT_SIZE set to 2048), again about a 6% improvement in elapsed time. The test database contained information about 50,000 customers, 200,000 customer orders and 200,000 ordered items. The test fetched data about 2,000 random customers and all of their associated orders/items. This was a read only test, all data was in the SGA (database cache) and no I/O was being performed. The test was CPU bound.

In another set of Oracle 10g tests from two different production applications, the results were more impressive. The results are shown in the table below. The setting of VHPT_SIZE at 10,000 would be rounded up to 16,384.

	VHPT_SIZE = 1	**VHPT_SIZE=10,000**
Job 1 Elapsed Time	00::32.29.00 (1929 seconds)	00:25:31.00 (1531 seconds)
Job 2 Elapsed Time	00:08:36.11 (516.11 seconds)	00:06:32.00 (392 seconds)

Table 4: Production Oracle 10g Tests



Oracle 10g Production Run Tests

Little scientific method was used to come up with these initial settings, yet there was a gain in each case. It should also be noted that increasing VHPT_SIZE will improve the CPU time portion of elapsed time, it will do nothing for I/O related activities, which factor into elapsed time.

Can I Monitor VHPT Performance?

Many performance metrics are provided by the OpenVMS executive, that make it easy to monitor activities like page fault rate, CPU time usage, I/O, etc. The VHPT operates at a level lower than OpenVMS, so the O/S does not keep counters. However, there is a Performance Monitoring Unit (PMU) on the Itanium CPU/Core that can be used to gather low level statistics.

OpenVMS provides a PRF extension to the System Dump Analyzer. It allows you to track hundreds of PMU statistics. The information is not necessarily intuitive, but it does provide counters on VHPT walks and hits.

You need to load the PRF execlet with a PRF LOAD command in SDA. You then start the monitor with PRF START MONITOR. You can view the PMU statistics using PRF SHOW MONITOR. There are many statistics that are displayed. The following sample shows a command procedure to extract statistics of relevance to the VHPT. The statistics are pretty raw, but note how the number of walks and hits go up in the second run with a larger VHPT_SIZE.

Sample Run with 4 Jobs Followed by 4 More (VHPT_SIZE ==1)

```
$ ana/sys

OpenVMS system analyzer

SDA> prf load
PRF$DEBUG load status = 00000001
SDA> prf start monitor
Event Monitoring started...
SDA>  Exit
$
$ type mon_vhpt.com
$ pipe write sys$output "prf show monitor" | ana/sys | search sys$input "VHPT","HPW"
$
$ @mon_vhpt
            VHPT_WALKER.ren                              0                    0.000%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                               2                    0.000%
$ @mon_vhpt
            VHPT_WALKER.ren                        9443219                    3.464%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                          132533                    0.135%
$ @mon_vhpt
            VHPT_WALKER.ren                       20238930                    4.271%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                          306863                    0.214%
$ @mon_vhpt
            VHPT_WALKER.ren                       33465222                    4.145%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                          487726                    0.193%
$ @mon_vhpt
            VHPT_WALKER.ren                       51768604                    4.470%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                          758175                    0.225%
$
```

Sample Run with 4 Jobs Followed by 4 More (VHPT_SIZE ==1024)

```
$ ana/sys

OpenVMS system analyzer

SDA> prf load
PRF$DEBUG load status = 00000001
SDA> prf start monitor
Event Monitoring started...
SDA>  Exit
$
$ @mon_vhpt
            VHPT_WALKER.ren                              0                    0.000%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                              166                    0.015%
$ @mon_vhpt
            VHPT_WALKER.ren                       23268887                    8.452%
        HPW_IDEMAND_HITS.ren                         0                    0.000%
        HPW_DHITS.calc                           379741                    0.407%
$ @mon_vhpt
            VHPT_WALKER.ren                       54159928                   10.834%
        HPW_IDEMAND_HITS.ren                        0                    0.000%
        HPW_DHITS.calc                          804501                    0.615%
$ @mon_vhpt
            VHPT_WALKER.ren                      102637892                   11.281%
        HPW_IDEMAND_HITS.ren                       13                    0.000%
        HPW_DHITS.calc                         1442057                    0.597%
$ @mon_vhpt
            VHPT_WALKER.ren                      128836279                   11.681%
        HPW_IDEMAND_HITS.ren                       13                    0.000%
        HPW_DHITS.calc                         1828584                    0.663%
$
```

Conclusion

If you have well behaved, relatively small applications (virtual address space is about 8 MB or less), you probably do not need to worry about sizing the VHPT_SIZE parameter. If you have larger applications, with poor locality (the two usually go hand-in-hand), consider increasing the VHPT_SIZE. Initial sizing should be based approximately on your largest peak virtual address size. The actual gains in performance are obviously application specific but may well be in the 4% to 9% range.

Acknowledgements

The author would like to offer his appreciation to Guy Peleg of Maklee Engineering for his input and Oracle 10g benchmarks and Norm Lastovica of Oracle Corporation for his insights and recommendations for improving the article.