Distributed Shared Memory: Bridging the Granularity Gap

Ayal Itzkovitz*  
Department of Computer Science, CIMS  
New York University  
ayali@cs.nyu.edu

Assaf Schuster  
Computer Science Department  
Technion – Israel Institute of Technology  
assaf@cs.technion.ac.il

Abstract

In this position paper we explore a very recent technique, called MultiView, its applications, and its implications on the design and usage of distributed shared memory systems (DSMs) [6]. MultiView can be used to bridge the gap between the large, fixed-size memory pages handled by the hardware and operating system, and the relatively-small, varying-size minipages that are used by applications. Using MultiView, the distributed shared memory system can adapt to the native granularity of the application in a natural way. While originally proposed for supporting fine-granularity sharing, MultiView can also be used by all the accompanying DSM services, including sharing across machines, protection and consistency manipulation, detecting racing accesses, collecting garbage, tracing true sharing by application threads, etc. Thus, MultiView simplifies the design and usage of DSM systems in a significant step towards making them a popular technology.

1 Introduction

Consider the services that are supported by a distributed shared memory system (DSM), including sharing across machines, protection and consistency manipulation, detecting racing accesses, garbage collection, tracing true sharing by application threads, etc. Clearly, these services use the application native granularity, i.e., all these services manipulate fragments of data that are being used by the application as integral units (e.g., variables, objects, data-structures). Unfortunately, the hardware and operating systems’ memory managers are based inherently on large, fixed-size pages, and do not adapt to the application native granularity.

The basic mechanism proposed in Ivy, the first DSM [10], makes use of the protection mechanism which is provided per memory page by a cooperation of the hardware and the operating system. The page size is determined by the hardware, and is currently set at the level of a few KB, with a growing tendency to further enlarge it. For DSMs, however, the page size defines the unit of sharing that can be supported using the protection mechanisms. In a distributed cluster, the penalty for sharing in granularity different from that of the application variables and objects is very high, resulting in false sharing, consistency complications, inflated communication volume, and high overhead on accompanying services.

Thus, since the introduction of Ivy, most of the work in the evolution of DSMs attempted to overcome the obstacles put forth by this problem. Despite the large amount of work that was invested and the resulting progress (see for instance [1, 3, 11, 13]), the problem was never completely solved. Because of this reason many still consider DSMs as doomed, and it was claimed that most work in this field is incremental [4].

We present here a simple approach, called MultiView, that can be used to bridge the gap between the native memory elements of the application and the fixed operating system page size. We show that a non conventional memory layout eliminates false sharing, provides efficient support for strict consistency, reduces communication to minimum, and allows natural integration with satellite services, such as garbage collection and data race detection. Indeed, using MultiView results in a substantial simplification of protocols and a corresponding reduction of overhead, thus making DSMs at the same time both simpler and more efficient.

We proceed below to present MultiView. We then draw conclusions from our experience in implementing MultiView in a DSM system called MilliPage. We discuss an avenue of potential developments extending from the basic idea of MultiView. Interestingly, much of the

*Much of this work was done while the author was with the Computer Science Department at the Technion, Israel.
research can be viewed as “Back to the Future”, revisiting previous works that were handicapped by the fixed, large page assumption, repeating these studies under the new model where this is a non-issue.

2 Bridging the Gap: From Pages to Minipages

2.1 MultiView

Consider three variables \(x\), \(y\), and \(z\), whose total size is smaller than that of a page. In a page-based DSM system, the memory for these variables may be allocated on the same page, resulting in potential false sharing. Consider a mapping of the page that contains these variables to three different, non-overlapping, virtual address regions, starting at addresses \(v_1\), \(v_2\), and \(v_3\). Each of the locations in the page can now be viewed via three different virtual addresses. We call each of these regions a view. The application uses different views to access different variables (so variable \(z\) is accessed via \(v_1 + \text{offset}(x)\), \(y\) via \(v_2 + \text{offset}(y)\), and \(z\) via \(v_3 + \text{offset}(z)\)). Since access permission is controlled through the virtual memory mechanism, protection and fault handling for these variables (which reside in the same physical memory page) can now be applied in three different and independent ways, one for each view. It is now possible to manage a separate access privilege for each variable using its respective view. Moreover, an independent consistency protocol can be implemented for each variable.

2.2 Views, Vpages, and Minipages

Most DSM systems provide consistency guarantees for a single, large, contiguous region of shared addresses. In systems which implement MultiView, the shared space is mapped to several such regions of virtual addresses, called views. Consequently, each memory element can be accessed via the virtual memory mapping mechanisms, using any of the views. Since views must be managed in granularity of pages, each view consists of a sequence of virtual memory pages which we call vpages.

MultiView lets the application manage each memory element through a dedicated view, or a vpage, which is said to be associated with this element. Because each memory element (in the example above, a variable) can now be managed independently regardless of its size, we call it a minipage. Minipage sizes vary; they can be as large as the virtual page size or as small as the basic memory addressing unit.

A minipage is identified by the associated vpage number and a pair \(<\text{offset}, \text{length}\>\) which indicates the region inside the vpage where the minipage resides. A protection is controlled for the minipage using the virtual memory mapping mechanisms by manipulating the associated vpage protection. A NoAccess protection indicates a non-present minipage, a ReadOnly protection is set for read copies, and a writable copy gets a ReadWrite protection. Copying minipages between the hosts, invalidating minipage copies, and changing access permissions are all done according to the consistency guarantees and the protocols implementing them.

In addition to the fine granularity capabilities, MultiView enables another useful feature. An additional, separate view is constructed, called the privileged view. The protection of the privileged view is fixed and set to ReadWrite. The DSM server threads may use it at all times to access the memory, and are thus not constrained by the DSM memory protections, as determined by the consistency guarantees imposed on the application views. The privileged view may also be used by zero-copy communication protocols to directly send/receive minipages from/to the user space.

Figure 1 depicts a sample memory layout of a general DSM system that uses MultiView.

3 Simplicity and Efficiency: The Thin Protocol Layer

Overcoming false sharing and the struggle to provide per-variable coherency guarantees led to techniques which trade additional memory resources and computational overhead with higher performance of the DSM. One of the most popular ideas in this direction is the Release Consistency (RC) protocol.

RC systems provide multi-writers consistency, by allowing concurrent writes to variables residing in the same page. During synchronization times, the written locations in the pages have to be identified, and the modifications are synchronized in a consistent way. To this end, the protocol saves copies of pages before the application is allowed to write them (twinning). At synchronization times, the protocol calculates the byte-wise differences between the two local copies and prepares a diff message (differing). The diff message is later sent to other hosts which use these modifications.

Thus, RC is claimed to overcome false sharing thanks to the protocol layer, which, in turn, pays in memory and computing overheads. However, we argue that false
sharing is not really eliminated, but appears in a different custom: Processors are busy in creating twins and applying diffs to sections of pages that contain variables which they do not access at all!

The major benefit of using the RC protocol appears to be the relative resilience of the memory layout to sharing patterns. We believe, however, that DSM systems should consist of thin protocol layers that consume as little memory and computational overhead as possible. Using MultiView, a strict consistency protocol can be implemented in a straightforward way complying to the Single-Writer/Multiple-Reader (SW/MR) semantics.

Consider a system which uses MultiView, where variables are distributed among distinct minipages. When the program contains no data races then no concurrent conflicting accesses (which include writes) are expected. Hence, Sequential Consistency (e.g., through the SW/MR protocol) is perfectly suited to this situation: it guarantees consistent modifications while avoiding protocol-related overheads. Indeed, our implementation, as reported in [6], shows initial performance results that support this claim.

### 3.1 The DSM System Prototype

We have implemented a prototype DSM system called millipage. The main goals in building millipage were experimenting with MultiView and exploring the use of a thin protocol layer.

The programming model in millipage is Sequential Consistency, which is implemented through the SW/MR protocol. Aside from the initial setting of the maximal number of views and the size of the shared-memory, the allocation process is transparent from the programmer’s point of view. The malloc-like API returns a pointer which may point to any of the application views (but never to the privileged one).

The thin protocol layer implements the DSM functionality in a straightforward way. On each host a single millipage process is started, running both the application and server threads. One of the processes is the manager, whose role is to maintain the minipage-table (MPT), containing directory information of minipage copy locations, minipage sizes, and the association of view addresses with their minipages. On an access fault, a request message is sent to the manager, which resolves the minipage boundaries (offset and size), and reroutes the request to the appropriate target host. The design ensures that buffering of requests is only done at the manager, resulting in further simplification of the protocol. Latency hiding is implicitly achieved through multithreading.

### 4 Observations on Using MultiView

Clearly, while there are many advantages in using MultiView, the limitations of this technique and its behavior should be further understood.

#### 4.1 Performance Issues and Limitations

MultiView heavily uses the virtual-to-physical memory translation mechanism. Loosely speaking, MultiView attaches a separate PTE to each individual minipage. We have experimented with a simple application that traverses a very large array of small-size elements, each allocated in a different minipage, while the number of minipages per page varies. Figure 2 shows the results. For less than 32 views (i.e. less than 32 minipages per page) and for a moderate size of memory objects (up to 32MB) there is a minor slowdown of 1-4%, attributed to the high rate of TLB misses. At certain breaking points,
which appear precisely when the L2 cache becomes congested with PTEs (and regardless of the data misses), the slowdowns become significant.

We thus get a tradeoff. For an application with a reasonably local memory access pattern, the advantages of MultiView dominate the negligible overhead coming from the excessive TLB misses. However, when the application attempts to access too many minipages at the same time (more than 128K for our Pentium II having 1/2MB L2 cache), the translation and protection information exhaust the cache, imposing overhead which is too high.

4.2 Locality and Virtual Locality

The principle of locality implies that in many cases nearly allocated minipages are also accessed together. However, these minipages will be accessed through different vpages which belong to different views. Thus, the PTEs corresponding to these vpages do not reside contiguously in memory, resulting in fetching irrelevant PTEs and failing to fetch the relevant ones when a cache miss occurs. We call the locality of access in the page table virtual locality.

As discussed earlier, reducing the virtual locality has a major performance implication when many views are used. One way in which we could improve the virtual locality is to interleave the vpages which belong to different views. Alternatively, the PTEs in the PT itself could be interleaved.

4.3 Overlapping and Composed Views

An obvious extension of MultiView is to overlap views and compositions of views. A minipage can be associated with a set of views, each representing a different consistency or a different sharing granularity [5]. Accessing the same variable through different views invokes different protocols, which in turn, may imply a different behavior.

This can also improve the performance of fine-grain DSM systems which suffer from poor performance during coarse program phases [2]. For example, consider a matrix whose rows are associated with minipages. At certain points in the program, it might be desirable to read the entire matrix, bringing in all the minipages (matrix rows) at once. In order to do so, an additional, large minipage could be defined, which overlaps with all the minipages associated with the matrix rows. When the large minipage is accessed, the DSM system ensures it is brought in, which implicitly means aggregating and prefetching all the sub-minipages. Note that in this example, overlapping minipages associated with matrix columns, diagonals, sub-matrices, etc., can also be used.

5 Relations to Other DSM Services

The discussion above concentrated on the DSM major task, namely, to share memory among hosts. As it turned out, using MultiView may improve other applications and DSM services as well. We mention some of these below.

5.1 Global Memory Systems

Recently, there has been a lot of interest in harnessing remote memories for storing memory pages, thus establishing an additional level in the memory hierarchy. It was shown that using subpages as the basic transfer unit leads to substantial performance boost [8]. Clearly, MultiView can be used for implementing subpages (minipages in MultiView terminology) in global memory systems. Furthermore, prefetching can be implemented, by referencing large, composed minipages. The sub-minipages may be pipelined through the communication layer, so that when the first arrives it can be instantly used while the others are still flowing in.
5.2 Data Race Detection

Programming errors in parallel programs commonly show themselves in unsynchronized accesses to data, called data-races. Unfortunately, the complexity of discovering such bugs is very high, which gave rise to run-time detection techniques. In run-time methods the detection overhead may become very large when applied to memory fragments that are irrelevant to the checked program variables, as is the case in page-based DSMs.

In contrast, using MultiView we can ensure that only the relevant minipage is checked, by manipulating the protection of the associated vpage. Since the minipages access pattern represents the variables access pattern by the application, data race detection is efficiently integrated in the DSM by embedding a detection protocol in the minipage consistency management.

We have recently used MultiView for this purpose and found a dramatic performance improvement. The overhead dropped more than two orders of magnitude, from hundreds percent reported in previous works to only a few percent in our system [7].

5.3 Distributed Garbage Collection

MultiView can help collecting garbage distributively in native application granularity, at the level of a standard DSM service. The main contribution of MultiView here is that marshaling and tracing pointer assignments are easy when different objects are pointed via different vpages. Thus, implementing counting methods, which are best suited for distributed garbage collection, become a lot easier and employ less penalty.

We integrated a distributed garbage collector in our system Millipage using the method developed in [9]. The performance results will be reported soon.

5.4 Tracing True Sharing

As was mentioned for data race detection mechanisms, tracing and collecting information regarding accesses to application-level shared variables becomes a natural service of the DSM. Once again, the basic property is that the minipages access pattern also represents the variables access pattern. Thus, it is relatively simple to implement a light-weight (remote) access logging mechanism, maintaining access histories per variable. This gives rise to all kinds of optimization techniques, which, for instance, may relocate threads and balance the load according to the true-sharing pattern [12].

References