

# Managing Persistent Address Space

## A Technical Whitepaper

By Lorinda Visnick



14 Oak Park  
Bedford, MA 01730 USA  
Phone: +1-781-280-4000  
[www.objectstore.com](http://www.objectstore.com)

## Introduction

This paper defines address space, describes how ObjectStore uses address space for persistent objects, discusses why an understanding of address space is important, and provides techniques for managing persistent address space, which is a limited resource.

### Address space: What is it and how is it used?

From the *Managing ObjectStore* manual in the ObjectStore<sup>®</sup> documentation set:

“Address space is a range of virtual memory addresses. Each client process has its own address space. Address space includes

- persistent address space – this is a range of addresses that ObjectStore uses to store only persistent data
- transient address space – this includes all addresses in the address space except those designated for persistent data
- 

Just as each client has its own address space, each client has its own persistent address space, called the persistent storage region, or PSR. Two environment variables control the size and address range of the PSR. OS\_AS\_SIZE and OS\_AS\_START have different default values on different platforms.”

The PSR is critical to the function of ObjectStore. As an application accesses objects in the database, those objects are read into pages in the PSR. Additional pages in the PSR might be reserved based on pages being read into the PSR or already read into the PSR.

For example, suppose there is a person object that has pointers in it to a mother object and a father object. This person object, call her Sally, resides on some physical page in the database. Call this page P. Sally has pointers to her mother (Ethel) and her father (Thomas). Neither the Ethel nor Thomas person instances reside on page P. When the Sally object is read from the database in the application, ObjectStore does its patented memory mapping techniques (for more information, see the white paper [ObjectStore Architecture Introduction](#)) and places a copy of the Sally instance into the PSR. Additional accesses for object Sally occur at in-memory speeds.

Mapping the Sally instance into the PSR consumes some address space. ObjectStore allocates persistent address space in sections referred to as chunks. Address space chunks are 64K. Consequently, the amount of address space consumed when the Sally object is accessed is:

- 1 chunk to cover the page (or pages) on which Sally physically resides
- possibly 1 or 2 other chunks to cover the page or pages on which the Ethel and Thomas objects physically reside

If the Ethel and Thomas objects reside on pages in the database that are within 64K of the Sally object, then it is possible that only one 64K chunk of the total PSR will be consumed. For example, Sally, Ethel, and Thomas might have been clustered to improve concurrency. For more information, see [Clustering Techniques](#), the whitepaper. However, if Sally, Ethel and Thomas were each physically located at least 64K away from one another in the database, then three 64K chunks of address space would be consumed. It is somewhat important to note that only one of the possible three address space chunks is actually being used at this point in time. The other two are reserved, but not yet in use. The distinction is subtle. The 64K chunk of address space that includes the page on which Sally resides is physically in use. That chunk has had a physical database page mapped into it. The other chunk or two chunks are reserved for use. If the application dereferences the mother pointer or the father pointer from Sally, then there **MUST** be space in the PSR for those objects to be mapped in from the database. Thus, the address space is reserved for those objects. At transaction end time (commit or abort) all address space that was being used either via consumption or via reservation is released. Thus, in a subsequent transaction, the entire PSR resource is available for use by the newly started transaction.

### **Address space: Why is it important?**

As with any resource, there is a limited amount of space within the PSR. The PSR is actually a subsection of the total virtual memory (VM) for a process. In addition to the PSR, the VM is used for calls to `mmap`, loading shared libraries and the process heap and stack. Because things other than the PSR occupy virtual memory, the PSR cannot grow infinitely; thus, it is a limited resource.

If an ObjectStore application attempts to access more data in a single transaction than can fit in the PSR, then `err_address_space_full` is signaled. There must be enough space to map the objects from the material database on disk into the section of virtual memory assigned to be the PSR. As outlined above, the PSR can be filled by consumption (pages physically mapped into the PSR) as well as by reservations for possible future mappings.

Therefore, it is important to note how persistent address space can be consumed, and design accordingly so as not to deplete the limited PSR resource. Although `err_address_space_full` is not a fatal exception (it can be caught and handled) it is an exception nonetheless. It is better to code so as to avoid the exception rather than try to handle it after the fact. Approaches to managing the PSR range from not at all intrusive (no code changes required) to quite intrusive (code and database design changes). We will examine each of these starting with the least intrusive.

### **Managing the PSR**

The techniques for managing persistent address space include:

- Increasing the size and/or changing the location

- Recycling space in the PSR
- Designing for less consumption

Before discussing these techniques, an understanding of the default size and location of the PSR is required.

### Default size and location of the PSR

Previous to R6, ObjectStore attempted to improve the performance of applications via a technique called relocation optimization. The process of mapping an object from the database into the PSR for the client application to use is called relocation. The idea of relocation optimization is this: if each time a given object is referenced it can be mapped into the PSR in the same location, then less work needs to be done by ObjectStore to prepare the object for use. For example, using our Sally object from before, she has a pointer to her mother, Ethel and a pointer to her father, Thomas. When each of these three objects is read from the database by the client application for the first time, ObjectStore must reserve address space and then “fix up” the pointer values from Sally to Ethel after the Ethel object has been mapped to an actual PSR location. At transaction end, all address space is released. In a subsequent transaction, if Sally and Ethel are accessed, and can be mapped into the same location as the first access, then the ObjectStore client has less work to do. Thus the application runs faster with relocation optimization

A key component to relocation optimization is keeping the starting location of the PSR constant. Therefore, in older versions of ObjectStore, the PSR had a standard starting location and size on each platform. Starting with R6 of ObjectStore, relocation optimization no longer requires the PSR to be in a fixed location. Thus there is no default starting address of the PSR. There is, however, a default size chosen for each platform as shown in the following chart:

Operating System	PSR Size
AIX	1024 Mb
HP 32bit	192 Mb
HP 64bit	768 Mb
Linux	256 Mb
Solaris 32bit	192 Mb
Solaris 64bit	192 Mb
Windows	512 Mb

### Increasing the size and/or changing the location

If your application is raising the exception `err_address_space_full` and you are using the default values (as shown in the chart in the previous section) then a short-term solution

could be to increase the size of the PSR. PLEASE NOTE: increasing the size of the PSR is intended as a short-term solution. Although it might help to alleviate the immediate problem, the PSR is a limited resource and as such, this solution does not scale. As your application scales to handle more data, you might find the need to further increase the size of the PSR. There is a limit to both the general address space available on any machine as well as the amount of PSR on any machine. From the *Managing ObjectStore* manual: “Address space is a range of virtual memory addresses. For most 32-bit computers, the address space is slightly less than  $2^{32}$  bytes.” The PSR is a subset of the total amount of address space available to a client process. Within the total address space must fit the PSR, shared libraries, the heap and stack, and any calls to mmap. For this reason, increasing the PSR is considered a short-term solution that does not scale.

With the above caveat, the way to increase the size of your PSR is to use the environment variable `OS_AS_SIZE`. This variable controls the total size of your PSR. Using the default value chart shown above, on the Solaris platform, the default value for `OS_AS_SIZE` is `0xC00000` (192 MB). The size of the PSR could be increased by setting `OS_AS_SIZE` to a larger value. It is critical to note that the maximum available size for the PSR is application dependent. If your application makes calls to mmap or if you have a large number of shared libraries (which are loaded into the address space) then there will be less total address space in which to allocate the PSR.

There is another closely related environmental variable: `OS_AS_START`. This variable specifies the location of the beginning of the PSR range of virtual memory addresses. As noted above, the ObjectStore code no longer sends a location parameter when calling to the operating system to create the PSR. Rather, a zero is passed as the starting location and the operating system chooses a location that is not occupied and that can accommodate the specified size of the PSR.

In order to maximize your PSR, you might have to move its starting location. There might be times when the application developer wants to manually place the PSR within the open space of the total address space range. For example, if the application grows the heap a great deal over the course of the process, then it would be best if the PSR were located far away from the heap. In order to accomplish this, set the `OS_AS_START` environmental variable to an appropriate address.

The support organization has developed a tool to help analyze how much of the total address space is being consumed and where. In essence, this tool lets you visually see the holes in the overall address space, the sizes of those holes and thus where you can place your PSR in order to maximize it. On the Windows platform this tool is called MemView. On the system IV derivative UNIX platforms, there is no executable prebuilt; however the code to analyze the address space is available for customers to compile and run. For information on how to download the MemView tool, or the system IV derivative code, contact the support organization.

**Recycling space in the PSR**

Even with a maximized PSR, an application might need to access more information within one transaction than can fit within the PSR. In this case, the application must use alternative approaches to managing the persistent address space. One approach is to recycle the address space as it is being consumed. The application processes data within a transaction, and if the data is no longer needed, the application frees the address space that was consumed to process that data.

Consider an example: the database contains a large collection of objects. There is some method that must iterate over the entire collection, examining each item in the collection, and then summarizing the findings. For the sake of data consistency, the iteration and summary must occur in one atomic transaction. If the data contained in the collection were larger than the maximum possible PSR, then the operation would not be able to complete. However, with the ObjectStore API object called `address_space_marker`, the operation can complete. Using an `address_space_marker`, you mark the beginning of the iteration process. Then, every N iterations through the collection, make a call to release the address space consumed since the marker was placed. In this way, items 0 through N are examined in the application and then the address space for that set of items is released. The iteration is moving forward through the collection, thus items 0 through N are not going to be revisited. Whatever information needed to be examined or collected from those items should have been extracted when the object was visited by the iteration. Because the items will no longer be examined, the address space that was consumed to map those objects into the PSR can be released. Once released, the address space can be reused for the next N items to be examined.

It is important to note that the address space consumed for soft pointers is NOT released when the release method is called on an `address_space_marker`. If you want to have address space released from the soft pointers, then you must make an explicit call to `decache` the soft pointers.

It is also important to note that `address_space_markers` are not thread safe. If an application has multiple threads in the same session, then you must take care to ensure that no threads are accessing any of the objects covered by an `address_space_marker` before that marker is released. Threads in other sessions are permitted to access objects without respect to the `address_space_markers` in other sessions as each session has its own address space.

**Designing for less consumption**

In the above example, the application lends itself to the use of `address_space_markers`. However, it might be better to design the application and/or data so that less address space is necessary. Again, think of scaling the application. If/when the application needs to scale, will the `address_space_markers` still solve the overall issue of address space consumption? Here are several different design techniques each of which can help reduce address space consumption:

- 1) Use smaller transactions
- 2) If using transaction checkpoint, commit regularly
- 3) If using `objectstore::retain_persistent_addresses`, exercise caution
- 4) Use soft pointers

### **Use smaller transactions**

Address space is consumed per transaction. At transaction commit or abort all address space that has been consumed is released. Within a given transaction, the amount of data accessed must fit within the PSR. If address space is being exhausted the most straightforward solution is to reduce the amount of data accessed in the transaction. If, for instance, the application code has a loop to process N items, the boundary of the loop can be changed so that fewer objects are visited and processed in the loop. Fewer objects accessed means fewer objects mapped in, which in turn means less address space is consumed.

### **If using transaction checkpoint, commit regularly**

Transaction checkpoint can be used to commit data without incurring all the overhead of ending a transaction and then starting a new one. Unlike transaction commit, after a checkpoint, the validity of pointers to persistent objects is retained. In order for ObjectStore to retain validity of pointers to persistent storage, the address space being used in the transaction *cannot* be freed. In a transaction commit or abort, address space is freed. Therefore, if using transaction checkpoint and address space is needed, the application must call `commit` periodically to release address space.

### **Use caution when calling `objectstore::retain_persistent_addresses`**

The API `objectstore::retain_persistent_addresses` can be used to retain the validity of all persistent addresses across transaction boundaries. This approach can be used to quickly convert existing applications to use ObjectStore. It can also be used to simplify application code so that there is no need to retrieve database roots and traverse pointers to obtain a given object in each transaction. As with transaction checkpoint, ObjectStore must keep address space in use in order to implement this feature. Therefore, like transaction checkpoint, address space can become an issue – especially as an application scales or a large amount of data is accessed. The solution in this scenario is to periodically call to release all persistent addresses. If there is some number of addresses that the application always needs, those can be retained by using `os_retain_address` on a per address basis rather than retaining all addresses.

### **Use soft pointers**

Soft pointers offer an alternative to regular application pointers, also called hard pointers. The advantage of soft pointers is that they consume less address space. Consider our Sally example – she has pointers to her mother and father objects. If those pointers are standard C++ pointers, then they are hard pointers and we must reserve address space when object Sally is read from the database. If, on the other hand, the pointers were soft, only one address space chunk would be required when accessing object Sally. The possible extra chunk or two of address space would not be reserved. Soft pointers reduce

address space because they defer the reservation of address space until such time as the soft pointer is actually dereferenced.

A disadvantage of soft pointers is that a dereference is slower than that for a hard pointer. To counter-act the disadvantage, soft pointers are cached. This is important to note because, for instance, the `address_space_marker` API does NOT release address space for soft pointers. If, in fact, the application is using soft pointers and wants to release address space consumed by those soft pointers, an explicit call to decache the soft pointers is required. Whether a pointer is declared hard or soft does not matter with regard to schema compatibility. This means that one application can use a schema that defines Sally's pointers as soft, while another application can use a schema that defines Sally's pointers as hard. No code changes are required; no schema evolution; no conversion from one pointer type to another. Thus, the use of soft pointers is relatively easy and can provide significant address space savings.

## **Conclusion**

ObjectStore can provide in-memory speeds for access to data. To accomplish this, a section of virtual memory is used to store persistent objects. This resource, the Persistent Storage Region (PSR), is a critical and limited resource. As such, it must be managed. This paper has presented an overview of the reasons why address space is critical, and why it is limited, as well as several approaches for managing the resource. Any number of the options can be combined, or used individually to best accommodate a given application.

ObjectStore is a registered trademark of Progress Software Corporation in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners