Murks – A POSIX Threads Based DSM System

M. Pizka, C. Rehn
Institut für Informatik, Technische Universität München, 80290 Munich - Germany
email: {pizka,rehn}@in.tum.de

ABSTRACT
The shared memory paradigm provides an easy to use programming model for communicating processes. Its implementation in distributed environments has proofed to be harder than expected. Most distributed shared memory (DSM) systems suffer from either poor performance or they are very complicated to use. The DSM system Murks, presented in this paper, is the result of the sobering experiences we made by trying to integrate an existing DSM system into a distributed OS.
Murks distinctive features are 1) full support for POSIX multithreading instead of an own proprietary thread-package and 2) it does not burden the programmer with additional, difficult to use, memory management services. These advantages are achieved by a tight integration of the DSM subsystem into the OS. By this, Murks is able to provide a better combination of performance and ease-of-use than other DSM systems.

KEY WORDS
distributed systems, operating systems, memory management, multi-threading, distributed shared memory

1. Motivation

Murks is a page-based distributed shared memory (DSM) system providing full transparency concerning the physical distribution of the hardware (HW) configuration. Instead of focusing on peak performance, our major design goal was to develop a DSM that can be seamlessly integrated into a distributed and parallel operating system (OS). The reason for this was the observation and (frustrating) practical experience, that there are certainly many different DSM systems with well-elaborated management strategies but none of them proved to be suitable as a OS level service cross-cutting several applications of a long-lived system.

To substantiate these statements, we will first sketch the key design principles of the distributed OS MoDiS [PE97] for which a DSM system was sought after, in 1.1. Afterwards, we are able to define requirements for a suitable DSM system and to discuss the pros and cons of the memory and execution models provided by existing DSMs in section 2.. Section 3. gives an explanation for the lack of support for POSIX multi-threading [IEE95] in most DSM systems. Following these observations, we introduce the concepts of our DSM system Murks, which aims at overcoming the observed shortcomings in section 4., before we describe implementation details, such as Linux kernel modifications, in 5.. Section 7. concludes with a brief summary of our observations and Murks' capabilities after discussing its performance in 6..

1.1 Distributed OS Context

In the research project MoDiS (Model oriented Distributed Systems) [EP99] we follow a *language-based*, *integrated*, and *top-down* oriented approach to develop a distributed OS that provides a high-quality programming environment (i. e. ease-of-use, reusabilty, maintainability, etc.) as well as proper performance for general purpose distributed computing.

1.1.1 Language-based

The term language-based means that the development of the OS is driven by the development of high-level language concepts for the specification of parallel and distributed systems. E. g. concurrency, cooperation and synchronization are language concepts instead of runtime level services. By this, we are able to provide suitable programming concepts without being restricted neither to the features of a predefined OS nor to the concepts of an already existing programming language, such as passing object references in Java, which must be regarded as inadequate in distributed environments.

INSEL This programming model is implemented and offered to the programmer by the syntax of the Ada-like [Ada83] programming language INSEL [Win96] and its optimizing source to machine-level compiler [Piz97]. INSEL can be characterized as a high-level, type-safe, imperative and object-based programming language, supporting explicit task parallelism.

Objects are dynamically created during execution as instances of class describing objects. To prevent from dangling pointers, objects are implicitly deleted by a garbage collector. Determined with the class definition, objects may either be active, called actors, or passive ones. An actor defines a separate flow of control and executes concurrently to its creator.

Actors may communicate directly in a rendezvous style as well as indirectly by accessing shared passive objects (shared memory paradigm). All requests to objects are served synchronously.

DSM-Requirements Part 1

R1 trivial: We need a DSM subsystem that efficiently implements actor interaction via shared passive objects of arbitrary granularity. Note, passive objects may be as small as a single integer variable or complex data structures composed of other passive and active objects.

R2 The DSM must be compatible with varying degrees of parallelism defined by actors of arbitrary number and size (in both, space and time). Due to the high-level of abstraction of the language concepts, it would be completely inappropriate to rigidly map the actor concept to e. g. heavy-weight processes. Instead, an actor might be implemented as a kernel-level or user-level thread or even inlined into another thread. The DSM must not obstruct this flexibility.

R3 The language-based philosophy demands, that the application level is left unspoiled by DSM concepts. The programmer must not be burdened with any memory management concepts besides creating objects in stack or heap space.

1.1.2 Integrated

The entire distributed system in execution, comprising several applications and the distributed OS, is regarded as a *single globally structured system* [PE97], which can be dynamically extended with new applications and maybe also OS functionality.

One the one hand, this holistic view provides an ideal integration of knowledge about the properties of distributed objects and their interdependencies. Thus – at a conceptual level – the OS may consider non-local information for important resource managements decisions instead of being limited to local optimization a priori. On the other hand, the single system view implies *longevity* of the system, which has a surprisingly strong impact on low-level OS management mechanisms and therewith also on the DSM subsystem.

DSM-Requirements Part 2

R4 The DSM subsystem has to support a single (integrated) virtual address space (VA) across all nodes. Note, it is not obvious that a DSM system also implements a single VA!

R5 The DSM must support multiple applications simultaneously and provide partial cleanup of memory partitions previously allocated by terminated applications.

R6 It must also support long-lived large-scale applications that eventually migrate between arbitrary nodes of the system.

1.1.3 Top-Down Management

The philosophy behind our approach to automate resource management is to free the application level from all tasks but specifying parallel and cooperative computations. This especially holds for all tasks dealing with the physical distribution of the execution environment. Note that this principle is a prerequisite for achieving portability, scalability and enhanced maintainability of distributed applications.

It is solely within the responsibility of the OS to map the abstract requirements of applications specified by means of the language INSEL to physically distributed resources, such as implementing the abstract creation of a dynamic integer object by allocating a memory object in shared heap space. Conceptually, this mapping between abstract concepts and concrete HW features is achieved by top-down oriented stepwise refinement, performed by the OS. Practically, some of this transformation steps are performed at compile-time while others are executed at runtime. Hence, all mechanisms, strategies and services of the OS are to be tailored to the language-level concepts to match the requirements of distributed applications [ea97].

1.2 Generalization

It should be obvious, that most of the requirements stated above are relevant for other distributed systems, too and not limited to the scope of MoDiS. Thus, MoDiS should only be regarded as the initial spark for our considerations. Its major contribution is to deliver greater creativity for the design of a distributed computing infrastructure by being language-based and top-down oriented instead of concentrating on how to extend legacy systems.

We believe that integration, automation and transparency are important prerequisites to achieve the desired ease-of-use and performance of distributed systems in general. Hence, these design principles will also be significant for DSM systems beyond the context of MoDiS.

2. Related Work

DSMs have been of strong interest to the OS community from the mid 80s to the late 90s [Li86, ea99]. The main motivation was to provide a simplified programming model relative to the omnipresent message passing paradigm [Sun93, For94]. But several attempts to implement the DSM concept exposed difficulties that proved to be hard or undoable without specialized HW. [Car98] summarizes that DSM systems suffer from

- being too difficult to use, or (non-exclusive)
- severe performance penalties.

Since we are in need of support for a combination of shared memory and parallelism, we have to examine both, memory and execution models, of DSM systems.

2.1 Memory Models

Ivy [Li86] was the first page-based DSM, followed by improvements such as Mirage+ [ea94], TreadMarks [ea96] or Odin [Pea96]. All of theses systems use HW pages as management units and employ a single coherence protocol (CP). Because of false sharing, the performance of these systems strongly depends on the partitioning of data and the access characteristics of the distributed computation [BK98]. Midway [BZS93] is not bound to HW pages. All store operations are performed through the Midway library and no page-faults are triggered. False sharing is circumvented but frequent writes degrade system performance. The CP can be chosen by the programmer but is common to all objects, regardless of different access characteristics. The object-based DSM Munin [Car95] tries to solve both problems. The CP respects the size of individual objects; i. e. for each object a different CP can be chosen.

Independent of being page or object-based, all of these systems share the same motivation: achieving high performance. Little attention is paid to the programming model! Page-based systems require the programmer to explicitly allocate shared regions with awkward additional DSM services that do not provide any heap or stack alike management of data objects within these regions. Thus stack pointers and heaps have to be reimplemented inside applications! Even worse, some systems require that all sharable regions are allocated before starting the actual computation. There is no dynamics concerning sharable regions once the computation has started, which is acceptable in case of a single distributed application but not for a distributed OS. Similarly, object-based systems (e. g. Munin) require the programmer to explicitly mark sharable objects. Again, there is no dynamic transition from private to shared.

Thus, the memory models of the DSMs investigated proofed to be unsuitable in the context of a distributed OS, because of a lack of support for dynamic (de)allocation and the absence of an internal stack and/or heap alike organization of shared regions.

2.2 Execution Model

The programming model introduced in section 1.1.1 allows the user to specify computations with an arbitrary degree of parallelism. An efficient implementation of this model must obviously be based on a light-weight threading concept. Therefore, a suitable DSM system for this kind of distributed and parallel processing must either provide its own thread package or be compatible with existing ones.

One could argue that the execution model is not within the scope of DSMs. But in practice, these two issues can not be completely separated. It is not possible to use an arbitrary DSM system together with some multi-threading package since the DSM directly effects the state machine of processes.

Single-Threaded DSMs
Most DSM-Systems provide a single threaded execution model with one or more heavy-weight processes running on each node; e. g. Ivy, CRL [JKW95], and JIAJIA [E+99]. Obviously these systems are not suitable to implement fine-grained parallelism. Furthermore, the "few-processes-on-each-node" paradigm, requires the programmer to manually split the problem space into partitions that reflect an optimal matching of the problem to the currently available HW resources. Hence, systems designed according to this principle do not scale properly. Neither with problem size nor with the HW configuration.

Multi-Threaded DSMs
The DSM system Quarks [Kha96] focuses on using UNIX workstations for the computation of parallel problems that are usually performed on shared memory multiprocessors. It therefore supports fine-grained parallelism by providing its own light-weight thread package. Alas, this package is based on the (obsolete) `cthread` standard and therefore not compatible with the widespread POSIX standard. As a consequence, Quarks applications can hardly be ported. Furthermore, this thread package is not integrated with other libraries besides the Quarks DSM. As a result, the use of standard C library calls, such as `open, read, printf`, is serverly restricted. In addition to this, the only synchronization primitive fully supported so far are barriers. Mutexes and condition variables are not available, yet.

DSM-Threads [Mue97] partially overcomes these handicaps by supporting an API closely resembling POSIX Threads. It facilitates porting shared memory applications to distributed environments while offering elaborated performance tuning options. Still, DSM-Threads requires its own POSIX alike package and can not be used in conjunction with POSIX thread packages that already exist on target platforms (e. g. Linux, Solaris, HP, ...). Again, this causes the same awkward two-level scheduling [ea92] conflict as in the case of Quarks. Libraries besides DSM-Threads cannot be used unrestricted.

Summarizing these experiences, our goal is to provide a simple page-based DSM system that is fully compatible with POSIX threading as available on modern workstations without imposing additional restrictions.

3. The `mmap` Race Condition

Following the observations of DSM execution models, we studied the reason for the absence of a POSIX threads compatible DSM. The answer is simple: *Conventional OS do not provide a non-interruptible service that could be used to atomically map a memory region as readable and copy the data received via the network to this region.*

Figure 1 shows the subroutine of a page-based DSM[1] that locally maps memory regions received via the network.

[1] Object-based DSM systems suffer analogously from the same problem during unmap!

```
map_netobj (..., addr, len, protocol)
{ ...
  /* stop_all_threads (); */
  mmap( addr, len,
        PROT_READ| PROT_WRITE, ...);
X
  read(sender, addr, len);
  /* cont_all_threads (); */
  ...
```

Figure 1. the mmap race condition

A memory region m first has to be mapped read/write using `mmap` before the contents of m can be written to m (here, using `read`). This is uncritical in a single-threaded system. But in a multi-threaded system it is possible that another thread z is set running between those two operations (position marked with "X"). Now, if z reads from m, it will read undefined values, because m is mapped but its content is not available, yet. Analogously, writes from z to m are lost as soon as the DSM subroutine is rescheduled. Thus, non-atomic mapping and copying of memory regions causes race-conditions that must be resolved. There are four options to prevent disruptions of the DSM handler:

1. modification of the OS kernel
2. increased priority of the DSM handler, which usually requires superuser rights
3. stopping and restarting all threads by sending STOP and CONT signals to all threads
4. using an own user level thread package to gain full control over scheduling

Option 1 requires difficult, non-portable work at the OS kernel level. 2 can not be expected from potential users of the DSM system and 3 (commented in fig. 1) degrades performance. Hence, DSM developers tend to either resign from multi-threading or to choose option 4, although this leads to very limited applicability due to incompatibility with standard libraries. Supposable, the evolution of DSM systems was negatively influenced by these simple reasons.

According to the language-based and top-down oriented approach taken in MoDiS we did not accept the restrictions dictated by an existing OS kernel but chose option 1 to be able to simultaneously provide both, a DSM and multithreading, in a single distributed OS.

4. Murks Concepts

Murks implements sequential consistency [Lam79] in software. In order to keep coherence maintenance simple and to reduce the average cost for read operations by allowing simultaneous reads at multiple hosts, Murks uses a multiple reader/single writer (read-replication) policy. The coherence unit supported by Murks is determined by the size of memory pages in order to use the HW supported page-fault mechanism and to save space needed by management datastructures for finer grained units. Due to the sharing of rather coarse grained memory pages, the CP relies on invalidation instead of update propagation.

4.1 Server – DSMR Management

Murks can be divided into a centralized server, a client runtime library and a kernel patch. The server manages *distributed shared memory regions (DSMRs)* that can be shared by multiple activities (processes or POSIX threads) distributed across interconnected hosts. A DSMR is a continuous memory interval starting and ending on page boundaries. In the following we write DSMR$[x, y[$ for a DSMR that starts at address x and ends at address $y - 1$. A DSMR d may be marked *read-only*. In this case there may be copies of d residing in the physical memories of multiple hosts and visible in multiple process VAs. Or d is marked *read-write* being visible in the VA of one process and residing on a single host, only. In addition to DSMR management the Murks server keeps track of Murks client connections and handles client requests.

To avoid a bottleneck due to the centralized server approach in case of a high rate of client requests the VA may be split into n subsets and the management of DSMRs within those subsets distributed among n Murks servers (fixed distributed-servers). In this case the mapping from memory regions to servers is performed by the client-side fault handler. Each server handles requests exactly in the same manner as a centralized one.

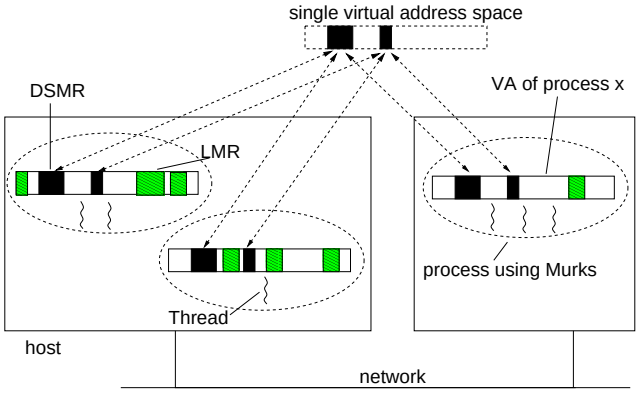


Figure 2. DSMR and LMR management in Murks

4.2 Accessing Shared Memory

Client library routines are used to register a page-fault handler, establish a connection to the Murks server, map and unmap DSMRs and handle page faults. In order to mark a memory region to be shared among multiple activities the region must solely be mapped by an activity. This registers the region at the Murks server and marks it accessible for other activities. In the following we call memory regions that are not shared among processes located on the

same host or activities across multiple hosts *local memory regions (LMRs)*. The interpretation of addresses in LMRs is process specific.

The concepts described so far are illustrated in figure 2. Each process running on a node of the distributed HW configuration may use the Murks DSM in order to get access to the DSMRs of the single VA. The DSMRs are mapped into the VA of each process and can be accessed by all threads running within those processes. Additionally, process specific LMRs may exist. The distinction between local and shared regions improves performance as there is no management overhead, no need for coordination, migration or replication for local memory.

Any process or thread can access memory cells in DSMRs in the same way as in LMRs, i.e. from the clients point of view there is no difference between DSMRs and LMRs besides latency. Thus the programmer is not burdened with additional concepts as for example distinguishing between ordinary and synchronized memory access or programming language level annotations. As a consequence, Murks can be used for remote accesses to heap memory as well as stack memory.

A memory reference to a cell of a DSMR may trigger a page-fault if the page containing this cell is not locally available with the required access rights. This invokes the client-side page-fault handler which requests the missing page from the server. There are two kinds of page-faults depending on the memory access operation. If an activity wants to read a page, but does not have a copy of it a "read page-fault" occurs and the page is requested from the server for read-only access. Analogously, a write-access to a page that is not available or has no write permissions results in a "write page-fault" and a read-write access request is sent to the server.

4.3 Handling Client Requests

All read and write requests are received by the server. It keeps track of the *copy-sets* (sets of all activities holding copies) of DSMRs, splits DSMRs if required and forwards requests for memory pages.
In case of a read-only request, a host owning a copy of the requested page is randomly choosen and asked to provide a copy which is forwarded by the server to the requesting host. In case of a read-write request the server sends invalidations to all sites of the copy-set and waits for confirmation. The sites unmap the invalidated page and confirm the invalidation.After receiving all confirmations the server forwards the contents of the requested page to the requesting host.

4.4 DSMR Splitting

As mentioned above, the Murks server may split DSMRs. In order to reduce the size of datastructures within the server and thereby increase the performance, Murks uses DSMRs instead of single memory pages. Accordingly, when a client does not request a complete DSMR but a sub-interval, e. g. a single page, the server has to split a DSMR into at most three DSMRs. The situation is illustrated in figure 3.

In this example *Client 2* sends a read-write request for DSMR$[a', b'[$ to the server. The server recognizes that $[a', b'[$ is a sub-interval of $[a, b[$, splits DSMR$[a, b[$ into DSMR$[a, a'[$, DSMR$[a', b'[$ and DSMR$[b', b[$ and sends an invalidate request for DSMR$[a', b'[$ to $Client\,1$. After receiving the confirmation together with the contents of DSMR$[a', b'[$ this region is forwarded to $Client\,2$ and the copy-sets are adjusted.

4.5 Unmapping Shared Regions

A shared memory region may be unmapped by an activity at any time. The activity must not be identical with the activity that mapped the DSMR and even sub-intervals or intervals including multiple adjacent DSMRs may be unmapped in a single request. The unmap request is sent to the server which splits or coalesces DSMRs if necessary, distributes invalidations to all activities within the copy set, and frees the corresponding management datastructures after receiving all invalidate-confirmations. After unmapping a DSMR it may be reused as a LMR.

4.6 Synchronization Concepts

Murks provides distributed barriers, condition variables, and mutexes to the application level. Details concerning these concepts are not within the focus of this paper.

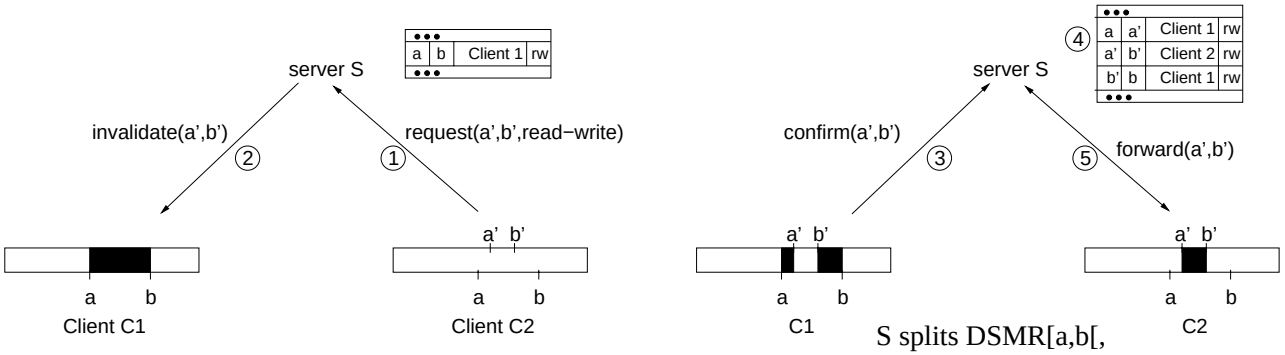
5. Implementation Issues

Murks has been developed as a student project at the Technische Universität München and is integrated into the decentralized single address space management [Reh00] of MoDiS. It has been implemented in C on x86 PCs because first, the MoDiS project is based on a modified Linux kernel running on a cluster of PCs and second, the kernel source code and a kernel-level thread implementation of POSIX threads are available for Linux.

Our current implementation satisfies the requirements R1–R6 motivated in section 1.1.1 and 1.1.2. Murks can be used in conjunction with POSIX threading and it allows transparent remote access to stack objects as well as heap objects.

5.1 Communication

Remote communication between clients and server(s) is implemented either by using TCP sockets or by loading a communicator module (CM) into the Linux kernel. CM allows remote communication by simply writing to or reading from special files of the proc filesystem. For simplicity reasons we will concentrate on the socket variant below.



(a) C2 requests read−write access to DSMR[a',b'[

S splits DSMR[a,b[,
(b) C1 confirms invalidation
S forwards DSMR[a',b'[to C2

Figure 3. DSMR splitting

All client processes have to establish a TCP connection to the server by connectiong to a predefined port by calling an initialization function. All threads within a process share the same communication link. If mutexes are used within an application, not only server connections but also all connections between any pair of client processes have to be initialized.

5.2 Server Concerns

In addition to the selection of a communication method, Murks users may also choose between the integration of the server into their application as a POSIX-thread or run it as a separated process or even a mixture of both in case of a fixed distributed-server approach.

A very important task of the server having great impact on performance is the DSMR management. In order to speed-up operations on the DSMR datastructure all information about DSMRs is stored in a red-black tree like datastructure, so DSMRs and sub-intervals can be searched in O(log n) time, where n is the number of DSMRs managed by the server. Insertion of DSMRs due to splitting or mapping requests of clients and deletion due to unmapping requests are as well performed in O(log n) time.

5.3 Kernel Modifications

The necessity to modify the operating system kernel is due to the non-atomicity of the operation that maps pages belonging to a DSMR into the VA of a client process as described in section 3. (mmap race condition). To avoid the selection of another runnable thread by the Linux scheduler and thereby guarantee atomicity of client DSMR mapping we added an additional system-call and modified the Linux-task structure and scheduling code. The new system call makes the following three services available:

1. mark a task as using extended scheduling by setting an attribute in the task structure,
2. enter a critical region and
3. leave a critical region.

When a task using extended scheduling enters a critical region by calling the appropriate system call service it gets the highest priority of all local tasks having registered to use the extension. The scheduling algorithm may henceforth select other runnable tasks that do not use the extension or the task within the critical region for running until it leaves its critical region.

These kernel extensions are transparent to the application level as the services are usually called from the Murks client library code which handles the mapping of DSMRs received from the server.

5.4 Synchronization

The implementation of barriers and condition variables in Murks is based on a centralized algorithm with the Murks serves as an arbitrator, whereas mutexes are implemented using a distributed algorithm.

As mentioned in section 5.1, the clients must not only establish a connection to the Murks server but also to all other Murks clients in order to use mutexes. Mutexes are implemented using a distributed wait-queue [ea90, NTA96] which reduces the number of messages to O(log n) per critical section.

6. Performance

We claim that there is no need to give any performance results for Murks as Lipton and Sandberg [LS88] proofed that the performance of any sequentially consistent DSM is limited by the minimal packet transfer time between nodes of the distributed system.

Murks' performance is comparable with other page-based, sequentially consistent software DSM systems because its implementation is rather straight forward. Performance of such systems have often been measured in the past.

Instead of weakening the consistency model and make the DSM unmanageable for the application programmer, Murks provides an easy-to-use abstraction that can be utilized to realize remote access to passive objects. In our opinion it is the task of the management system, (compiler, runtime-system, OS kernel, ...) to carefully place shared passive objects within the address space to avoid false sharing and to distribute passive and active objects according to access patterns in order to reduce network traffic. This task is supported by the integrated approach (see section 1.1.2) taken in MoDiS. Structural dependencies between applications are exploited by the distributed OS to make appropriate management decisions and thereby alleviate potential performance penalties.

7. Conclusion

In this paper we argued, that the development of DSM in the past was too much concentrated on achieving high performance — for which DSM systems without specialized hardware are not the ideal concept, anyway. On the basis of the distributed OS MoDiS, we furthermore exemplified important limitations of existing DSM system and argued that these limitations impair the possibility to integrate one of these DSM system into a distributed and parallel OS. From the lessons learned with the memory and execution models of existing DSMs we derived an own concept for a DSM subsystem that provides an optimal combination of ease-of-use by full integration into the OS infrastructure with satisfying performance.

This concept has been implemented and tested on the Intel x86 platform, mostly within a one semester student project. To achieve our goals, it was necessary to make changes to the kernel, which is noncritical, today, due to the availability of the source codes of OSs such as Linux and Solaris. The benefit of this strategy is, that Murks is the only DSM system known to the authors that is fully compatible with POSIX multi-threading facilities that already exist on modern systems.

References

[Ada83] Ada. *The Programming Language Ada Reference Manual*, volume 155 of *LNCS*. Springer–Verlag, Berlin, 1983.

[BK98] B. Buck and P. Keleher. Locality and performance of page- and object-based dsms. In *First Merged Symp. IPPS/SPDP*, pages 687–693, March 1998.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *IEEE COMPCON*, pages 528–537, February 1993.

[Car95] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, Sep. 1995.

[Car98] John B. Carter. Distributed shared memory: Past, present, and future. slides, 3rd Workshop on High-Level Parallel Programming Models and Supportive Environments, March 1998.

[E+99] M. R. Eskicioglu et al. Evaluation of the JIAJIA software DSM system on high performance computer architectures. In *Hawaii Int. Conf. on System Sciences*. IEEE CS, January 1999.

[ea90] Y.-I. Chang et al. An improved O(log N) mutual exclusion algorithm for distributed systems. In *Conf. on Parallel Processing*, pages 295–302, Urbana-Champaign, IL, August 1990.

[ea92] T. E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Transactions on Computer Systems*, 1992.

[ea94] B. D. Fleisch et al. Mirage+: A kernel implementation of distributed shared memory on a network of personal computers. *Software Practice and Experience*, 24(10), October 1994.

[ea96] C. Amza et al. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb. 1996.

[ea97] M. Pizka et al. Evolving software tools for new distributed computing environments. In *Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, 1997.

[ea99] C. Amza et al. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, March 1999.

[EP99] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. *Journal of Supercomputing*, 13(1):33–55, January 1999.

[For94] MPI Forum. MPI: A message-passing interface. Technical Report CSE-94-013, Oregon Graduate Institute of Science and Technology, April 1994.

[IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE CS Press, 1995.

[JKW95] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *ACM Symp. on Operating Systems Principles*, December 1995.

[Kha96] D. R. Khandekar. Quarks: Distributed shared memory as a building block for complex parallel and distributed systems. Master's thesis, Dep. of CS, University of Utah, March 1996.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multi process programs. *IEEE Transactons on Computers*, 28(9):690–691, September 1979.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dep. of Computer Science, Yale University, New Haven, CT, October 1986.

[LS88] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[Mue97] F. Mueller. Distributed shared memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.

[NTA96] Mohamed Naimi, Michel Trehel, and André Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, April 1996.

[PE97] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *HICSS*, Maui, Hawai, January 1997. IEEE CS Press.

[Pea96] A. N. Pears. Odin: Implications and Performance of a Novel DSM Design. In *Proc. of ICSE'96*, Jan. 1996.

[Piz97] Markus Pizka. Design and implementation of the GNU INSEL-compiler gic. Technical Report TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[Reh00] Christian Rehn. Top-Down Development of a Decentralized Single Address Space Management . In Hamid R. Arabnia, editor, *PDPTA–2000*, pages 573–579, Las Vegas, NV, June 2000.

[Sun93] V. Sunderam. The PVM concurrent computing system. In Anonymous, editor, *The commercial dimensions of parallel computing*, pages 20–84. Unicom Seminars Ltd, 1993.

[Win96] H. M. Windisch. The distributed programming language INSEL – concepts and implementation. In *High-Level Programming Models and Supportive Environments*, pages 17 – 24, April 1996.