Malleable Memory Mapping: User-Level Control of Memory Bounds for Effective Program Adaptation

Dimitrios S. Nikolopoulos
Department of Computer Science
The College of William&Mary
McGlothlin Street Hall
Williamsburg, VA 23188

Abstract

This paper presents a user-level runtime system which provides memory malleability to programs running on non-dedicated computational environments. Memory malleability is analogous to processor malleability in the memory space, i.e. it lets a program shrink and expand its resident set size in response to runtime events, without affecting the correct execution of the program. Malleability becomes relevant in the context of grid computing, where loosely coupled distributed programs assume to run on busy computational nodes with fluctuating CPU and memory loads. User-level malleable memory is proposed as a portable solution to obtain as much as possible out of the available memory of a computational node, without reverting to more drastic solutions such as job suspension or migration, and without causing the system to thrash. Malleable memory mapping is also a solution to cope with the unpredictable behavior of existing virtual memory management policies under oversized memory loads. The current prototype is simple but leaves plenty of room for application-independent or application-specific optimizations, compiler support and other extensions. Our performance evaluation is a proof of concept that grid programs with malleable memory can improve their performance by an order of magnitude as opposed to grid programs that let their memory being reclaimed and reallocated by the OS.

1 Introduction

Multiprogramming has been a thorny problem in the development of efficient programs for non-dedicated parallel platforms. Sharing of processors, memory, and network links may negate any assumptions that the programmer makes on the availability of resources while the program is running. Although the related issues have been a subject of investigation for almost two decades, they are still very much relevant

due to the advent of grid computing. Computational grids are based on malleable resources and significant research effort is placed on developing programming and runtime support for malleable grid programs.

Our work in this context focuses on the *micro-management* of a grid program at the node level. We investigate ways to tune grid programs so that each computational task scheduled to a node can make the most out of the resources available at the node, not only at the time of scheduling, but also at runtime.

We believe that the problem of sharing memory on multiprogrammed servers, clusters or computational grids has received less attention than it deserves. Typically, programmers develop distributed programs under a simplifying assumption of the underlying memory constraints. The programmer has two choices. The first is to measure the size of the resident set of the program and examine if the program fits or doesn't fit in the memory of a grid node. If the problem doesn't fit in memory, the higher level grid scheduler will probably opt for finding another node with higher memory capacity. The second, and most difficult to implement option, is to restructure the program so that the size of the footprint is reduced. One may consider numerous approaches to this problem, including traditional compiler optimizations for memory hierarchies, algorithmic restructuring, or implementing an out-of-core version of the program. We are investigating if there is any "middle ground" between the two aforementioned options, i.e. provide a solution that will help the program run at a reasonable speed with less memory resources than needed, without thrashing local nodes. In addition to that, we are looking for a solution which is portable, does not require OS modifications and can be customized to application-specific characteristics.

Relying on existing virtual memory (VM) management systems does not appear to be the best option in this context. Most VM systems fail to handle oversized memory loads without thrashing or penalizing certain kinds of applications. They are designed to make the common case fast and secure a fair share of memory resources allocated to each job in the system in the *long-term*, but the notion of fairness and common cases is being defined arbitrarily in the OS. VM systems are also hard to understand and writing code for adapting to a specific feature of a particular VM system makes the code non-portable. Changing VM systems to incorporate better VM algorithms for more types of workloads is challenging, and the related work is probably reaching its limits [12]. At the other end, changing the VM system to enable application-specific VM and physical memory management schemes is the most aggressive and customizable solution, but needs a major redesign of the operating system itself. Both options share an important drawback, they need modifications to the operating system, which may or may not be possible and they are most likely non-portable.

1.1 Problem statement

The problem that we are attempting to address with the work presented in this paper is the following: How can we provide runtime support to a program running as guest on a non-dedicated node of a computational grid, so that the program runs as efficiently as possible without thrashing, when the memory available to it fluctuates and the full resident set of the program may or may not fit in physical memory at different snapshots of execution. For this problem, we assume that the program is already optimized for a fixed-sized memory hierarchy. We also assume that the program runs on top of a VM system, with policies which are generally unknown to the program and may harm the performance of the program under certain conditions.

There is a solid motivation for writing programs with malleable memory. Non-dedicated, multiprogrammed servers become increasingly more popular as components of clusters and computational grids. The application basis for these platforms is expanding towards applications which are more data-intensive and have larger resident sets. Grid computing [9] is offered as an alternative for harnessing the available cycles and memory of widely or locally networked systems [1, 2], but faces the problem of harmonic co-existence of local jobs and "grid,, jobs, in any given node which contributes computational resources to the grid.

1.2 Contribution

In this paper, we propose a user-level mechanism which provides memory malleability to grid programs and present its preliminary implementation and performance results. We define memory malleability as the ability to dynamically shrink and expand the resident set of a program, with a mechanism controlled by a user-level runtime system, in response to oscillations of the memory available to the program by the OS. We emphasize three aspects of the runtime system: First, it is a user-level solution for memory malleability. It does not require modifications to the OS and it uses system services common to most, if not all, contemporary OSes, so that it can be portable. Second, it is fairly transparent to the application, in the sense that the application can use malleable memory mapping by linking in a runtime system which provides wrappers to memory allocation functions. It is not binary-transparent (i.e. can not be immediately linked to object code), but we believe that achieving memory malleability of unmodified binaries is feasible, and we consider it as one of the first priorities for our future work. Third, it is expandable in many ways. The runtime system as is, unmaps and remaps application memory transparently using application-independent metrics and does a good job in controlling the resident set of the program and throttling memory consumption when thrashing becomes an issue. It can be easily extended to incorporate application-dependent metrics, hints provided by the application to the memory manager, or compiler support.

We present results obtained on two systems, a small Linux cluster with four dual Intel Xeon-based nodes and a 4-node partition of an SGI Origin2000. The experiments

use a synthesized distributed program to provide a proof of concept on how user-level malleable memory can dramatically improve the performance of programs with memory footprints that do not fit in loaded systems. Significant further investigation is needed to put this research in the context of specific applications.

The rest of this paper is organized as follows: Section 2 overviews related work. Section 3 presents the design and implementation of our malleable memory mapping system. Section 4 presents preliminary performance evaluation results from two platforms, a Linux cluster and an SGI Origin2000. Section 5 summarizes our conclusions and presents the directions of our future work.

2 Related Work

The idea of malleability has been thoroughly explored in the context of job scheduling on parallel systems [8] and dynamic space sharing policies [14] in particular. Although memory malleability resembles processor malleability, the technical details of implementing seamlessly a malleable memory mapping scheme are radically different. In general, the impact of taking processors away from the program and re-scheduling the program's computation on the remaining available processors is relatively easy to understand and model. On the other hand, the impact of taking memory away from a program is much harder to model without full knowledge of the program's memory references. Even if modeling this impact is possible, coming up with efficient methods for adaptation to memory shortage is challenging.

A method to cope with memory shortage in applications with very large working sets is to use out-of-core algorithms. Significant work has been done on providing optimized out-of-core implementations of popular mathematical routines [5, 7, 15] and compiler support for effective composition of out-of-core programs [3, 13]. In principle, out-of-core methods assume problem sizes that do not fit in the memory of the system on which the program runs. We rather address the problem of memory malleability for programs that do fit in the memory of the target platform, but their performance suffers due to contention for memory resources and undesirable interferences with the scheduling and VM management policies of the OS.

Application-specific, user-level memory management is an intensively researched area of operating systems design and implementation [4, 11, 18]. The similarity between these works and ours is that all are attempting to improve the performance of programs in cases where the native VM management algorithms of the OS are likely to fail. There are two important differences though. Application-controlled memory management mechanisms are primarily designed to improve the performance of specific applications, when the OS VM management algorithms do not match well with the application's data access patterns. In general, these algorithms do not consider multiprogramming and memory sharing, or consider it as an orthogonal problem treated inside the OS. We are targeting a different problem, which is how to enable effective adaptation of jobs running as guests on hosts with local owners and

limitations in available memory, without leaving free memory resources go unutilized.

In the context of grid computing [9], research efforts are judiciously concentrated in the numerous challenging problems of programming, partitioning and scheduling computations to run on heterogeneous systems over heterogeneous networks. Although the idea of harnessing as much as possible the shared resources available of the Internet is dominating grid computing, most of the efforts concentrate on discovering, negotiating and scheduling those resources, rather than micromanaging grid programs on a specific resource, once this resource is granted to the program. The Active Harmony project [16, 17] is a notable exception. The project has investigated changes that need to be implemented in the operating system to enable an efficient symbiosis between grid programs and local programs on hosts available for grid computing. We differentiate from this work in two ways. First, we propose a user-level solution designed for portability, while the researchers in Active Harmony proposed solutions implemented in the OS. Second, we are proposing a solution which is more application-centric, that is, it can be customized to the memory reference patterns of specific applications with additional programming effort.

3 Malleable Memory Mapping

The rationale behind user-level malleable memory is to provide a dynamic memory allocation and deallocation scheme which runs at user-level, is portable, and allows the program to run efficiently under changing execution conditions. Memory malleability is implemented in a runtime system that sits between the program and the OS. The program is assumed to be a task of a *grid job*, submitted remotely to harness idle resources. By definition, the grid job executes at a lower priority[1] than any local job in the system.

The runtime system biases the OS VM management policy in two ways: If the amount of free physical memory in the system falls below a thrashing threshold, the grid program forces immediate deallocation of sufficient memory, rather than dynamic reclaiming of memory from the operating system. The working assumption here is that since the program is a guest program submitted over the grid, it should not thrash the system. Conversely, if more physical memory becomes available to the program at runtime and the program has already released memory for reducing memory pressure, the program can try to reclaim as much of the released memory as possible, or needed, rather than waiting for the VM algorithm to redistribute memory among programs.

Malleable memory mapping is encapsulated in a dynamic shared object. The task of the runtime system is to intercept the program's memory allocations and redirect the anonymous memory mappings that are requested by the operating system to

[1] In the context, the term *priority* is used broadly. It signifies the fact that a grid job should not actually use resources when they are needed by local jobs.

named memory mappings which are controlled by the application. Named memory mappings are backed up by application-defined files in the disk and their consistency is maintained at user-level, by flushing updates to in-core memory-mapped regions before any attempt to unmap pages.

The two critical issues that need to be explained in a malleable memory system is how the runtime system deallocates and allocates memory back to the program. The general strategy is to design memory allocation and deallocation for adaptability to memory shortage and fast reclamation of previously released memory, if the execution conditions permit so. The runtime system provides an automatic mechanism which detects memory shortage at runtime and deallocates "enough" program memory to alleviate memory shortage. There are four technical issues that need to be addressed. The first is when to deallocate memory, the second is how much memory to deallocate, the third is what part of the memory to deallocate and the fourth is how to ensure that the program keeps running correctly despite the deallocation. Symmetric issues occur with memory reallocation. We elaborate on these issues in sections 3.2 and 3.3. Before discussing policies, we provide a brief description of the logistics and mechanisms used in the runtime system.

3.1 Basic Mechanisms

The fundamental mechanism for memory malleability is dynamic mapping and unmapping of allocated memory, at user-level. For memory shrinking, the runtime system maintains the memory maps established for program data and selectively or randomly unmaps regions of these maps. All regions are backed up by designated files in secondary storage. Unmapping makes sure that a program makes a fraction of its allocated physical memory immediately available to the OS. Note that this differs from just freeing memory (e.g. with a call to `free()`), which invalidates a region of the address space of the program but does not necessarily return the memory to the OS for immediate reallocation. If the program faults on a page which was previously unmapped by the runtime system, the runtime system redirects the fault to a user-level handler, which remaps all or part of the previously unmapped region, if and only if the remapping is valid. Segmentation faults outside the regions controlled by the runtime system are released to the OS for handling. The same action is taken when the runtime system decides to reclaim memory on behalf of the program, if sufficient memory becomes available. Protection and access rights of mapped regions are also controlled by the runtime system, via the `mmap` and `mprotect` system calls.

The runtime system maintains a mapping table which contains the user-level memory-mapped regions of the program's data space. The table is initialized with one entry per mapped region created with `mmap`. As the program executes, an unmapping of a previously mapped region is reflected on the table in the following ways: The still-mapped region is maintained in the table with its new bounds and size. The unmapped portion of a previously mapped region is also maintained in a separate

```
while (1) {
  obtain a load index L from the /proc filesystem;
  obtain resident_size, system_memory_size;
  if (resident_size > system_memory_size/L) {
      release (resident_size - system_memory_size/L);
  }
}
```

Figure 1: Memory deallocation heuristic.

entry in the table with its bounds and size. The reason for this is that this region is "valid,, from the application's point of view, but is temporarily invalidated and unmapped to cope with memory pressure.

The entries in the table contain a recency bit which is used for unmapped pages or contiguous sets of pages. When the runtime system decides to remap a previously unmapped region, the bit is set to indicate recent access. The recency bit used for remapped regions is exploited as an indication of the working set of the program at user-level. This is a lazy evaluation of the working set, in the sense that we only designate regions as parts of the working set, only if these regions get unmapped and remapped later by the runtime system. We are using a simple second-chance algorithm for unmapping regions based on the recency bit.

The map table is maintained as a forest with the initially mapped regions at the tree roots and decompositions of the initially mapped regions at the lower levels. Splitting and coalescing of contiguous regions is handled by rearranging the tree. Using more space or time-efficient data structures for the memory map is a subject of further investigation.

3.2 Shrinking Memory

The runtime system deallocates memory when the memory system is about to thrash. In order to check this, the runtime system polls periodically the `/proc`[2] filesystem and checks how much free memory is available and what is the instantaneous load. The polling period for experimentation is set to one second, but it is a tunable parameter. The condition for shrinking the memory of the program is the following: if free memory is lower than a system-specific threshold, shrink the memory of the grid program down to a *fair share*.

We decide to use program-independent metrics for deriving the *fair-share* of a guest program in memory. The current heuristic is shown in Figure 1. The heuristic is biased towards keeping small programs in memory and reducing the resident set of large programs. We consider as large programs the programs with resident sizes that

[2]So far, we have only experimented with UNIX systems, hence the dependence on the `/proc` interface. Similar ideas can be applied for other operating systems though given the respective interfaces.

exceed their *proportional memory share*, i.e. the memory size divided by the load of the system, given by the parameter `L`. The latter is an approximation of the ready queue size using the length of the run queue during the past ten seconds. This is similar to the value reported by `uptime`.

Note that the heuristic uses information available locally to each program. It does not use centralized information on the sizes of other programs and does not assume any kind of synchronization of the checks made by different programs. It is designed for simplicity and portability. Application knowledge could be passed to the runtime system and improve the heuristic in a non-intrusive manner, e.g. by indicating that the program can actually use less than the fair share of memory.

Deciding what part of a mapped region to deallocate is a tougher problem. With perfect knowledge of the application access pattern and the timing of memory references, one could compose an ideal deallocation scheme which always deallocates the regions which will be accessed as far as ahead in the future as possible, or will not be accessed at all. One solution to this would be to use program traces, however the traces should describe both where and when memory is referenced and they should also be independent of the input. Moreover, analyzing memory reference traces at runtime could be prohibitively expensive.

We implemented a scheme that starts with round-robin deallocation and progressively adapts the deallocation to the observed memory reference pattern. Round-robin is a reasonable starting solution for sequential access patterns. Initially, if a deallocation decision has to be made, the runtime system simply deallocates memory proportionally from the beginning of mapped memory regions. Subsequent deallocations, if needed, are satisfied at each memory region from the point where the last deallocation stopped. If there are N memory regions with sizes (in pages) $S_i, i = 1 \dots N$, and the program needs to deallocate M pages in total, the runtime system deallocates $\frac{S_i}{\sum_{i=1}^{N} S_i} M$ pages from each region. This blind decision is refined at later stages of the algorithm. If deallocated regions get reallocated (with the scheme described in Section 3.3), their recency bits are set. A region with the recency bit set is not considered immediately for deallocation and gets a second chance. It is deallocated without a second chance only if the runtime system can not find enough memory with cleared recency bits to deallocate. This algorithm can get as elaborate as a low-level OS algorithm for reclaiming pages, however we prefer to keep it simple, since it has a non-negligible runtime cost and the runtime system already consumes resources needed by user programs.

3.3 Staying in the Memory Band and Expanding Memory

As long as the execution conditions of the program do not change, the runtime system tries to keep the program running in the given memory band. More specifically, if reallocating the unmapped memory back to the guest program will bring the amount of free memory below the critical threshold, the guest program keeps executing with

its restricted resident set. If the program faults on deallocated pages, those pages will be remapped in place of already mapped pages, which are in turn unmapped using the scheme described in Section 3.2. Pages get unmapped round-robin, unless their recency bits indicate otherwise.

We have implemented a lazy memory reallocation strategy and used an adaptive prefetching scheme to accelerate the mapping of contiguous pages upon faults, to amortize the cost of memory reallocation. Lazy reallocation amounts to postponing the unmapping of previously mapped pages, until the program needs to access these pages again. The motivation for prefetching is that if the deallocation algorithm has unfortunately deallocated a significant part of the program's working set, the reallocation of this part should be accelerated.

Prefetching is facilitated with a simple predictor, similar to the adaptive predictors used for data prefetching in microprocessor architectures [6, 10]. We are using a small (32-entry) stream prediction table in memory. The table is indexed with page addresses and the entries are managed with a clock algorithm. Each entry in the table corresponds to a reference stream. The entry contains the last page accessed in the stream, a test page, and a prefetching degree. Assume for simplicity that pages are numbered sequentially. If a page p is faulted and there is no entry in the table indexed with p, either as a last-accessed page or a test page, p is inserted in the table, the test page in the same entry is set to $p+1$ and the prefetch distance is set to 1. If the next fault in the reference is to page $p+1$, page $p+2$ is prefetched and the prefetch degree is increased to 2 and so on. The intuition here is to progressively increase the prefetch degree if a persistent sequential fault pattern is observed. Prefetch distances and degrees are reset upon interruptions of contiguous reference streams.

The same mechanism is used when the runtime system decides to re-expand the resident set of the program, with the only difference being that pages are not remapped in place of already mapped pages. The expansion decision is taken by reversing the criterion for the shrinking decision, i.e. if mapping back all the mapped regions of the program does not overcommit memory. Nevertheless, expansion of the program is done with lazy remapping and prefetching, as described previously. This means that we do not immediately give the memory back to the program, but we do it gradually, so that the program reloads only needed data which are not mapped in memory. Lazy reallocation is also a defensive mechanism to shield the guest program and the system from instantaneous spikes of memory load.

3.4 Possible Extensions

The memory malleability techniques described so far are designed for simplicity, low overhead and portability. We currently have a malleable memory mapping scheme which is good for mostly sequential memory reference patterns. Clearly, shrinking and expanding the resident set at runtime can be improved with application hints, compiler support, or by observing the application's memory reference pattern.

The application can provide hints for dead or live memory regions, both in space and in time, by specifying the limits of used/unused data regions and the code boundaries within which these data regions remain used or unused respectively. Compiler support can provide similar information, although this is usually possible only for regular array-based scientific codes. The compiler can reduce the programming effort required to develop a malleable program. Another interesting issue is to investigate if the runtime system can co-operate with garbage collectors or use ideas from garbage collection to improve performance.

Analyzing the memory reference stream of the program at runtime is more challenging. Our current system performs limited analysis of the access stream for improving the selectiveness of memory unmapping and the effectiveness of prefetching. Obviously, the more the information about the reference stream made available to the runtime system, the better job the runtime system can do while unmapping and remapping memory. Unfortunately, maintaining excessive amount of information on the memory reference stream at runtime can be very expensive. At best, the runtime system can collect as much information as the operating system itself collects for allocating and reclaiming memory, albeit limited to the local scope of a program.

Finally, for full portability, it would be desirable to have a runtime system that embeds memory malleability directly to object code. One solution we are investigating is to obtain the memory mappings of a running program from the `/proc` interface and manage them with the runtime system. Managing dynamically allocated heap space is another open issue.

4 Evaluation

We ran experiments on two platforms: a cluster of four Dell servers, each with two Intel Xeon processors running at 1.4 Ghz and 1 Gigabyte of RAM per node; and a 4-node cluster of an SGI Origin2000, with two MIPS R10K processors per node running at 250 MHz and 768 Megabytes of memory per node.

We setup the following synthesized experiments. We run a pseudo-distributed application, which consists of identical copies of matrix-matrix multiplications and a reduction performed at the end of the multiplications, using MPI. Together with the distributed matrix multiplications, we run a script on each node of the cluster. The script offers two types of memory load, shown in Figure 2. The left chart shows a contiguous offering of memory load and the right chart shows a memory load modeled with a step function. In the first case (contiguous memory load) we commit 75% of the memory available to a node, by running repeatedly a program that touches random pages in a resident, memory-mapped array. The program completes after touching the entire set of pages in its address space and then runs again. In the second case, the offered memory follows a step function. We commit a time-variant fraction f_t of system's memory and touch random pages in it. f_t is given by:

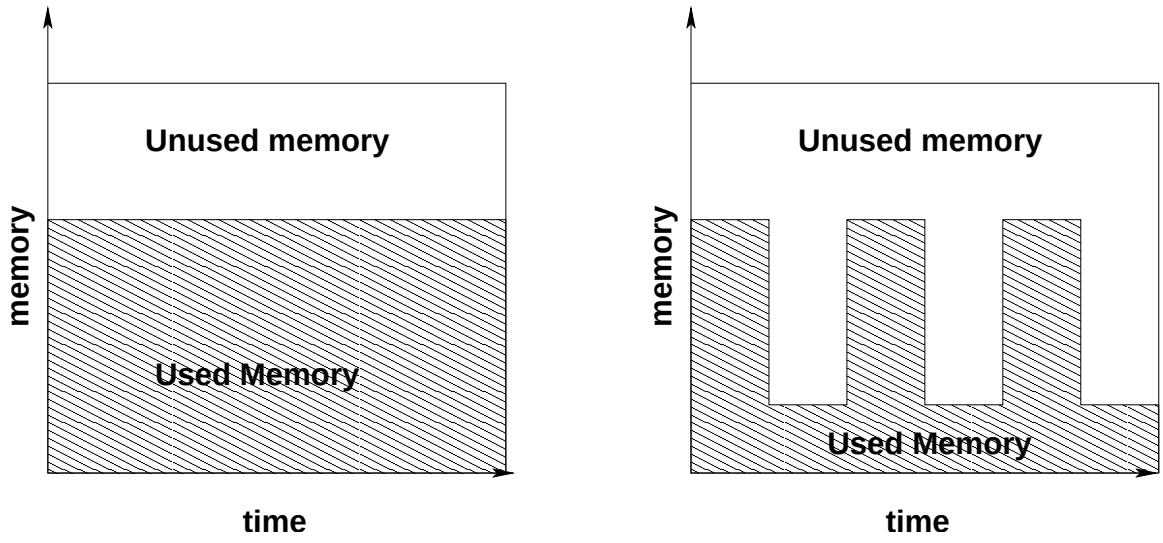


Figure 2: Types of memory load used in the experiments.

$$f_t = \begin{cases} 75\% & t, even \\ 25\% & t, odd \end{cases} \quad (1)$$

where t denotes a time interval the length of which is user-defined.

We run the synthesized distributed application with different matrix sizes, to produce a resident set which ranges between 60% and 100% of the memory available on each node. This translates to matrix sizes of 200–340 Mbytes on the Linux cluster and 150–250 Mbytes on the Origin. Distributed matrix multiplications are fed back-to-back to the clusters in a closed system setting. We measure the normalized throughput of matrix multiplications at different degrees of memory use, ranging from 135% to 175% of the available node memory. The normalized throughput is calculated by inverting the average execution time of 100 consecutive instances of the benchmark on the loaded system and multiplying it with the average execution time of 100 standalone executions of the same benchmark. A throughput of 1 implies that the benchmark suffers no slowdown due to memory contention or thrashing. The thresholds for memory load were chosen arbitrarily. We only verified experimentally that offering a memory load that equals or exceeds 135% of the available memory caused thrashing on the two systems we tested. We also believe that memory overcommitment at a factor of 2 or less is quite realistic in production settings and higher memory loads are not frequently encountered in practice.

The results are preliminary and should be interpreted as such. We are using a program with long streams of sequential accesses and problem sizes selected solely to control the memory load, rather than representing a realistic application. We evaluate the simplest case of adaptability, i.e. adapt the program to a memory space that stays fixed throughout execution and a form of periodic adaptability, i.e. adapt the program

to a memory space that fluctuates periodically, with varying period lengths.

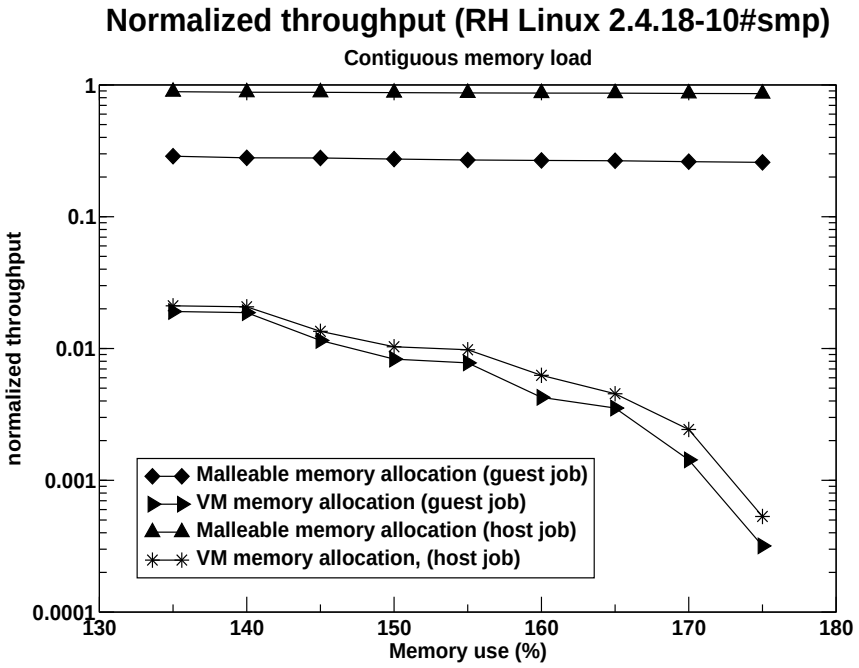
Figure 3 shows the normalized throughput of malleable memory allocation and the Linux (left) and IRIX (right) VM systems with a contiguous offer of memory load at 75% memory use. Each matrix multiplication requests between 60% and 100% of the node memory. The throughput of the malleable memory allocator starts at 0.28 and drops gradually to approximately 0.25 in Linux and 0.24 in IRIX. The throughput of the Linux VM system starts at 0.02 and drops rapidly to 0.0003 at 175% memory utilization. There is an order of magnitude of difference between the malleable memory allocator and the VM system at 135% memory utilization. This grows to 3 orders of magnitude at 175% memory utilization due to the effects of thrashing. There is a noticeable improvement of throughput with the IRIX VM allocator (2-fold to 10-fold improvements over the Linux VM system) but the overall results do not change significantly. We observed long temporary program suspensions in IRIX and we suspect that the improvement is attributed to these suspensions, but we did not investigate the behavior of the IRIX VM policy further, since this was beyond our intentions.

Figure 3 shows also the throughput of the host application, which in this case is the synthetic benchmark that touches pages of its address space in random order. As expected, if the guest job runs within its memory band (25% of memory), there is no significant impact on the host job, other than sharing system resources such as the bus (note that the two jobs run on different processors). The throughput of the host job ranges between 0.87 and 0.93 when the malleable memory allocator is used for the guest job. On the contrary, the VM systems of both IRIX and Linux favor only marginally the host job, which suffers from thrashing practically as much as the guest job.

The almost horizontal slopes of the normalized throughput of the malleable memory allocator for both host and guest jobs are an encouraging sign. Ideally, we would like our allocator to keep the part of the working set of matrix multiplication that occupies exactly 25% of the available node memory stable, throughout the execution of matrix multiplications. The observed behavior matches this expectation.

The experiments with the periodic memory load were conducted to test whether the malleable memory allocator can exploit idle memory intervals and investigate how long should these intervals be to provide meaningful performance improvements to guest jobs. We conducted experiments with three intervals set to 5, 10 and 20 seconds. All three intervals are shorter than the length of any individual matrix multiplication in stand-alone mode. Setting intervals shorter than the execution time of the test program indicates whether the memory allocator provides runtime adaptability. We are primarily interested in checking if the program can take advantage of additional memory while it runs and not before it starts running.

From Figure 4, we observe that our memory allocator does not seem to be particularly responsive to 5 and 10-second intervals of idle memory in Linux. However, there is a significant improvement in throughput with 20-second idle memory intervals. In



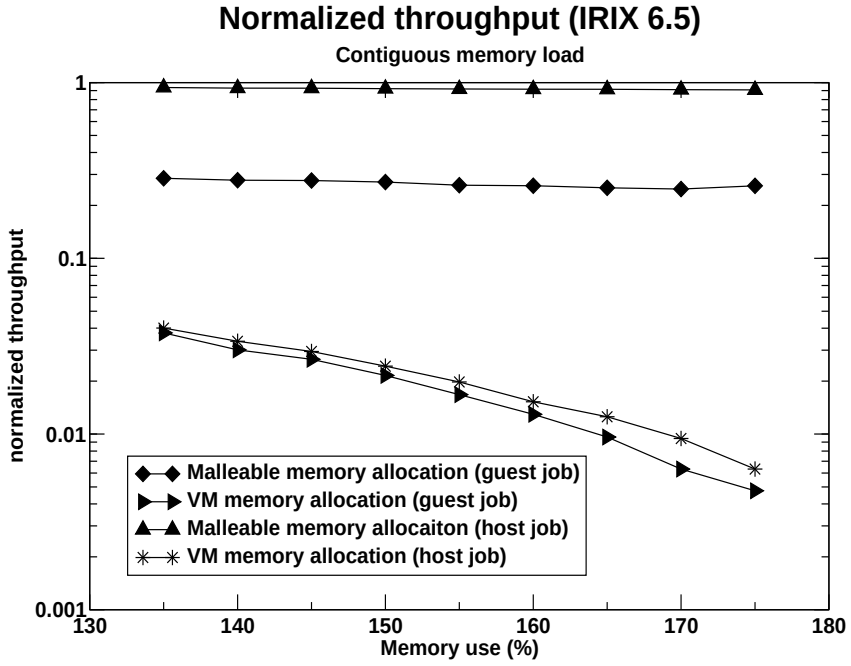


Figure 3: Normalized throughput of the malleable memory allocator and the standard VM allocator of Linux (left) and (right), with a contiguous memory load.

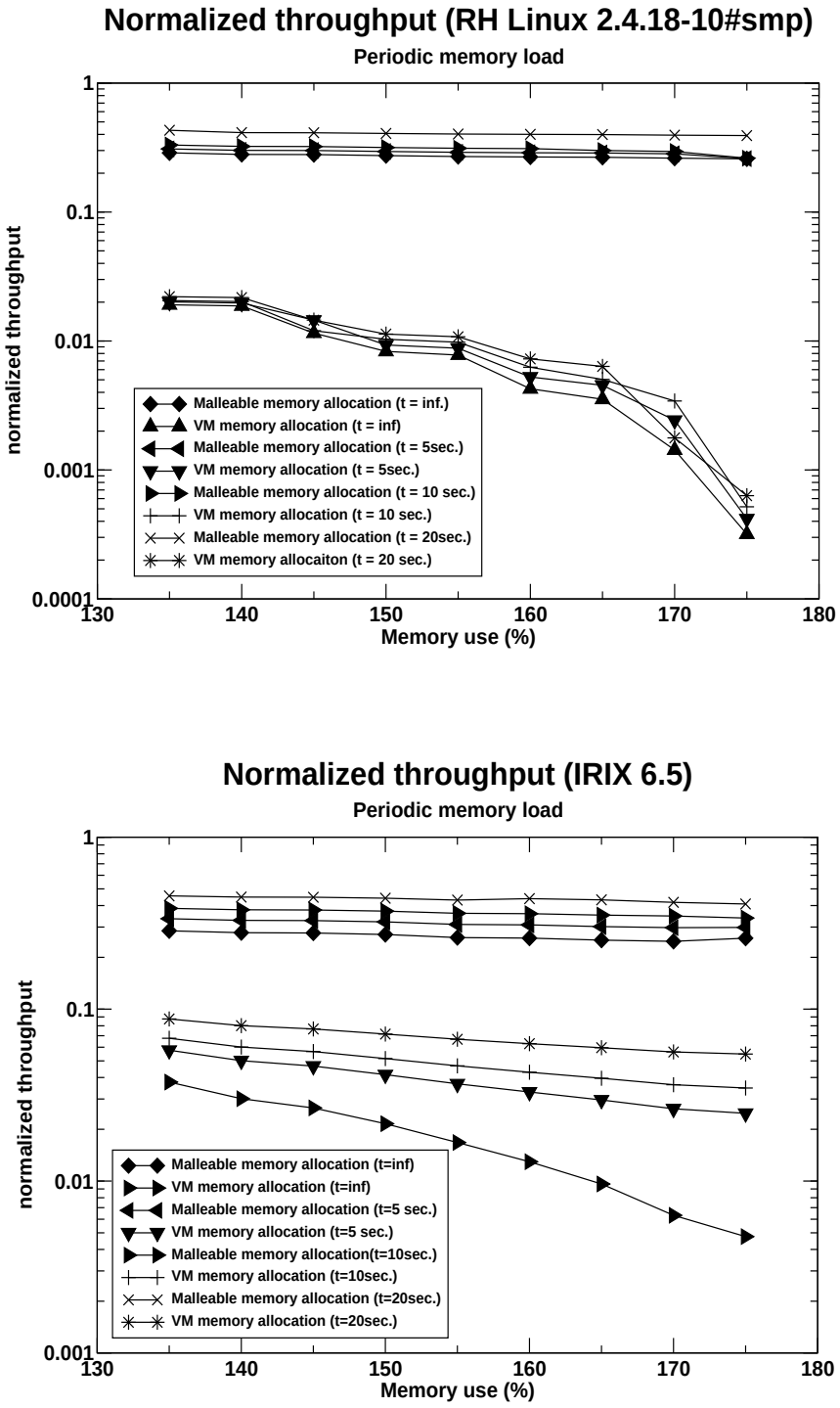


Figure 4: Normalized throughput of the malleable memory allocator and the standard VM allocator of Linux (left) and IRIX (right), using a step function for memory load.

IRIX, both the malleable memory allocator and the kernel exhibit similar behavior, with roughly constant rate of throughput improvement when increasing the length of the interval. We notice that the IRIX VM system benefits significantly from the additional memory space, which is not the case for the Linux VM system. Again, we do not analyze this behavior further, as our intention is not to improve existing VM systems, but build runtime support for performance portability on top of them. The message from the results is that the malleable memory allocator can exploit coarse-grain idle intervals at runtime. Naturally, more experiments with different functions for memory load are required to draw more accurate conclusions.

5 Conclusions and Future Work

We have presented a malleable memory mapping scheme which aims at enabling effective adaptation of jobs submitted to harness idle memory and CPU cycles in non-dedicated, remotely owned systems. We have proposed malleable memory mapping as an alternative to coarse-grain solutions for running these jobs without thrashing the system and without claiming additional memory from local jobs. We have argued that this scheme is more portable than schemes based on modifications to the OS and evaluated its effectiveness with controlled experiments on two different operating systems. We have presented preliminary results which have shown that guest jobs can sustain reasonable throughput even if only a small fraction of the memory needed by them is available on the system. Furthermore, we have shown that given changes of memory load that happen in coarse-grain time intervals malleable memory mapping can benefit from additional idle memory and improve throughput.

The presented work is a first step towards implementing runtime support for adaptive programs submitted for execution over computational grids. Several directions of further investigation were already pin-pointed in the paper, such as exploiting application-specific knowledge for improved memory management and using compiler support. We plan to investigate these issues in detail. We have already presented at least one significant open problem, which is eliminating the need for source code modifications. Ways around this problem, i.e. using a malleable memory mapper which operates directly on the program's binary need to be investigated as well.

Acknowledgments

This work was partially supported by the NSF ITR Grant No. 0085917. Part of this work was carried out while the first author was with the Coordinated Science Lab, at the University of Illinois, Urbana-Champaign. The author would like to thank Constantine Polychronopoulos for several contributions to this work.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proc. of the 1997 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*, pages 225–236, Seattle, Washington, June 1997.

[2] A. Acharya and S. Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proc. of the 1999 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, pages 35–46, Atlanta, Georgia, May 1999.

[3] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data Parallel Programs. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, pages 1–10, Santa Barbara, California, July 1995.

[4] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[5] T. Cormen, J. Wegmann, and D. Nicol. Multiprocessor Out-of-Core FFTs with Distributed Memory and Parallel Disks. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'97)*, pages 68–78, San Jose, CA, November 1997.

[6] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 International Conference on Parallel Processing (ICPP'93)*, volume 1, pages 56–63, August 1993.

[7] J. Dongarra, S. Hammarling, and D. Walker. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1–2):49–70, April 1997.

[8] D. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, August 1997.

[9] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, July 1998.

[10] E. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, 1995.

[11] K. Harty and D. Cheriton. Application-controlled Physical Memory Using External Page-Cache Management. In *Proceedings of the 5th International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS'V), pages 187–197, Boston, Massachusetts, October 1993.

[12] S. Jiang and X. Zhang. TPF: A System Thrashing Protection Facility. *Software: Practice and Experience*, 32(3):295–318, 2002.

[13] Z. Li, J. Reif, and S. Gupta. Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms Using Block-Cyclic Data Distributions. *IEEE Transactions on Parallel and Distributed Systems*, 10:297–315, March 1999.

[14] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[15] E. Rothberg and R. Schreiber. Efficient Methods for Out-of-Core Sparse Cholesky Factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, January 2000.

[16] K. Ryu, J. Hollingsworth, and P. Keleher. Mechanisms and Policies for Supporting Fine-Grain Cycle Stealing. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS'99)*, pages 93–100, Rhodes, Greece, June 1999.

[17] K. Ryu, J. Hollingsworth, and P. Keleher. Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing. In *Proc. of the ACM/IEEE Supercomputing'2001: High Performance Networking and Computing Conference (SC'2001)*, Denver, Colorado, November 2001.

[18] S. Sechrest and Y. Park. User-Level Physical Memory Management for Mach. In *The Second USENIX Mach Symposium Conference Proceedings*, pages 189–200, Berkeley, CA, November 1991.