

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 23187-8795
dsn@cs.wm.edu

Abstract

This paper presents a user-level runtime system which provides memory malleability to programs running on non-dedicated computational nodes. Memory malleability is analogous to processor malleability in the physical memory space. It lets a program shrink and expand its resident set size in response to runtime events, while preserving execution correctness. Malleability becomes relevant in the context of grid computing, where loosely coupled distributed programs run on non-dedicated 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 physical memory of a computational node, without thrashing, and before reverting to coarse-grain load balancing via checkpointing and migration. Malleable memory mapping copes also with the unpredictable behavior of existing virtual memory systems under over-sized memory loads. The current prototype is simple but leaves plenty of room for both application-dependent and application-independent optimizations. The presented results show that user-level malleable memory can improve the throughput of remotely executed programs running on non-dedicated computational nodes by at least an order of magnitude.

1 Introduction

Multiprogramming has been a thorn in the development of efficient programs for non-dedicated computational platforms. Sharing of processors, memory, and network links may nullify 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 effort is placed on developing programming and runtime support for adaptive 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 program can make the most out of the available resources on any given node, at any point of execution.

The problem of sharing physical memory on multiprogrammed systems has received less attention than the problem of sharing other resources, such as processors and network bandwidth. Typically, distributed programs are developed under a simplifying assumption of the underlying memory constraints. One choice is to measure the size of the resident set of the program and examine if the program fits or doesn't fit in the physical memory of a node. If the program doesn't fit in memory, the higher level scheduler opts for another node with higher memory capacity, if such a node is available.

A second choice, which is more difficult to implement, is to restructure the program so that the size of its memory footprint is reduced. The programmer may consider numerous approaches to this problem, including compiler optimizations for memory hierarchies, algorithmic restructuring, or an out-of-core implementation of the program. We are investigating if there is middle ground between the aforementioned choices. This means, a mechanism that lets the programs run at a reasonable speed with less physical memory, without thrashing the hosting nodes and without slowing down the jobs of local users. To increase the benefit for distributed computing, we are looking for a solution which is portable, does not require modifications to the OS and can be customized to the characteristics of the targeted applications.

Relying on existing virtual memory (VM) systems does not appear to be the best option in this context. Most VM systems fail to handle over-sized 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 physical memory allocated to each job in the system in the *long-term*. The notions of *fairness* and *common case* are defined arbitrarily in each VM system. Furthermore, VM systems are hard to understand and writing code for adapting to a specific feature of one VM system makes both code and performance 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 [8]. On the other hand, changing the VM system to enable application-specific VM and physical memory management schemes is the most customizable solution, but requires major changes in the design of the operating system. These changes may not be feasible or desirable for general-purpose operating systems.

1.1 Problem statement

The problem that we are addressing in this paper is the following: How can we provide runtime support to a program running as guest on a non-dedicated computational node, so that the program runs as efficiently as possible without thrashing the memory system, when the physical memory available to the program fluctuates at runtime and the data of the program may not fit in the available physical memory at certain points of execution. We assume that the program is already optimized for a fixed-size memory hierarchy. We also assume that the program runs on top of a typical VM system, with swap space on disk to back up paged out data.

We define memory malleability as the ability to dynamically shrink and expand the resident set of a program in physical memory, with a mechanism controlled by a user-level runtime system, in response to oscillations of the physical memory available to the program by the OS. The concept is analogous to processor malleability, which is the ability of a parallel program to run seamlessly on a dynamically varying number of processors.

Grid computing provides strong motivation for writing programs with malleable memory. Non-dedicated, multi-programmed servers become increasingly popular as components of clusters and computational grids. The application basis for these platforms is expanding towards programs which are more data-intensive and have larger resident sets. Grid computing is offered as an alternative for harnessing the available cycles and memory of widely networked systems [1], but faces the problem of harmonic co-existence of local jobs and grid jobs on privately owned workstations.

1.2 Contribution

In this paper, we present a runtime system which provides memory malleability to programs and discuss its implementation and preliminary performance results. 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 all desktop and server OSes. A user-level mechanism enables also the implementation of application-specific policies for managing physical memory, including application-specific page placement and replacement schemes.

Second, it is almost entirely transparent to the application. The application can use malleable memory mapping by dynamically loading a runtime library that provides wrappers to memory allocation functions. This mechanism 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 with the proposed runtime support.

Third, the mechanism is expandable in many ways. The current implementation of the runtime system unmaps and remaps application memory transparently using application-independent metrics. It does a good job in controlling the resident set of the program and throttling memory consumption when thrashing is eminent. The mechanism 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 from two systems, a small Linux PC cluster with four dual Intel Xeon 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. We also present preliminary results from experiments with actual distributed applications.

The rest of this paper is organized as follows: Section 2 overviews related work. Section 3 presents the design and implementation of malleable memory mapping. Section 4 discusses the experimental setting and results. Section 5 concludes the paper.

2 Related Work

The idea of malleability has been explored in the context of job scheduling on parallel systems [4]. Taking processors away from a program and re-scheduling the program's computation on less processors is relatively easy to implement on shared-memory multiprocessors. It is harder and more computationally expensive on clusters. On the other hand

the impact of taking memory away from a program is hard to assess without full knowledge of the program's memory access pattern. Even if assessing this impact is possible, coming up with efficient methods for user-level adaptation to unpredictable memory shortage is challenging.

A method to cope with memory shortage in applications with large problem sizes is to use out-of-core algorithms. Significant work has been done on providing optimized out-of-core implementations of popular mathematical routines [3, 10] and compiler support for effective composition of out-of-core programs [9]. 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 implement malleable memory mapping in programs that may actually fit in the memory of the target node when the node is idle, but their performance suffers due to contention for memory and undesirable interferences with the scheduler and the VM system with the OS, when the node is loaded.

Application-specific memory management is a thoroughly investigated area of operating systems design and implementation [2, 7]. The similarity between these works and ours is that both are attempting to improve the performance of programs in cases where the VM system of the OS is likely to fail. There is one important difference though. Application-controlled memory management mechanisms are primarily designed to improve the performance of a stand-alone application, when the page replacement algorithm used by the OS does not match the application's data access pattern. In general, these algorithms do not consider multiprogramming and memory sharing, or treat them as orthogonal problems. We are targeting a different problem, which is how to enable effective adaptation of the memory footprints of jobs running as guests on multiprogrammed hosts, without leaving free memory go unutilized.

In the context of grid computing, research efforts are judiciously concentrated in the numerous challenges of programming, partitioning and scheduling computations to run on heterogeneous systems over heterogeneous networks. Although the idea of harnessing the shared resources available on the Internet is dominating grid computing, most of the efforts concentrate on discovering, negotiating and scheduling these resources at a coarse granularity.

The Active Harmony project [11] has proposed changes that need to be made in the operating system to enable symbiosis between grid programs and local programs on hosts available for cycle stealing. In [11], the authors presented a mechanism for limiting the amount of physical memory consumed by guest programs to as much as the memory left over by host programs, using kernel extensions. We differentiate from this work in two aspects. First, we propose a user-level solution designed for better portability. The proposed solution is based on virtual memory mapping, a service available to all operating systems that support virtual

memory with backing storage, including all UNIX flavors, Windows and Mac OS. Second, we are proposing a solution which is more application-centric. User-level memory mapping can be customized to the memory reference patterns of specific applications with some additional programming effort. Assuming that the memory access pattern of the program is easy to analyze and optimize, a user-level memory management mechanism can guarantee that the application adapts and obtains the maximum out of the available physical memory on a privately owned workstation, even if the available memory is less than the memory required to run at full speed.

3 Malleable Memory Mapping

The objective of 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 varying execution conditions. We have implemented a malleable memory mapping system in a dynamically linked runtime library.

The malleable memory mapping system biases the OS VM system in two ways: If the amount of available physical memory falls below a thrashing threshold, the program forces immediate deallocation of memory, to avoid the reclamation of pages by the operating system. Conversely, if more physical memory becomes available to the program at runtime and the program has already released memory to reduce memory pressure, the program tries to reclaim as much of the released memory as possible, or needed, rather than waiting for the VM system to reallocate the released physical memory.

The runtime system intercepts the program's memory allocations and redirects the anonymous memory mappings which are requested by the operating system to named memory mappings which are controlled by the application. Named memory mappings are backed up by application-defined files on 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 most important requirement that needs to be met by a malleable memory system is timely memory deallocation and reallocation at user-level. The runtime system provides an automatic mechanism which detects physical memory shortage at runtime and deallocates enough memory to alleviate memory pressure. There are four technical details that need to be addressed in this context. The first is when to deallocate memory, the second is how much memory to deallocate, the third is what part of the address space should be deallocated 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. Be-

fore discussing them, 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 parts of the program's address space at user-level. To shrink the memory footprint of the program, the runtime system maintains a list with the memory mapped regions allocated for program data and selectively unmaps parts of these regions. The runtime system makes sure that a program makes a fraction of its allocated physical memory immediately available to the OS, by directly unmapping contiguous sets of pages. Note that this differs from simply deallocating pointers (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 takes a fault 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. 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, when sufficient memory becomes available. Protection and access rights of mapped regions are also controlled by the runtime system.

Both mapped and unmapped memory regions are maintained in a list, as sets of contiguous pages of the program's address space. Over the course of execution, regions may be split or coalesced, according to the execution conditions and the policy used by the program to release and reclaim memory. Each region maintained in the list contains a contiguous set of pages and is marked as valid or invalid, depending on whether it is mapped to physical memory or not, at a specific point of a execution. A recency bit is also associated with each region. When the runtime system decides to remap a previously unmapped region (or a part of it), the bit is set to indicate recent access. Recency bits are used as indications of the working set of the program at user-level. The data structure used to maintain information for memory regions is a simple linked list, which is converted to an AVL tree, when the number of disjoint memory regions exceeds a certain threshold. This implementation mimics the implementation of memory regions within address spaces in most UNIX-based operating systems.

3.2 Shrinking Memory

The runtime system deallocates memory when the memory system is about to thrash. To detect thrashing, the run-

time system polls periodically the `/proc`¹ filesystem and checks how much free memory is available in the system and what is the instantaneous load. The polling period is a tunable parameter. Our experiments have shown that a period of one second is sufficient to provide timely and accurate information about the execution conditions. The condition for shrinking the memory of the program is the following: if the amount of free physical memory is lower than a system-specific threshold, the address space of the program is shrunk to the program's *fair share* of physical memory.

The amount of physical memory released by the program to shrink its resident set size to the program's fair share is calculated as:

$$r - \frac{M}{L}$$

where r is the current size of the resident set of the program, M is the total amount of physical memory available in the system, and L is the current load of the system. Memory is released only if the size of the resident set of the program exceeds its fair share (i.e. $r > \frac{M}{L}$). The heuristic is biased towards keeping small programs in memory and reducing the resident set size of large programs. We consider as large, programs with footprints that exceed their *proportional share* of physical memory, i.e. the memory size divided by the load of the system. The latter is an approximation of the size of the ready queue, weighted over time to capture the impact of arrivals and departures of jobs.

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 is passed to the runtime system and improves the heuristic in a non-intrusive manner. For example, an application can hint the runtime system that the program can actually use less than the fair share of memory and reduce memory consumption.

Deciding what part of a mapped region to deallocate is a tougher problem. With perfect knowledge of the application access pattern and the timestamps of memory references, one could compose an ideal algorithm which deallocates the pages that will be accessed farthest ahead in the future. Unfortunately, this solution is unrealistic. The next solution, which would be an algorithm that approximates LRU using reference bits, is also difficult to implement, primarily because at user-level the runtime system does not have the ability to access page tables and the information therein. It would be possible to maintain a separate page table at user-level, but this would introduce a number of problems,

¹So far, we have only experimented with UNIX systems, hence the use of the `/proc` interface. Nevertheless, the mechanism is portable to other operating systems, through the respective interfaces.

including excessive memory consumption by the program, delays in the service times of page faults and inaccuracy in the maintained information, since the runtime system can not detect easily events like page faults and page replacements by the VM system.

We implemented a simple scheme that starts with round-robin deallocation and progressively adapts the deallocation to the observed reallocation pattern. Round-robin is a reasonable starting point for sequential access patterns. Initially, if a deallocation decision has to be made, the runtime system 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.

The blindfolded deallocation 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. We prefer to keep it simple, since it has a non-negligible runtime cost and the runtime system already consumes some of the resources needed by the program.

We have also implemented an interface that lets the application pass a hint to the runtime system, which indicates the access pattern of a given memory region. This hint allows the implementation of application-specific memory unmapping and remapping policies. We have implemented application-specific schemes for sequential and strided access patterns. The relevant details are omitted due to space limitations.

3.3 Staying on 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 on the given memory band, without trashing the system. More specifically, if reallocating the unmapped memory back to the program will bring the amount of free physical memory below the critical threshold, the guest program keeps executing by maintaining a reduced, constant-size resident set. If the program faults on deallocated pages, these pages get remapped in place of already mapped pages, which are in turn unmapped using either an application-hinted replacement policy, or the generic scheme described in Section 3.2.

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 released a significant part of the program's working set, the reallocation of this part should be accelerated.

Prefetching is implemented with a simple adaptive predictor, similar to the adaptive predictors used for data prefetching in microprocessors [6]. We are using a small (32-entry) stream prediction table in memory and adapt the number of prefetched blocks based on the observed pattern of remapped pages. The prefetching mechanism does not expand the size of the resident set of the program beyond the limit set by the runtime system.

The same mechanism is used when the runtime system decides to re-expand the resident set of the program, the only difference being that pages are not re-mapped in place of already mapped pages. The expansion decision is taken by reversing the criterion for shrinking, i.e. the runtime system checks if mapping back an unmapped region of the program does not overcommit physical memory. Expansion of the resident set is done with lazy remapping and prefetching, as described previously. This means that the runtime system gives the memory back to the program gradually, rather than immediately, so that the program reloads only required data. Lazy reallocation is a defensive mechanism that shields the program and the system from instantaneous spikes of memory load.

The memory malleability techniques described so far are designed for simplicity, low overhead and portability. We currently have a malleable memory system which handles well 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 memory reference pattern at runtime. These issues are investigated in ongoing work.

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 partition of an SGI Origin2000, with two MIPS R10K processors 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. The first type represents a contiguous memory load, while the second type represents a time-

variant memory load. In the first case (contiguous memory load) we commit 75% of the physical memory available on a node by running repeatedly a program that keeps all data resident in memory. The program touches its pages in a pseudo-random pattern and completes after touching each page in the entire address space at least once. In the second case, the offered memory is modeled with a step function. We commit a time-variant fraction ($f(t)$) of physical memory, given by:

$$f(t) = \begin{cases} 75\% & t, \text{ even} \\ 25\% & t, \text{ odd} \end{cases}$$

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 physical memory available on each node. Distributed matrix multiplications are fed back-to-back to the nodes of the cluster 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.

4.1. Constant Memory Load

Figure 1 shows the normalized throughput of the malleable memory system and the Linux and IRIX VM systems with a contiguous offer of memory load at 75% memory utilization. The throughput of the malleable memory system starts at 0.28 and drops gradually to approximately 0.25 in IRIX and 0.24 in Linux. The throughput of the Linux VM system starts at 0.02 and drops rapidly to 0.0003 at 175% memory utilization. There is a difference of one order of magnitude between the malleable memory system and the VM systems of IRIX and Linux at 135% memory utilization. The difference grows to 3 orders of magnitude at 175% memory utilization, due to thrashing. The IRIX VM system performs significantly better than the Linux VM system. IRIX outperforms Linux by factors of 2–10. We observed temporary program suspensions that alleviate thrashing in IRIX and we speculate that the improved throughput is attributed to these suspensions.

Figure 1 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 physical 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 guest job runs with malleable memory. The VM systems of both IRIX and Linux favor only marginally the host job, which suffers from thrashing almost as much as the guest job.

4.2. Time-Varying Memory Load

The experiments with the time-varying memory load were conducted to test whether the malleable memory system 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. Note that these intervals are much shorter than the length of a single distributed matrix multiplication in stand-alone mode. Using intervals shorter than the execution time of the tested program helps us check whether the runtime system provides the desired runtime adaptability to the program. The experiment verifies if the program can take advantage of additional physical memory made available to it while it is running.

From Figure 2, we observe that the runtime system does not seem to be particularly responsive to 5 and 10-second intervals of idle memory in Linux. However, there is an improvement of throughput (16% on average) with 20-second idle memory intervals. In IRIX, both the malleable memory system and the OS VM systems exhibit similar behavior, with roughly constant rate of throughput improvement every time the length of idle intervals is increased. Throughput increases on average by 52% at the lowest memory utilization levels and 125% at the highest memory utilization levels. We notice that the IRIX VM system benefits significantly from the additional memory space. This is not the case for the Linux VM system. In general, certain idiosyncrasies of the VM system can be inferred and used to improve the management of malleable memory, at the cost of reduced performance portability.

4.3 Preliminary Results with Applications

To obtain a feeling of the effectiveness of using malleable memory in real applications, we experimented with distributed versions of the three applications benchmarks in the NAS benchmark suite, namely BT, SP and LU. The distributed versions of the benchmarks were implemented by following the guidelines for the NAS Grid Benchmarks [5]. During an experiment, each node in the cluster runs an instance of each benchmark. The instances proceed independently and synchronize at the end, by sending their results to a designated master node. Upon completion of the col-

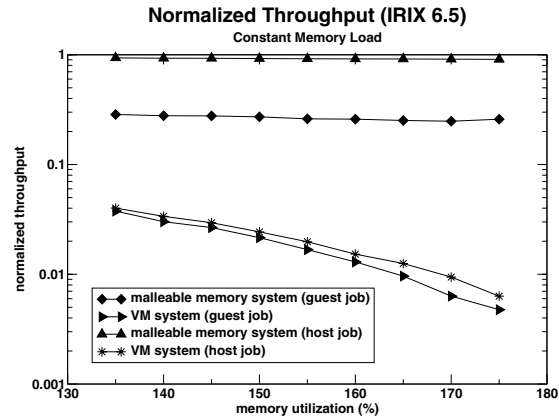
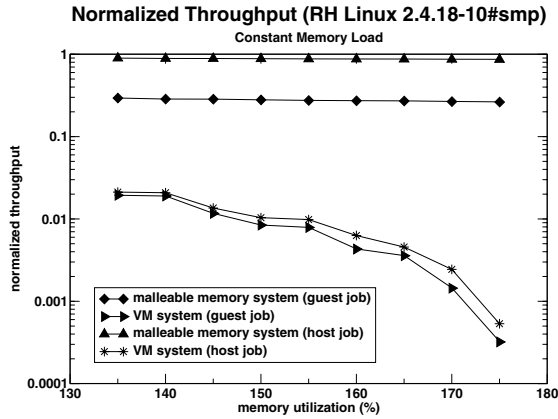


Figure 1. Normalized throughput of jobs with constant memory load.

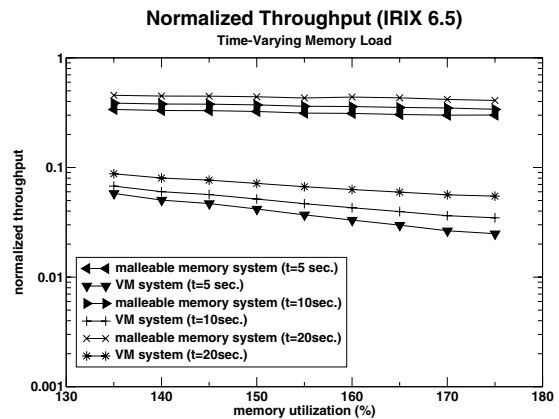
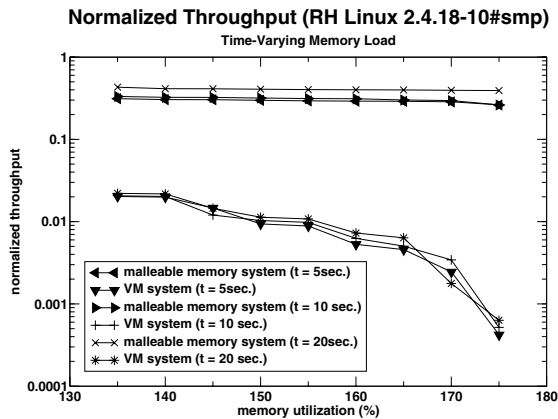


Figure 2. Normalized throughput of jobs with time-varying memory load.

lective operation, a new instance of the same benchmark is forked. This process is repeated 100 times.

During an experiment, together with one instance of a NAS benchmark, each node runs a script which offers a time-varying memory load. The script offers a load equal to 90% of the available memory during even intervals and 0% of the available memory during odd intervals. The length of the intervals is set to 10 seconds. We used the Class B problem sizes of NAS. The benchmarks require 168–197 Megabytes of physical memory per node. The total memory load offered at even time intervals varies between 116% and 121% of the physical memory available on each node. At memory load peaks, the benchmarks maintain only 50%–60% of their data in memory.

Figure 3 shows the normalized throughput achieved with malleable memory mapping and with the VM systems of

the two platforms on which we experimented. Each couple of bars corresponds to one benchmark running on one platform, denoted by the labels in the first and second row under the X axis respectively. The throughput with malleable memory mapping ranges between 0.27 and 0.30. The throughput of the VM systems flattens around 0.03. The results agree remarkably with the results obtained from the experiments with synthetic workloads.

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-

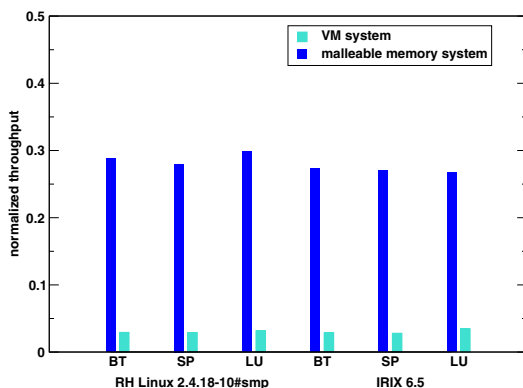


Figure 3. Normalized throughput of application benchmarks with time-varying memory load.

grain solutions for running these jobs without thrashing the system and without claiming additional physical 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, using synthetic benchmarks and actual distributed applications.

Several directions of further investigation were already pin-pointed in the paper, such as exploiting application-specific knowledge to tune the memory management policies, using compiler support, and avoiding source code modifications by plugging memory malleability directly into the binary. More experiments with real applications are required to evaluate the runtime system in tightly coupled and loosely coupled platforms, including heterogeneous systems.

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 and S. Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proc. of the 1999 ACM SIGMETRICS Joint International Confer-*

ence on Measurement and Modeling of Computer Systems (SIGMETRICS'99), pages 35–46, Atlanta, Georgia, May 1999.

- [2] 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.
- [3] 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.
- [4] D. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, August 1997.
- [5] M. Frumkin and R. Van der Wijngaart. NAS Grid Benchmarks: A Tool for Grid Space Exploration. In *Proc. of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC'10)*, pages 315–324, San Francisco, CA, August 2001.
- [6] E. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, 1995.
- [7] 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.
- [8] S. Jiang and X. Zhang. TPF: A System Thrashing Protection Facility. *Software: Practice and Experience*, 32(3):295–318, 2002.
- [9] 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.
- [10] 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.
- [11] 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.