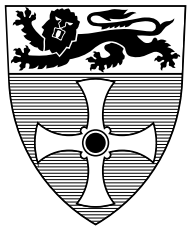
UNIVERSITY OF
NEWCASTLE



COMPUTING SCIENCE

An Object-based Software DSM for the NIP Parallel System

Savas Parastatidis and Paul Watson

TECHNICAL REPORT SERIES

No. CS-TR-678
July, 1999

Contacts: Savas Parastatidis and Paul Watson
Savas.Parastatidis@ieee.org and Paul.Watson@ncl.ac.uk

A version of this Technical Report appears in the proceedings of the 1st Workshop on Software Distributed Shared Memory, Rhodes, Greece, 1999.
(http://www.csrd.uiuc.edu/ics99/ws3p.html)

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

An Object-based Software DSM for the NIP Parallel System

Savas Parastatidis
Parallelism Research Group
Dep. of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK
+44 191 2228789
Savas.Parastatidis@ieee.org

Paul Watson
Parallelism Research Group
Dep. of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK
+44 191 2227653
Paul.Watson@newcastle.ac.uk

ABSTRACT

There has been a recent growth of interest in object-oriented parallel systems. This paper describes an investigation into the design of a specialist software Distributed Shared Memory (DSM) to support such systems. The vehicle for this investigation was the NIP parallel run-time system. NIP was designed to support implicitly parallel languages combining the object-oriented and functional approaches to programming. The NIPDSM is the object-based software DSM component of NIP. The paper shows that it is possible to design a set of optimisations to exploit the characteristics of object-oriented systems in order to deliver higher performance than could be achieved by a conventional DSM design which took no account of the object-oriented nature of the system. In the paper, the design and implementation of the NIPDSM are described, and performance results are presented.

Keywords

Object-based Software Distributed Shared Memory, Parallelism, Networks of Workstations.

1. INTRODUCTION

There has been a recent growth of interest in object-oriented parallel programming. This has raised the question of how to design an efficient Distributed Shared Memory (DSM) component to support such systems. In particular, should a conventional, general purpose, DSM scheme that has not been specifically designed to support object-based systems be adopted? Alternatively, can the object-oriented nature of the parallel systems be exploited in the design of specialised, object-based DSM schemes that can deliver higher performance? This paper addresses the question by describing an investigation into the design of an object-based DSM for a parallel system.

The vehicle for the investigation was the NIP run-time system [1]. NIP was designed to support implicitly parallel languages, such as UFO [2], that combine the object-oriented and functional approaches to programming. It runs on distributed memory parallel platforms, in particular networks of workstations, which do not provide shared memory. Therefore, an essential component of NIP is its software DSM system, the NIPDSM.

The aim of the NIPDSM was to exploit the object-oriented nature of NIP in order to provide higher performance than a general purpose DSM could deliver. An essential feature of the NIP computational model is that it is strictly object-oriented; all data is encapsulated in objects and can only be accessed through method calls. NIPDSM exploits this restricted form of access to data in order to manage caching and consistency more efficiently than it is possible in conventional DSM systems where no such restrictions apply. Whilst NIP was mainly targeted at the UFO language, the NIPDSM is general enough to support a range of other object-oriented languages. For example, current work is investigating its use to support the execution of parallel and distributed Java programs.

The rest of this paper is structured as follows. The next section gives a brief overview of NIP and the requirements that it places on NIPDSM. Section 3 then presents the NIPDSM design and implementation. This is followed by performance results (Section 4) and conclusions (Section 0).

This paper makes the following contributions: it describes the design of a DSM that is optimised for object-oriented systems; it discusses a set of caching optimisations which exploit relationships between objects (rather than just spatial locality); it provides performance results that demonstrate the behaviour of the system.

2. NIPDSM REQUIREMENTS

The NIP parallel runtime system was designed to support implicitly parallel languages that combine the object-oriented and functional approaches to programming. NIP includes a novel lazy task creation scheme which allows parallel tasks to be created only when they are required [3], and a dynamic load-balancing scheme that keeps the parallel system busy by moving work from busy to idle nodes. A full description of NIP can be found in [1] and [3], but is outside the scope of this paper, which focuses only on its DSM component.

In NIP, all data is encapsulated as state within objects and the only way to access the state is to call a method on the object. A parallel task may create objects and call methods on any object in the system, irrespective of the location of the object. To provide high performance, the task must be able to locally compute the result of a method call (the alternative of always executing a method on the node which 'owns' the object was ruled out, as it could create hot-spots). Therefore, the NIPDSM must allow the state of any object to be locally accessible on any node. Before a method is called on a node, the NIPDSM copies the state of the object onto that node. The state is cached so it can be reused.

A key feature of NIP, which greatly influences the design of the NIPDSM, is that objects are locked during method calls. Before a method call, a lock is taken on the object, and it is dropped when the method call is completed. A method that may alter the state of an object (a write method) takes a write lock, while a method that does not (a read method) takes a read lock. Multiple read methods may operate on an object simultaneously, but write methods are serialised. The next section describes how the design of the NIPDSM can exploit the characteristics of NIP described above to deliver high performance.

3. THE NIPDSM

The NIPDSM system is designed to provide an *object view* of the shared memory—the unit of sharing is the object. There are a few similarities between the NIPDSM object space, the *tuple space* of Linda [4], and the data-structured DSM systems like Emerald and Clouds [5, 6] but there are significant differences as well, as will be described.[1]

In this section, an overview of the NIPDSM architecture is presented and the motivation behind the design decisions is explained. Furthermore, it describes the memory consistency model and the caching techniques utilised to improve performance.

In the following, as in the NIPDSM, a distinction is made between objects that can be updated (*mutable objects*) and objects whose state cannot be updated after their construction and initialisation (*immutable objects*).

3.1 The Caching and Consistency Model

In order to reduce access times, NIPDSM allows objects to be replicated across the nodes of the distributed system. The *entry consistency model* [8, 9] was chosen to guarantee that the states of the objects remain consistent as the semantics of this consistency model meet the requirements imposed by NIP (Section 2). However, a variation of the original technique was designed and implemented to deliver higher performance by exploiting the object-oriented nature of NIP.

In NIPDSM, every mutable object is implicitly associated with a lock that is used whenever a method is applied to the object. These locks provide the synchronisation required when the state of the object is accessed. In addition, as will be described (Section 3.3), the same lock is used for consistency-related operations ensuring that the state of the object is always kept up-to-date. In contrast, most conventional DSM designs are less efficient as memory consistency related operations require the use of two locks; one used by consistency-related operations and one for synchronisation-related operations.

3.2 Managing Consistency

When an object is created by the NIPDSM, the local node is designated as its permanent *Manager Node*. The Manager Node maintains information about the object and is responsible for satisfying cache

[1] For a good overview of DSM and an extensive list of existing systems, the reader is referred to [7].

requests from other Nodes. When an object is cached on another node, that node becomes either a *Read Proxy Node* or a *Write Proxy Node* for the object. The following rules maintain consistency:

- The Manager Node of an object allows the local execution of write or read methods on that object, provided there are no proxies for that object.
- The Manager Node of an object allows the local execution of read methods on that object, provided no Write Proxies exist for that object.
- A Write Proxy for an object allows the local execution of write or read methods on that object.
- A Read Proxy for an object allows the local execution of read methods on that object.
- A node can be a Manager Node, a Read or Write Proxy for any number of objects but can never be more than one of these at the same time for a particular object.
- An object always has one Manager Node. At any one time, an object may have none or one Write Proxy, or alternatively, none or any number of Read Proxies.
- More than one task on the same node can execute read methods on the same object at the same time but there cannot be more than one task executing a write method on the same object.

3.3 Implementation

The Manager Node remains the same throughout the lifetime of an object in contrast to Bershad's implementation of the entry consistency model where the Manager Node—in NIPDSM terms—is always the node with write permission to the object. Consequently and unlike the original implementation, NIPDSM does not require a distributed directory scheme. The overhead associated with maintaining the directory is eliminated and the implementation is simplified. Furthermore, optimisations are introduced that reduce the number of messages exchanged (Section 3.5).

Conventional DSM systems use in their implementation the page fault mechanism of the underlying operating system. This usually results in traps to the kernel and the invocation of user-defined handlers. In contrast, NIPDSM avoids using the operating system to detect memory accesses and trap cache misses because it uses the lock operations required by the consistency protocol to detect object accesses (an object can only be accessed via method calls made using a NIPDSM Reference as described in Section 3.4). This is more efficient because these lock operations would still have been required even if NIPDSM were to use the underlying operating system's page-fault mechanism. The other advantage of this approach is that it makes the NIPDSM highly portable: no operating system specific calls are required.

If a method is called on an object that cannot be immediately locked, the calling task suspends and the NIP Load Balancer is informed. If there are no other tasks waiting for the processor, a new task will be created in order to keep the node busy [3].

3.4 NIPDSM Reference and NIPDSM Virtual Object Table

When a NIPDSM object is created, it is assigned a unique object ID, known as the *NIPDSM Reference*. The NIPDSM Reference acts like a virtual memory pointer providing access to the methods of objects and it is used by the NIPDSM system in all locking and caching operations. The NIPDSM Reference consists of: (1) the ID of the Manager Node, (2) a page number, and (3) an offset.[2]

Every node in the parallel system maintains a *NIPDSM Virtual Object Table (NIPDSM VOT)*. A NIPDSM Reference defines a path in the Virtual Object Table that leads to a data structure with information about the object (Figure 1). In order to save memory, space is pre-allocated only for the pointers to the pages with data structures (second column of the VOT in Figure 1) and only for those entries in the Virtual Object Table that would lead to an object created on that node. The pages containing the data structures with information about the objects are allocated lazily. Furthermore, the whole part of the Virtual Object Table that maintains information about the cached objects at a node is constructed lazily as objects arrive.

[2] The current implementation uses 32bit NIPDSM References: 10bits for the node ID, 14bits for the page, and 8bits for the offset (2^{22} objects of any size per node).

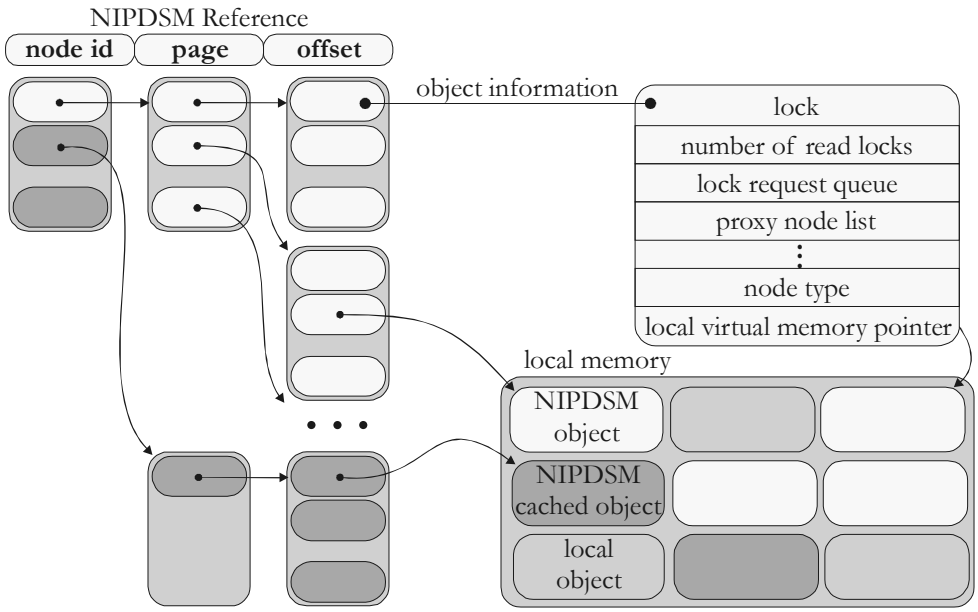


Figure 1: The NIPDSM Virtual Object Table

A compiler can reduce NIPDSM costs by taking advantage of the fact that the object-locking process returns the virtual address of the local copy of the object. The virtual address can be used until the object is unlocked after the method call is completed. Therefore, only one traversal of the Virtual Object Table is required per method call.

A further optimisation that can be made by a compiler is to wrap consecutive method calls on the same object within only one pair of lock operations. This optimisation reduces the runtime costs associated with the locking and unlocking processes, and with mapping the NIPDSM Reference to a virtual address.

3.5 Caching Optimisations

As an object-based DSM system, the NIPDSM has an advantage over page-based systems in that it does not suffer from *false sharing*. However, pure object-caching DSM systems are inefficient, as they cannot benefit from the *spatial locality* found in memory accesses. To overcome this, NIPDSM provides a set of new caching techniques for object-based DSM systems. One is a variant of page caching, but others utilise runtime information in order to improve further the cache-hit rates of an object-based software DSM system.

The NIPDSM caching techniques are based on *object grouping*: each technique selects a group of objects that make up an object group. The choice of the objects depends on the caching technique used but the following are common to all the techniques: The total number of objects that make up the object group is a configuration parameter of the NIPDSM. The selection of the optimal number of objects in the group is currently being investigated. A potential problem with the object grouping approach to caching, if used on its own, is that the size of the object group is expressed independently of the size of the objects that it contains (e.g., a group of movie objects will occupy considerably more memory than a group of the same size that contains complex number objects). As it may be undesirable to transfer and cache very large amounts of memory, a maximum memory size can be set for the group, the *cache block size*. Therefore, objects are added to the group until the limit on the group members is reached, or until the memory size of the group reaches the cache block size. If the size of the requested object happens to exceed the cache block size, then the object is sent as is (i.e., it is not split across several cache blocks) but no object grouping takes place.

Object grouping based on location. When a Manager Node receives a request from a node that wishes to become a Proxy Node for an object, a set of objects stored around that object are included in the object group. The requesting node will become a Proxy Node for all the objects in the group. As will now be described, the technique allows spatial locality to be exploited in a way that is more flexible than in conventional page caching, as the object group does not have to be of fixed size or even of fixed position.

Firstly, the requested object is added to the object group. Then, the objects in the NIPDSM VOT situated immediately before and after the requested object are added to the object group if the requesting node is allowed to become a Proxy Node for them (see rules in Section 3.2). The process continues until:

- The cache block size is reached.
- The limit on the number of objects in a group is reached.
- No more objects are available to be added.

The object group may contain objects situated on several pages of the Virtual Object Table (Figure 2).

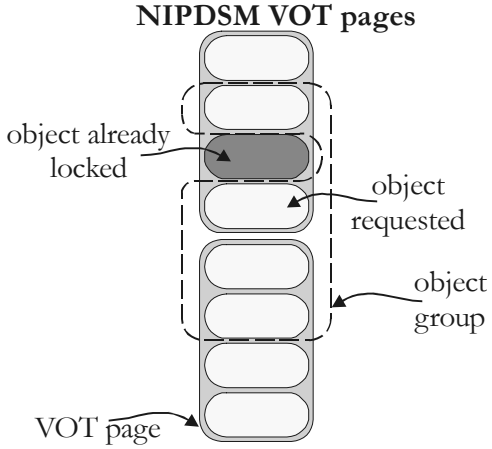


Figure 2: Objects grouping based on location

In order to avoid false sharing, only those objects for which the requesting node is allowed to become a Proxy Node are included in an object group (i.e., the objects that are free if a Write Proxy request was submitted, or the ones that are free or read locked if the request was for a Read Proxy).

The NIPDSM invalidation scheme also avoids false sharing. Unlike lock operations, invalidations only affect one object at a time. When a node receives an invalidation request for an object from the Manager Node of that object, it does not invalidate a group of objects but only the one that was requested. If more than one object were to be invalidated at the same time, the NIPDSM would suffer from false sharing, as do page-based DSM systems. However, a grouping approach to object-invalidation that does not lead to false sharing is currently being investigated. In the grouping approach to invalidation, the Proxy Node that receives an invalidation request for an object selects other objects, based on location, and invalidates them if they are not locked by a task.

As with paging schemes, object grouping based on the location of the objects in the NIPDSM VOT benefits computations that exhibit spatial locality in their memory accesses. However, unlike paging schemes, it does not suffer the disadvantage of false sharing, as locking is still performed per object rather than per page.

Object grouping based on locking history. When the Manager Node satisfies an object lock request from a local task or an object Proxy Request from a node, it links the affected object in its *locking history* list.[3] Only the objects managed by that node can be added to the list.

Figure 3 shows an example of how object grouping based on locking history works. When a Manager Node receives a request from another node that wishes to become a Proxy Node for an object *obj1*, the objects *obj1-obj4* are added to the object group (only if the requesting node is allowed to become a Proxy Node for those objects) because they are linked in the history list (Figure 3a). If a local task acquires a lock on *obj5* or a node submits a request to become a Proxy Node for *obj5*, then *obj5* is automatically added at the end of the list (Figure 3b). If the next object to be locked is *obj2*, it is added at the end of the list but it is still pointing to *obj3* (Figure 3c). Although the 'list' does not terminate anymore, the scheme still works. For example, if *obj3* were to be locked, all objects following *obj3* in the list up to and including *obj2* would have been included in the object group.

[3] The data structure for the object in the NIPDSM Virtual Object Table is added to the list.

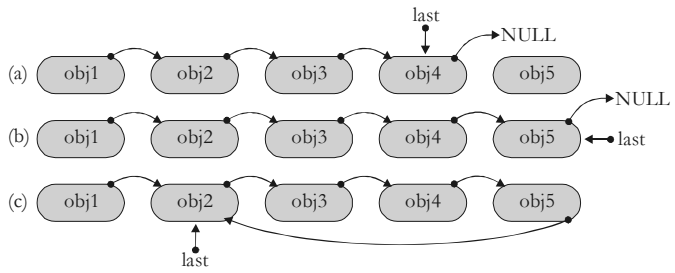


Figure 3: Locking history list

The following rules determine when to stop adding objects to the object group:

- The cache block size is reached.
- The limit on the number of objects in a group is reached.
- The first object in the object group (the one that the lock or the Proxy Node request was for) is reached (i.e., to avoid cycling through the list).

Two locking history lists are maintained on a per node basis: one for the write locks and one for the read locks. An object may be part of both lists at the same time.

The locking history technique benefits parallel computations that repeatedly operate on a set of objects that are not related neither spatially nor via references (see below).

Object grouping based on relations. For applications in which dynamic data structures are used, page-caching systems are inefficient if the objects comprising the structure are not spatially related (possibly because they were created by separate parallel computations allocating objects in different areas of memory, or even on different nodes).

In NIPDSM, it is possible to provide information about the relations between the objects. For example, in a tree-like dynamic data structure, if object O is a node of the tree and O_1, O_2 are leaves of that node, then O can be associated with O_1 and O_2. When a Manager Node receives a request from a node wishing to become a Proxy Node for object O, objects O_1 and O_2 are also added to the object group, if possible. The algorithm works recursively, in a breath-first manner, so that the objects associated with O_1 and O_2 are also added to the group (if the requesting node is allowed to become a Proxy Node for those objects). The object selection process ends when:

- The cache block size is reached.
- The limit on the number of objects in a group is reached.
- No more objects are available to be added.

Object relations are maintained as a list of NIPDSM References on per object basis, and can be altered at runtime as the dynamic data structures change. An additional approach is also investigated in which the associations are maintained on a per class basis, rather than per object, in order to reduce memory usage and simplify compilation.

Object grouping based on relations is beneficial for computations that operate on dynamic data structures (e.g., lists, trees, etc.). Although, the current implementation supports object relations across different nodes, there is not an advantage in doing so because those objects will not be included in the object group. An object can only be added to the object group if the node that submitted the Proxy Node request can become a Proxy Node for that object. Only the Manager Node of an object can decide whether a node can become a Proxy Node for that object or not. Therefore, an object relation with an object created on another node is ignored.

4. PERFORMANCE RESULTS

The NIPDSM has been implemented as part of the NIP library in C++ and tested on a variety of operating systems (e.g., Windows98, WindowsNT, Linux). It fully supports distributed memory systems with both multiprocessor and uniprocessor nodes, including networks of workstations.

The performance results presented in this section were obtained after the execution of test applications on a cluster of eight PentiumII 233MHz workstations interconnected by a shared Fast Ethernet

100Mbps network. Each workstation had 64MB of main memory and was running an instance of the Linux operating system (RedHat6.0, with kernel 2.2.6). In all tests, the egcs 1.1.2 C++ compiler was used with the –O2 option enabled. The current implementation of NIP uses the TCP/IP protocol provided by Linux for inter-node communication rather than an optimised, low-cost communication subsystem like U-Net [10].

Vector size: 100,000 **– Object size:** 28 bytes	
Virtual memory object allocation and de-allocation	2.3
NIPDSM object allocation and de-allocation	8.1
NIPDSM array object allocation and de-allocation	5.1
NIPDSM lock and unlock operations	3.4

Table 1: Runtime costs per vector element (in usecs)

Table 1 presents the runtime costs of the main NIPDSM operations. 100,000 28bytes-long objects were used to measure the NIPDSM allocation and de-allocation costs and show how they compare to the cost of the virtual memory equivalent operations. NIPDSM also provides an interface for C-style arrays allocation and de-allocation, which brings down the runtime costs by *3usecs* per object. The runtime cost of locking and unlocking an object is shown.

In the following tests, NIPDSM is configured to use a group size of 256 objects and a cache block size of 2Kb.

In order to show the importance of the caching techniques presented in Section 3.5 the runtime cost of a lock operation on an object that is not cached was measured (the cache-miss runtime cost). In NIPDSM, the latency of locking and caching a 28byte-long object was found to be *9.6msecs*. NIP tries to overlap communication with computation by creating a new task to keep the node busy until the object is cached and the task that issued the lock request can be resumed. However, there may not be other work available in the runtime system. Therefore, caching is very important.

The performance of each of the different caching techniques described in Section 3.5 is now presented.

Vector size: 100,000 **– Object size:** 28bytes	
Function granularity (msecs)	**Speedup (8 nodes)**
0.21	2.67
1.18	3.70
2.41	4.11
5.44	4.52
9.67	4.72
21.79	4.97
38.76	5.15

Table 2: Speedups achieved

First, the object grouping caching technique based on the location of the objects in the NIPDSM VOT is investigated. A vector of 100,000 28byte-long objects was created on one of the nodes of the parallel system. Then, a task is spawned on every node to map a function onto a portion of the vector (12,500 objects). Every object has to be (read) locked before the function can be applied to it. The locking operation guarantees that the object is cached on the node before the function application is performed.

As expected, the application exhibits a very high degree of spatial locality in memory access. Indeed, the cache-hit rate on each of the nodes was found to be *98.65%*. Of course, the node where the vector was created exhibits a cache-hit rate of *100%*. Table 2 shows the speedups achieved for different granularities of the function. The performance results confirm that it is advantageous to exploit spatial locality in memory access for caching purposes even on object-based software DSM systems.

A tree-sum algorithm was used to test the performance of the object grouping caching technique based on relations. The nodes of a quad-tree of depth 10 (349,525 28byte-long objects) were added recursively.

A vector of size 349,525 was allocated in the NIPDSM. The quad-tree is recursively constructed by randomly choosing objects from the vector. During the construction of the tree, every tree-node is associated with its children. The nodes of the quad-tree are scattered in the NIPDSM and, therefore, object grouping based on location cannot be used to improve caching.

Tree depth: 9 **Total number of possible parallel tasks:** 87,381 **Average tree-nodes processed per task (location):** 38.8 **Average tree-nodes processed per task (relations):** 86.1		
Node	**Cache-hit rate (%) of object grouping based on location**	**Cache-hit rate (%) of object grouping based on relations**
1	100.00	100.00
2	75.60	87.12
3	85.10	98.82
4	72.84	97.59
5	83.41	98.82
6	79.14	95.29
7	74.68	95.27
8	51.97	67.68
Avg.	**74.68**	**91.51**

Table 4: Tree-sum with object grouping based on location and relations

In this test, the NIP lazy task creation technique for recursive computations [3] is used to dynamically create parallel tasks to sum sub-trees. NIP has full control over the number and location of parallel tasks in the parallel system. When a parallel task computing the sum of a sub-tree attempts to acquire a lock on a node of the tree, it has to suspend, if the tree-node is not cached, and wait until its state copied from its Manager Node. As a result, the NIP runtime will automatically create a new task to keep the local processor busy. However, the only available work on the whole parallel system is to calculate the sum of sub-trees. Therefore, all the new parallel tasks spawned will immediately suspend waiting for the tree-nodes they require to be cached, unless the objects are already cached. Hence, it is evident that NIPDSM caching plays a very important rôle in the performance and resource utilisation of the parallel system as it minimises the suspension/resumption costs.

Tree depth: 10 **Total number of possible parallel tasks:** 349,525			
Node	**Possible parallel tasks created (%)**	**Tree-nodes processed per task**	**Cache-hit rate (%)**
1	0.25	174.32	100.00
2	0.40	28.96	36.71
3	0.12	16.98	89.25
4	0.10	101.16	24.77
5	0.09	21.14	92.88
6	0.13	18.32	87.94
7	0.09	107.63	27.34
8	0.15	127.38	97.91

Table 3: Tree-sum with object grouping based on relations

In the tree-sum test, only *1.3%* of the 349,525 possible parallel tasks are actually created on the 8 nodes; the rest of the work is executed inline. The cache-hit rate on a node (Table 3) is highly depended on the granularity of the tasks created. If, for example, a task is dynamically created to add only the last four leaves of a path in the tree, then no cache-hits will be generated as the leaves are not associated with any other objects. Indeed, the performance results indicate that on some nodes the cache-hit rate is low although the number of tasks created on that node is higher than the number of tasks executed on other nodes (Table 3).

In order to show the performance benefits of the object grouping based on relations caching technique when compared to object grouping based on location, the same tree-sum application was also used for a tree-depth of 9. The results presented in Table 4 illustrate the difference in the cache-hit rates between the two techniques. Due to the high cache-hit rate of object grouping based on relations, NIP is not forced to create so many new tasks, therefore better utilising the available processors (the average tree-nodes processed per task in Table 4). Indeed, a speedup of *6.8* was achieved when using object grouping based on relations, over object grouping based on location.

Finally, the object grouping caching technique based on the locking history was investigated through a binary search algorithm. A sorted vector of 100,000 28bytes-long objects is constructed on one node and a task is spawned on another node to search for a random object in the vector. When the task completes, all the cached objects are invalidated so that the next task to be spawned will not find any

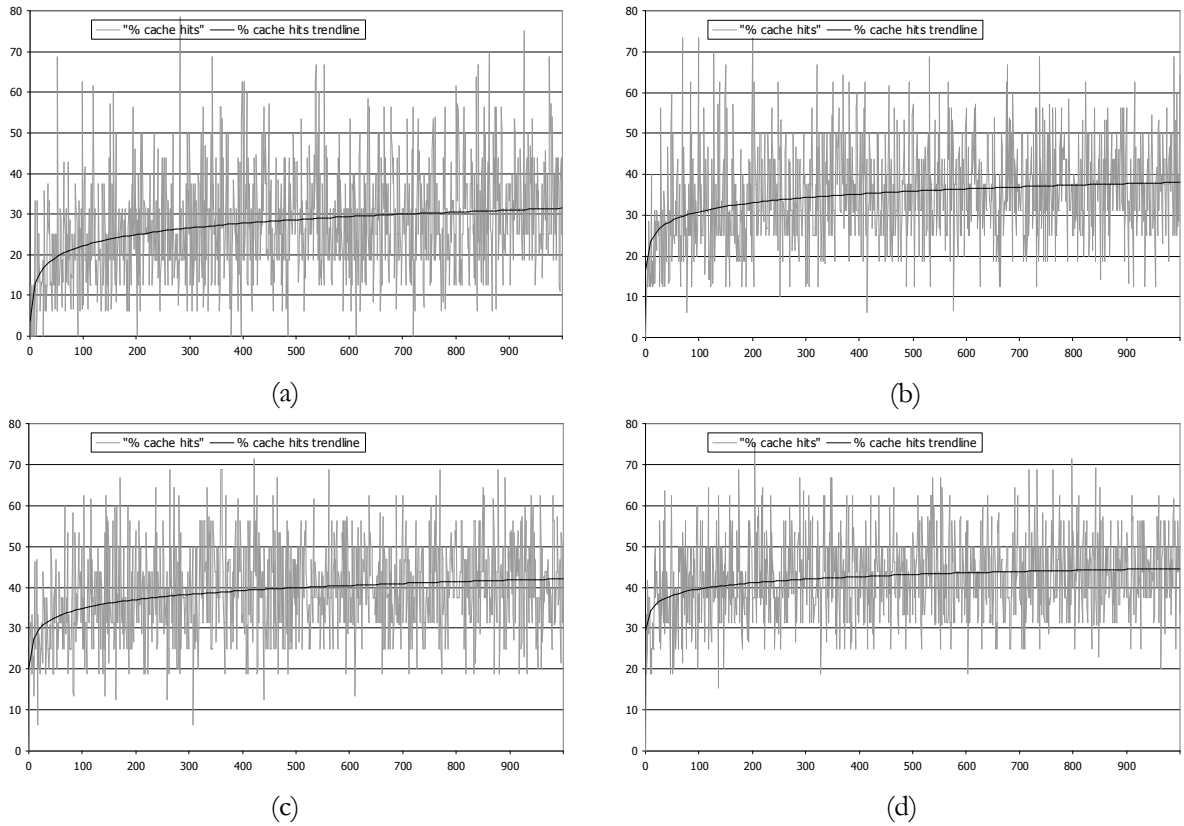


Figure 4: Cache-hit rates with the object grouping based on history when executing 1,000 searches using binary search through a sorted vector of 100,000 objects (each object is 28bytes long). A random object is searched in (a) the whole vector, (b) half of the vector, (c) a quarter of the vector, (d) an eighth of the vector

cached objects on that node. However, the locking history list is allowed to survive between runs. The process is repeated 1,000 times to allow the locking history list to develop.

The caching technique based on locking history was introduced as an optimisation for computations that repeatedly lock objects in the same order. In binary search, the computation may at each step, depending on the search key, choose either half of the remaining portion of the objects for the next step. Therefore, if the sequence of keys is randomly generated for the 1000 runs, the history list should be a poor predictor of future object accesses and cache hit rates should be low. This is shown in Figure 4a, in which a *30%* cache-hit rate is observed. Figure 4b to Figure 4d present the cache-hit rates when searching for an object in just a portion of the vector (i.e., 1/2, 1/4, and 1/8). The results show that the cache-hit rate increases and that the locking history list is developed more quickly. This is because, when searching for an object in only a portion of the vector, the order in which the objects are locked is largely repeated from one run to the next. Therefore, the objects that are cached due to the locking history based object grouping are more likely to be the ones that are required. As a result the cache hit rate increases.

5. FUTURE WORK AND CONCLUSIONS

The paper has described the design, implementation and performance of an object-based DSM for an object-oriented parallel system. This has shown that it is possible to design a set of optimisations to exploit the characteristics of object-oriented systems in order to deliver higher performance than could be achieved by a conventional DSM design which took no account of the object-oriented nature of the system. These optimisations include a reduction in lock operations, and new object-based caching techniques.

The paper shows that it is possible to take advantage of spatial locality in memory access in object-based software DSM systems in a way that is similar to page-based schemes. Furthermore, the NIPDSM can utilise information about the relations of the objects and their locking history to improve caching, as the experimental results—presented in Section 4—show.

These results demonstrate the performance benefits of the different object grouping techniques. However, the applicability of each technique to different types of applications requires further investigation. The effect on performance of the combination of the techniques is also being considered, as is the issue of predicting, at compile time or runtime, the technique most suitable for a particular application.

Although the paper has used the NIP parallel system as a vehicle for investigating these issues, a range of object-based parallel programming systems could exploit the mechanisms embodied within the NIPDSM. For example, NIPDSM is currently being applied to the execution of parallel Java programs, to which it has a natural fit. This is largely because in Java, as in NIP, all data is encapsulated in objects. Further, it is possible to automatically analyse the byte-code of Java methods to determine if they read or write the state of the object, and so ensure that the appropriate locks are taken when a method is called.

6. REFERENCES

[1] Watson, P. and Parastatidis, S., The NIP Parallel Object-Oriented Computational Model, in *Network-based Parallel Computing: Communication, Architecture, and Applications - Third International Workshop, CANPC'99*. Orlando, Florida, USA. Springer-Verlag, Lecture Notes in Computer Science series, vol. 1602, 1999.

[2] Sargeant, J. United Functions and Objects: An Overview. UMCS-93-1-4, Dept. of Computer Science, University of Manchester. 4-1-1993.

[3] Watson, P. and Parastatidis, S. An Optimised Lazy Task Creation Technique for Iterative and Recursive Computations. To appear in the *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications 1999*, Las Vegas, Nevada, USA. 28-6-1999.

[4] Ahuja, S., Carriero, N., and Gelernter, D. Linda and Friends, Computer, vol. 19, no. 8, pp. 26-34, 1986.

[5] Jul, E., Henry, L., Hutchinson, N., and Black, A., Fine-Grained Mobility in the Emerald System, ACM Transactions on Computer Systems, vol. 6, no. 1, pp. 109-133, 1988.

[6] Ramachandran, U. and Khalidi, M. Y. A. An Implementation of Distributed Shared Memory, Software Practise and Experience, vol. 21, pp. 443-464, 1991.

[7] Protic, J., Tomasevic, M. and Milutinovic, V. Distributed Shared Memory - Concepts and Systems, IEEE Computer Society, 1998.

[8] Bershad, B. N. and Zekauskas, M. J. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. School of Computer Science, Carnegie Mellon University CMU-CS-91-170, 1991.

[9] Bershad, B. N., Zekauskas, M. J. and Sawdon, W. A. The Midway Distributed Shared Memory System. Presented at *IEEE COMPCON Conference*, 1993.

[10] v. Eicken, T., Basu, A., Buch, V. and Vogels, W. U-Net: a User-Level Network Interface for Parallel and Distributed Computing. Presented at *15th ACM Symposium on Operating Systems Principles*. Copper Mountain, Colorando, 1995.