The User-level Remote Swap Library

Martin Rehr and Brian Vinter
eScience center, University of Copenhagen, Copenhagen, Denmark

***Abstract*—*This paper introduces the User-level Remote Swap Library, URSL, which enables memory-homogeneous Grid execution. This is obtained by letting the Grid infrastructure supply a remote memory bank to the connected resources. Insufficient memory at the contribution computer resource in a Grid is a major limitation when users seek to get high throughput in a Grid infrastructure, simply because a number of the available computers do not offer enough memory for the user job to run. URSL bypasses the lack of memory by offering the required memory as part of the Grid infrastructure in the form of a user-level swapping resource. A set of dedicated memory servers provide the additional memory that the computing resources then access remotely through the Grid infrastructure. Utilization of the extra memory is obtained without modifications to neither the operating system of the host resource nor the executed applications. The paper includes experiments that are made in an isolated Grid framework, and shows that the framework is fully operational, transparent and outperforms swapping to local disk in both the synthetic micro-benchmarks and in real-life scientific applications.*

Keywords: Remote Swap, Grid

I. Introduction

The scientific modeling community has a seemingly endless need for processing power as new areas of modeling arise steadily and existing models become increasingly fine grained and realistic. To be able to keep up with the growing demand for processing power the Grid[13] paradigm was introduced in the late 90's with the purpose of providing an infrastructure that combines super-computer installations located at different research institutions into one high performance infrastructure which in turn enables sharing the computer resources among the researchers across organizations. In the same period Berkeley University introduced Public Resource Computing (PRC) in the form of the SETI@home[4] project, which later turned into the BOINC[3] framework. The PRC paradigm differs from the Grid computing paradigm by focusing on the large set of computing devices that are located outside the super-computer facilities, which has been demonstrated to have a huge computation potential. At the time of writing the most successful PRC project FOLDING@home[8] has a total of 325638 active CPU's that contribute a theoretical total of 4432 TFLOPS[1], which potentially makes it the second fastest super-computer installation in the world right now[2]. A number of research projects have shown ways to combine PRC computing and PRC computing, the first using BOINC resources within a Grid infrastructure[16] and the latter by harvesting generic windows resources[1] and Playstation 3 resources[17] in a Grid environment. This work provides a possibility of utilizing the processing power that was previously only applicable to PRC computing in a Grid infrastructure, because the resources would not offer enough memory for the Grid jobs. While PRC resources offer a huge boost to the theoretical processing power that is available in a Grid infrastructure it also introduces a number of new challenges. Using millions of PRC resources is necessarily more heterogeneous than a few super-computer installations, which impacts the size of the set of resources that are capable of executing any given job and thus the overall utilization of the Grid system. There are four levels of homogeneity, which needs to be addressed: Hardware-, OS-, disk- and memory-homogeneity. Hardware- and OS-homogeneity may be addressed by wrapping the Grid execution environment into a virtual machine image, disk-homogeneity may be applied by using a remote file library[2], but so far there is no mechanism for providing memory homogeneity in a Grid framework. We propose the Remote User-Level swap framework in order to enable memory homogeneity to Grid computing and thus increase the total system utilization[3] This framework provides Grid resources with a mechanism for seamlessly accessing memory that is located in the Grid infrastructure, without modifying neither the resource nor the job that is executed.

A. Related Work

Several research groups have looked into the idea of swapping to a remote machine rather than a local disk. D. Comer and J. Griffioen[10] present a model where a dedicated memory server is used to store pages that are swapped out by the clients. E. Markatos and G. Dramitinos[14] present a reliable remote memory pager that makes use of free memory in the nodes of a cluster. E. Anderson and J. Neefe[5] present a user-level remote memory pager that targets Network Of Workstations (NOW's). R.T. Mills, C. Yue, A. Stathopoulos and D.S. Nikolopoulos[15] presents a user-level remote memory system for scientific applications that uses local disk and dedicated memory servers. R. Chu et. al.[9] present remote memory paging using NOW's in a Grid environment.

The kernel-level models presented in most of the previous work are poorly suited for global Grid environments, because these models require modifications to the operating system of

[1]According to their web-site:http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats

[2]According to the Top-500 list of the worlds fastest super-computer installations at November 2009 (http://www.top500.org/list/2009/11/100). The Top 500 list is based on the Linpack benchmark suite, which is not applicable to a PRC environment, therefore the comparison is theoretical.

[3]Providing the throughput is bound by memory constraints

the executing host resources. While this is acceptable in an isolated homogeneous cluster environment it's difficult if not impossible to deploy in global Grid environments that have thousands of executing hosts and host administrators. With this in mind we chose to extend the existing solutions by combining the idea of dedicated memory servers that was presented by D. Comer and J. Griffioen and the user-level idea presented by E. Anderson and J. Neefe integrated with the Grid approach taken by Chu et al. But opposed to the previous user-level solutions, which requires the use of non-standard memory routines, our solution works transparently to the application and thereby transparently to the application writer and ultimately the Grid user. This leads to a solution that differs from the previous work by providing a Remote Memory framework for global Grid environments with resources ranging from the traditional cluster computer installations all the way to PRC devices. That is, a solution which is deployed through the Grid infrastructure and is fully transparent to both the execution hosts and to the application writer.

II. The Grid infrastructure

The Grid computing paradigm is a natural extension to cluster computing, that aims at assembling a global network of computation resources into one shared, although not parallel, infrastructure. Where the devices used in cluster computers are homogeneous, the devices used in a Grid system are heterogeneous, in all aspects of the hardware configuration, the resource location, and the administrative domain they run within. In this work we aim at improving the usability of Grid resources by providing them with more memory than they offer locally. By letting the Grid infrastructure provide a memory service to supplement local memory, the Grid may take advantage of resources, which would otherwise be idle because no jobs fit the memory they offer. The end result is that the total utilization of the Grid system is increased as a broader set of resources are capable of executing the pending Grid jobs.

A. Grid Resource Memory

In the perfect world a computing device would hold sufficient physical memory to allow all running processes and their data in memory. Naturally this is not the case and we thus need mechanisms to administrate the available physical memory to help optimize utilization of the hardware. This is usually done through the OS memory manager, which has global knowledge of the memory usage of all running processes in the system. However, changes to the operating system is a limiting factor in the adaptation of any system, thus in this work we aim at deploying a user-level swap library, which is placed between the OS and the application and intercepts memory needs before they reach the OS memory manager. Using this approach we introduce a Grid-enabled memory manager that runs non-intrusively at any Grid resource.

1) Memory management: All modern computer architectures have a memory management unit and use an operating system, which provides a full virtual address space to each process. This gives the processes an impression of having exclusive access to a system where the upper limit of memory is bound only by the hardware architecture. While providing a full virtual address space for each process is a powerful abstraction, it also places a large responsibility on the OS to manage the available physical memory. Traditionally this is achieved by swapping memory to disk when the system runs short on physical memory. If the system exhausts the available disk space for swapping, the OS has to decide what to do. Typically the policy is killing a process to free up memory.

2) Kernel-level vs. user-level: The OS memory manager is responsible for swapping page-frames between physical memory and secondary memory, typically a hard-disk. In Linux the swap device is implemented as a generic block device, which makes it easy to change swap-target implementation as one can plug these directly into the kernel just by emulating the behavior of a block device. This approach has the advantage that the memory manager is left unmodified and works independently of which underlying media is actually used as storage for the swapped out memory. The downside to this approach is that it requires administrator privileges to deploy such a block device into the kernel, and implicitly that the administrator will have to trust the code that is added to the memory manager since it will run with kernel privileges.

While the kernel-level approach is easy to implement due to the cleaner interface with the OS, the user-level model is more flexible in a Grid context since it overrides the local memory manager and thus provides a homogeneous swapping mechanism to the Grid resources without requiring administration privileges or implicit trust to the swapping module. This enables the Grid infrastructure to fully control the amount of physical memory that the executing application is allowed to use on the resource [4]. The user-level model however has several drawbacks, primarily that you have to implement a new memory manager to work on top of the native memory manager. However reusing most of the OS memory manager and overloading only a few functionalities such as allocation of memory, freeing memory and swapping pages in and out of physical memory may be sufficient. Another drawback of the user-level approach is that it invokes frequent switching between user- and kernel-level, which results in an execution time overhead compared to the native model where everything is done in kernel space. In addition an overhead in memory consumption is imposed because the user-level library must maintain a set of internal data structures to keep track of the state and location of pages that are used by the application. Last but not least a user-level library doesn't have access to the hardware supported status bits of the process page-table, which has the effect that the widely used LRU eviction algorithm is not applicable [5]

Despite the overhead of the user-level approach compared to

[4]Naturally it's not possible to utilize more physical memory than present at the resource

[5]LRU uses the hardware page referenced bit, which is not accessible at user-level. A workaround can be made, but introduces a significant overhead due to switching between user- and kernel-level

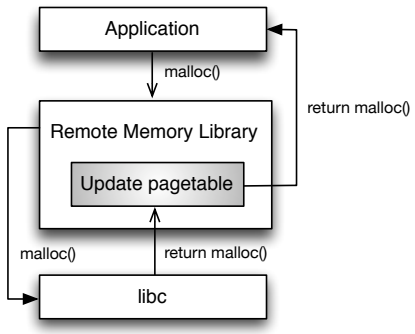

Fig. 1. The flow of a malloc call from the user application

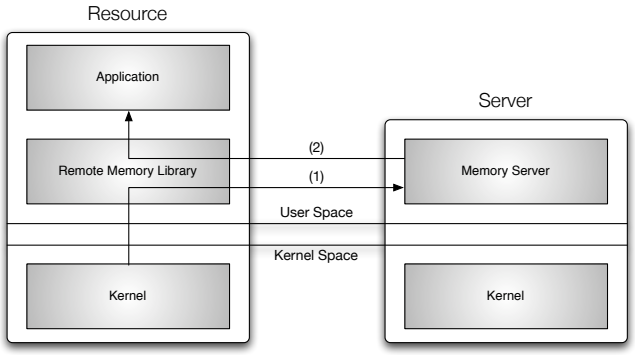

Fig. 2. The flow of swapping in a page from the remote server. (1) The kernel catches a page access violation which is sent to the user process but intercepted by RML and transformed into a server page request. (2) The server responds with the page needed which is transferred to the client and mapped into the right position in memory and control is given back to the user process

the kernel-level approach, we choose to make our remote swap library run at user-level. This is chosen because it imposes fewer requirements for deployment in Grid environments, as well as the opportunity to change the page replacement algorithms with algorithms that are more suited towards a Grid environment.

III. The Remote Swap framework

The Remote Swap framework consists of two components namely a memory server and a Remote Memory Library (RML). This memory library is interposed in user-level between the OS and the user process on the executing Grid resource to provide a transparent user-level swap mechanism. It's responsible for allocating memory, freeing memory and evicting pages to the remote memory server, as well as bringing pages back into physical memory, when they are once again needed.

A. The Remote Memory Library

The goal of making RML transparent to both the user application and the underlying OS implies that RML should support all the memory routines that are available to the application writer in the original environment. In this initial version of RML, Linux heap memory is the target for remote swapping, making the libc routines *malloc, calloc, realloc* and *free* the ones supported in RML. Rather than re-writing and maintaining these routines, they are merely overloaded with the purpose of maintaining a local page-table in RML to keep track of the pages in use by the user process. The actual memory allocation is done by calling the generic memory routines in libc from within the overloaded routines. This is illustrated in figure 1.

1) Page eviction: When the user process reaches its physical memory limit pages need to be evicted. The target pages are found using the second chance FIFO evict strategy rather than the LRU strategy, which is used by most OSs, because LRU is not feasible in a user-level environment. When the pages to be evicted are found, they are protected in read mode to ensure consistency by preventing any other active user threads from modifying them during eviction. If the chosen pages were modified while resident in memory, they are sent to the remote memory server through the network. The physical memory used for the evicted pages are then released and the virtual pages are protected in order to detect when the user process tries to access them.

2) Page retrieval: When the user process tries to access a protected page, the kernel will send, an access violation signal to the user process. This signal is intercepted by RML, which checks the state of the violated page in its local page-table. If the page has previously been swapped out, a page request is sent to the remote server in order to retrieve the page. The server responds with the page data, which is placed into the correct page slot and control is given back to the user process which can continue execution. This is shown in figure 2. If RML doesn't have any information about the violated page the access violation signal is forwarded to the user process which then has to handle the signal.

3) Page blocks: The memory manager in modern OSs arrange memory into an abstraction called page-frames, which is blocks of contiguous bytes. In the same manner RML arranges blocks of contiguous pages into what we call page blocks in order to swap out and retrieve several contiguous pages in a single evict or retrieve operation. This is beneficial if the executing application has a sequential memory access pattern across page blocks, that is the byte access pattern within a page block can be scattered as long as it doesn't access bytes outside the page block or its adjacent neighbors. Not all scientific applications have a strictly sequential memory access pattern, but even then they may still take advantage of using blocks of pages when evicting or retrieving pages. This is clarified in the experiments section. The optimal number of pages to block into one page block is dependent on the network latency between the client and the memory server, and the memory access pattern of the executing application. In this first version of RML the page block size is provided to the framework before execution. Adaptive adjustment towards the needs of the executing algorithm is subject for future work.

B. The Memory Server

The memory server is a user-level process communicating with the clients through a TCP socket. Two kinds of services are offered by the memory server: Page send and page retrieve. When the client asks for a service the index of the page to send/retrieve is sent along with the request. The server stores the pages in memory and uses a hash table with the page index as key and the memory address where the page is stored locally at the memory server as value. When a page is received at the server the page index is looked up in the hash table to check if memory has previously been allocated for the specific page due to an earlier swap-out. If the page has an entry in the hash table the memory associated with it is reused, otherwise memory is allocated for the new page and the key/value pair is inserted into the hash table.

Upon a page retrieve request the server makes a lookup in the hash table, if the page index is present in the hash table the server responds with the data associated with the page index otherwise an error message is sent to the client.

IV. Implementation details

The current implementation of URSL is bound to the Linux kernel, but is portable to any page based operating system that supports virtual memory and functions to manipulate the virtual page tables such as mmap, mremap, munmap and mprotect or equivalents. In addition to the page table manipulation functions, support for overloading the default signal handler is required in order to detect page access violations at user-level. URSL is loaded in between the system level libraries such as libc and the user application by using the LD_PRELOAD environment variable. This way the library can be a part of any Grid job without involving the resource owner, as it's delivered along with the Grid application and loaded as a part of the Grid job. When the library is initialized malloc, calloc, realloc and free is overloaded and thereby every call to these functions passes through URSL, which means that the user application doesn't need to use customized memory functions in order to use the framework, this is what provides the transparency to user applications.

A. Memory allocation

In order to avoid re-implementing the existing memory allocation routines, URSL merely uses the original malloc, calloc and realloc routines to allocate memory. When allocating a chunk of memory, the address returned by the original allocator along with the chunk size is translated into page indexes. These indexes are used as keys for a local page table containing the page state information. The first and the last page are typically used by the original allocator to store internal information about the allocated memory and thereby these are mapped to real memory, the pages in between are merely marked as allocated both at operating system level and in the local URSL page table. These pages are access protected by URSL using the mprotect routine in order to detect when they are activated. The framework then returns the allocated memory address to the running application which continues its execution. When one of the newly allocated access protected pages is accessed by the running application, an access violation signal is thrown by the kernel. This signal is caught by URSL and the page is looked up in the local page table. If the page is marked as allocated, but not yet used, the page is re-protected in mode write, the page table is updated, and execution is returned to the running application.

B. Page eviction

The user-level approach makes it possible to regulate the real memory usage of a single application without considering other active applications on the system. When a page becomes active, either by activating a new page or swapping in a previously used page, it's checked if the upper active page limit has been reached. If the limit is reached pages are chosen for eviction using the second chance FIFO algorithm and thereafter protected in *read* mode, to prevent other active threads from modifying the pages during eviction. Modified pages are then sent to the remote server along with a header containing the page indexes. Finally the page states are changed to *swapped-out* and the real memory is freed by mmap'ing the evicted pages in protection *none*. This atomically frees the real memory and maps the virtual pages in *no access* mode. The evicted pages are now resident at the remote memory server until they are once again accessed from the running application.

C. Page retrieval

When pages are swapped out they are protected in *no-access* mode, this means that whenever the executing application accesses such a page, an access violation signal is sent by the kernel and trapped by URSL. The page causing the access violation is looked up in the local page table and if the page is found to be resident at the memory server, a page request is sent. The server will respond with the requested page data, which is received in a local buffer reserved for swap-in page data. When the data is fully received, the receive buffer is protected in mode *read* to detect future modifications, used to determine if the page should be written back to the server upon its next swap-out. When the receive buffer is protected the page is put into the right position in memory by using the mremap routine. This routine atomically updates the kernel virtual page table keeping other threads from accessing the page until the page is in a write safe state[6]. Finally the page is marked in URSL as *swapped-in*, the page retrieve buffer is re-allocated using mmap for future swap-in's and the execution is returned to the thread which accessed the swapped-out page.

[6]Without the remap routine one would need to make the target page writeable, in order to copy data from the buffer page to the target page slot. This would allow other user threads to modify the page before it's completely in place.

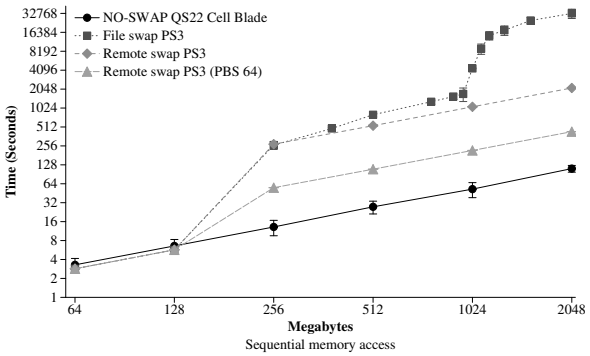

Fig. 3. Sequential memory access performance

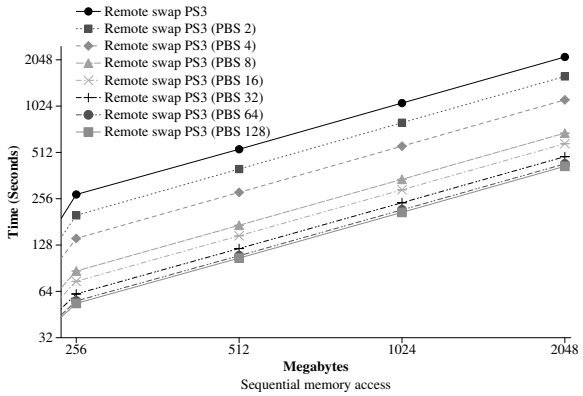

Fig. 4. Sequential remote memory performance with increasing page block sizes

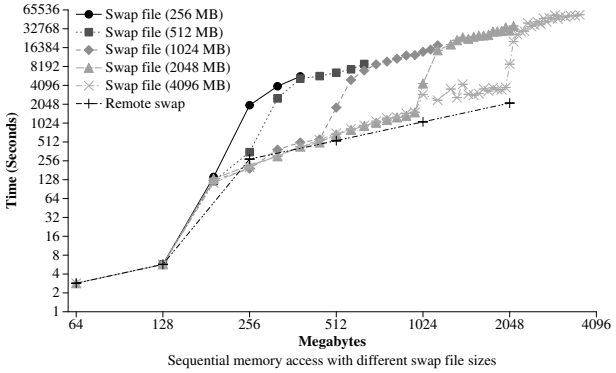

Fig. 5. Linux file swap performance

V. Experiments

To validate the Remote swap library and document its performance, experiments were done in an isolated execution environment. This consisted of a Playstation 3 execution node and an dual quad core Intel Xeon running at 1.60 GHz with 8 GB RAM as memory server. These machines were interconnected through a 1 Gb/s switch and controlled by the Minimum intrusion Grid[12]. The Playstation 3 was chosen as execution device because it represents a unextendible hardware device with a powerful processor but a limited amount of memory, namely 224 MB for the OS and user applications. As a reference machine without swap we used a Cell-BE QS22 blade. The reason for using an isolated high-bandwidth network with only one node, rather than a full Grid setup with a slower network and a number of nodes, is to validate the performance of the transparent user-level model under optimal conditions. If the model doesn't perform well under these conditions it will never perform in a real Grid system.

The URSL framework has been tested with special designed highly I/O bound sequential memory access and scattered memory access applications, as well as real scientific applications. Each of the applications were tested with different page block sizes to evaluate how this influences the performance. The results of the experiments are covered in the following subsections.

A. Sequential data access

The sequential data access tests were done by allocating N bytes of memory using malloc and then initializing the memory to make sure it was mapped to physical memory. Then a timer was started and the time spent traversing the memory start-to-end in 10 iterations (reading each byte to produce a checksum) was measured. Finally the performance was compared to the execution without swap on the Cell-BE QS22. This test is highly I/O bound as the only computation done by the executing machine is one integer addition per byte that is read. The performance of this execution with page block sizes 1 and 64 is shown in figure 3. The experiment shows that URSL outperforms swap to disk significantly, as the memory consumption increases. Furthermore, the performance increases with the page block size until it starts to converge at 64 pages per block, which is were the bandwidth of the network is saturated. This is shown in Figure 4. We found the steep raise in disk swap execution time when the memory consumption reaches 1024 MB peculiar, as we are only performing sequential reads while measuring time and would expect the disk and its caches to be able to prefetch the pages needed. Thereby we would expect the execution time to raise no more than linearly with respect to the memory consumed and perform better than a user-level remote swap library as this algorithm is highly I/O bound. To investigate this further we settled out to take a closer look at the Linux swapping scheme.

1) Linux disk swap: In this experiment we executed the sequential memory access program described in the last section using different swap file sizes to see if the execution time was affected by the size of the swap device. All swap files were made using dd with a block size of 4096 bytes. The result of the executions is shown in figure 5. This experiment shows that disk swap performance is highly dependent on the swap file size, which was an unexpected result. We discovered the reason for this by looking into the Linux kernel source code where we found that when half of the disk swap space is filled the memory manager starts to remove swapped-in pages from the swap device to prevent it from running out of space. The effect is that all swapped-in pages are marked as dirty and

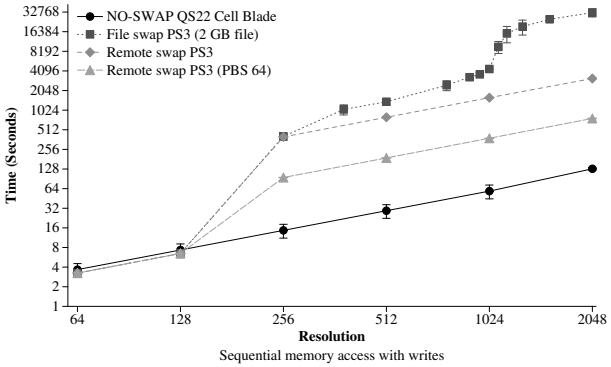

Fig. 6. Sequential memory access with writes performance

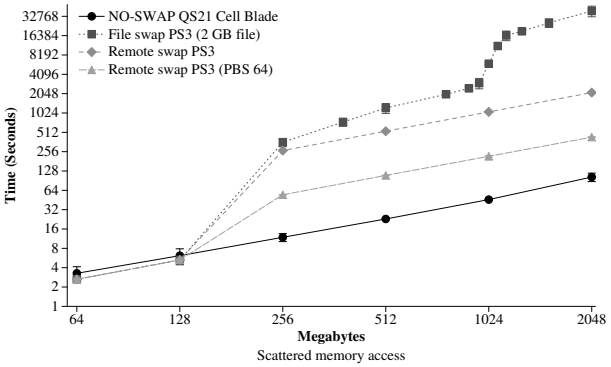

Fig. 7. Scattered memory access performance

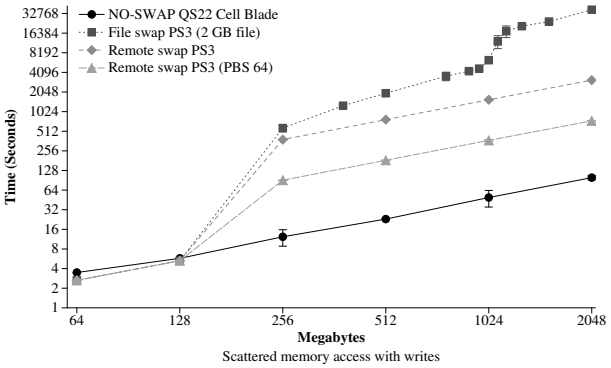

Fig. 8. Scattered memory access with writes performance

thereby needs to be re-written to disk when they are once again swapped out. This is what causes the steep raise in execution time, when the swap device becomes half full, as the sequential experiment doesn't perform any writes in the 10 iterations that are used for time measurement.

In the rest of the experiments we continued to use a 2 GB swap file to show how the artifact of the Linux kernels half-full dirty mark strategy will be expressed within the scientific applications used in the experiments.

B. Sequential data access with writes

In this test we used the sequential test described in the previous section and performed a write to each page accessed after its data had been read. The result is shown in figure 6. This experiment shows that even though a write was done to every page, it didn't eliminate the half-full Linux swap artifact, but the gap is smaller compared to the execution without writes. This is due to the fact that the Linux kernel in addition to marking the page dirty also removes it from the page cache and the swap device, which consumes time compared to the alternative of leaving the page on disk and then just writing it back out when needed. Beside the effect of the half-full artifact, it is observed that the execution time increases when using disk swap compared to remote swapping. This is caused by the mechanical structure of a disk, which causes the average seek time to rise as the amount of swapped out pages increases. As in the experiment without writes, the performance increases with the page block size until it converges at 64 pages per block, when the bandwidth of the network is saturated.

C. Scattered memory access

This test was designed similarly to the sequential access test, but instead of reading the pages in a sequential manner the data is read one page at a time, starting with the first page followed by the last page and then the second page followed by the second last page, this pattern is used until all pages are read. The result is shown in figure 7. This test shows that the execution time when swapping to disk is increasing relatively more than the execution time when swapping to the remote location. That is caused by the fact that disk swap can no longer take advantage of block prefetch combined with larger seek times when searching for the page to swap in, even before the half-full artifact arises. As with the sequential tests the performance increases with the size of the page blocks until the network is saturated at a block size of 64 pages.

D. Scattered memory access with write

This test is like the scattered access test described above, but with one write to each page like in the sequential write test. The result of this test is shown in figure 8. As with the sequential write test, the gap between Linux swap to disk and URSL decreases compared to the execution without writes and the half-full artifact of the Linux swap is still present. The performance increases with the size of the page blocks until the network is saturated at a block size of 64 pages.

E. Lattice Boltzmann

OpenLB[18] is a free library for lattice Boltzmann simulations. In this experiment we used the forcedPoiseuille2d example provided in the package with different resolution sizes. The results of this test is shown in figure 9, which displays that swapping to the remote location outperforms disk swap with a factor of 6 and is 11 times slower than no-swap execution with a resolution of 2048 and a page block size of 4, which proved to be the optimal page block size for this application (figure 10). The half-full artifact is clearly visible in this test.

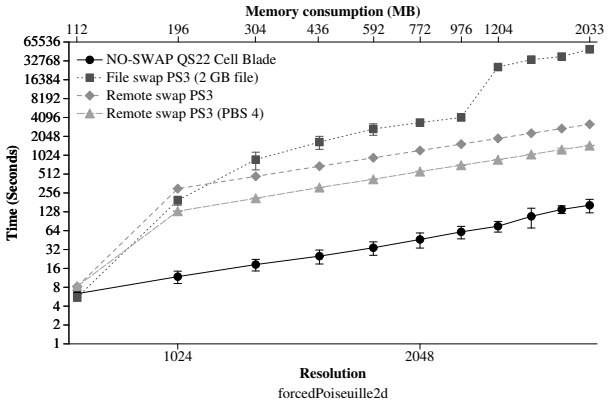

Fig. 9. Lattice Boltzmann performance

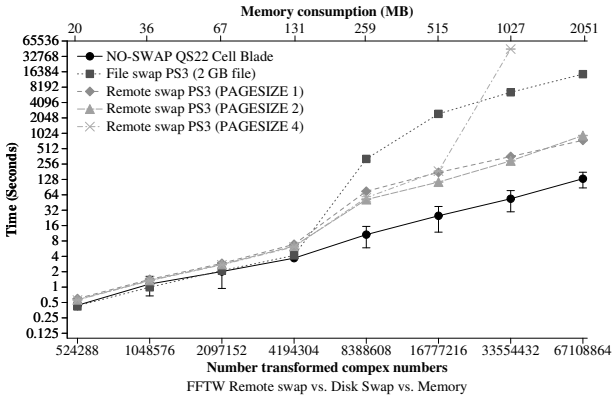

Fig. 11. FFTW performance

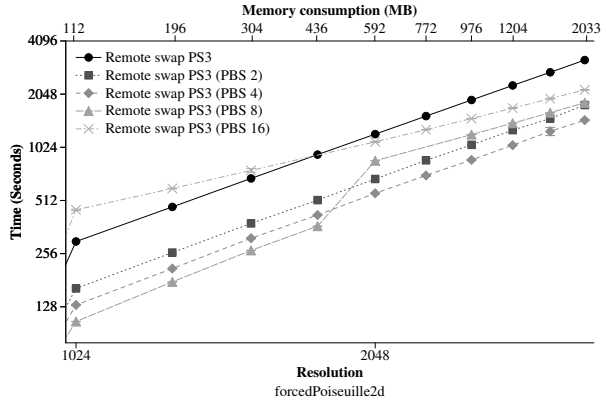

Fig. 10. Lattice Boltzmann performance with increasing page block sizes

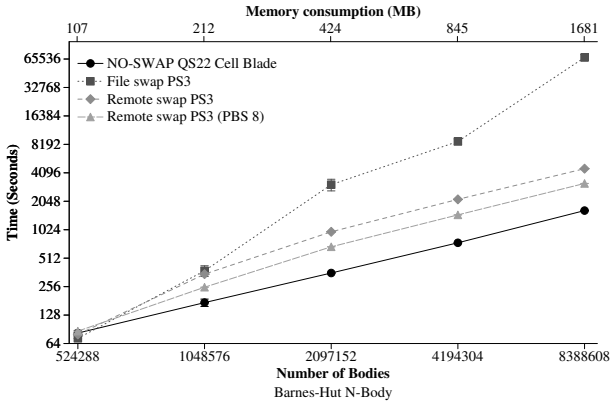

Fig. 12. Barnes-Hut performance

F. Fast Fourier Transform

Fftw[11] is a free implementation of discrete Fourier transformation. In this test we use Fftw to transform a vector of random complex numbers to their corresponding Fourier values and back to the original values. The result of this test is shown in figure 11, which displays that swapping to the remote location outperforms swapping to disk with a factor of 21 using a vector of 16777216 complex numbers and a page block size of 2. The slowdown compared to native executing at this instance is 4. Furthermore it shows that a page block size of 1 is the optimal when we reach a vector size of 67108864 complex numbers.

G. Barnes-Hut

The Barnes-Hut[6] algorithm is an O(nlogn) algorithm for performing N-Body force simulations. In this experiment we have used the code that is provided by one of the authors J. Barnes (the code can be downloaded from his ftp site[7]). The experiments were performed with the tree-body 6 test data, provided in the original source package, with a variable number of bodies and a random seed of 12345. The results is shown in figure 12, which displays that remote swap outperforms swap to disk by a factor of 6 and is a factor of 4 slower than native execution in real memory, when using 4194304 bodies and a page blocks size of 8, which is the optimal page block size (figure 13) for this application. It should be noted that the difference between swapping to disk and swapping to a remote memory location increases as the number of bodies grows and that the half-full artifact is present in this experiment.

VI. Experiment summary

The experiments show that swapping to a remote memory server outperforms swapping to disk on the Playstation 3 platform, both in the tests that are designed specially towards the framework, as well as in the generic scientific applications, which have not been modified in order to work with the framework. The speedup varies from 6 to 21 in the tested scientific applications and the slowdown varies from 4 to 11 compared to native executions. We have also shown that Linux disk swap has serious issues when it comes to scientific applications. The conservative strategy of removing pages from the swap device, whenever it becomes half-full, is poorly suited

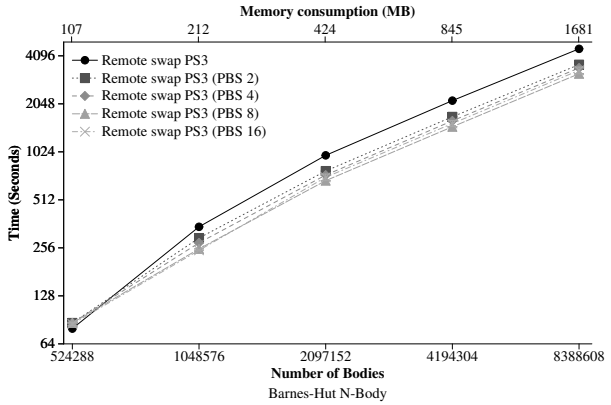

Fig. 13. Barnes-Hut performance with increasing page block sizes

for scientific applications, due to their nature of initializing data and performing several iterations on the same data set, which means that the same pages are accessed several times without any renewal. Furthermore devices that are dedicated to scientific applications should favor the scientific application over other processes running at the system regarding CPU and memory.

VII. Conclusion

In this work we have presented a method for providing remote swap to global Grid infrastructures. Opposed to previous presented models, we present a fully transparent user-level library, which can be submitted along with the Grid jobs eliminating the need to modify neither the OS of the executing Grid resource nor the Grid application to execute. Furthermore the user-level approach makes it possible to throttle the real memory usage of the running job, through the Grid middleware, and thereby increase the pool of resources capable of fulfilling the memory requirements of a given job. Last but not least the user-level approach ensures that only pages that are used by the Grid application are subject for eviction. The disadvantages of using the transparent user-level approach is the time overhead of passing signals, page mappings and page protections between kernel- and user-level, as well as the space overhead of keeping a local process page table within the framework as one can't access the page data-structures of the kernel from user-level.

Experiments show that the framework is operational and despite the introduced overhead still outperforms swapping to disk by a factor of 6 in the worst case. In the best case the framework outperforms swapping to disk with a factor of up to 21.

References

[1] Rasmus Andersen and Brian Vinter. Harvesting idle windows cpu cycles for grid computing. In Hamid R. Arabnia, editor, *GCA*, pages 121–126. CSREA Press, 2006.

[2] Rasmus Andersen and Brian Vinter. Transparent remote file access in the minimum intrusion grid. In *WETICE '05: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, pages 311–318, Washington, DC, USA, 2005. IEEE Computer Society.

[3] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[5] Eric A. Anderson and Jeanna M. Neefe. An exploration of network ram. Technical report, Berkeley, CA, USA, 1994.

[6] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[7] Joshua Edward Barnes. ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode.

[8] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Rui Chu, Nong Xiao, Yongzhen Zhuang, Yunhao Liu, and Xicheng Lu. A distributed paging ram grid system for wide-area memory sharing. In *IPDPS*. IEEE, 2006.

[10] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In USENIX, editor, *Proceedings of the Summer 1990 USENIX Conference: June 11–15, 1990, Anaheim, California, USA*, pages 127–136, Berkeley, CA, USA, Summer 1990. USENIX.

[11] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on 'Program Generation, Optimization, and Platform Adaptation".

[12] Henrik Hoey Karlsen and Brian Vinter. Minimum intrusion grid - the simple model. In *WETICE '05: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, pages 305–310, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[14] Evangelos P. Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.

[15] Richard Tran Mills, Chuan Yue, Andreas Stathopoulos, and Dimitrios S. Nikolopoulos. Runtime and programming support for memory adaptation in scientific applications via local disk and remote memory. *J. Grid Comput.*, 5(2):213–234, 2007.

[16] D S Myers, A L Bazinet, and M P Cummings. Expanding the reach of grid computing: Combining globus- and boinc-based systems. *Journal of Parallel and Distributed Computing*, 2004.

[17] Martin Rehr and Brian Vinter. The ps3 grid-resource model. In Hamid R. Arabnia, editor, *GCA*, pages 90–95. CSREA Press, 2008.

[18] Open source lattice Boltzmann code. http://www.openlb.org.