On the Design and Implementation of DSM-Threads

Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de *phone: (+49) (30) 20181-276* *fax: -280*
WWW: `http://www.informatik.hu-berlin.de/~mueller`

Abstract *This paper discusses design goals, design decisions, and implementation choices of DSM-Threads, a runtime system to support distributed threads with a distributed shared virtual memory (DSM). DSM-Threads provides a distributed runtime system with a kernel on each node, which relies on POSIX threads locally and a decentralized communication subsystem between nodes. Support for multiple data consistency protocols facilitates the migration from shared-memory POSIX threads to DSM-Threads in a distributed environment on one side and offers opportunities to fine-tune the program for DSM-Threads on the other side. The overall approach enhances portability of the system and allows support for heterogeneous environments without modifications of compilers or operating systems. The paper also describes the support for higher-order distributed language features by example for Ada95. Finally, a first evaluation of the system's performance is given. DSM-Threads is, to our knowledge, the first runtime system to support distributed threads on top of POSIX Threads via distributed virtual shared memory.*

Keywords: distributed systems, heterogeneous systems, distributed runtime systems, distributed shared memory, threads, POSIX threads.

1 Introduction

DSM-Threads is a runtime system to support distributed threads with a distributed shared virtual memory. This work was motivated by the emerging standards for concurrency, in particular the POSIX threads standard [1]. Applications that adhere to these standards rely on shared memory and thus experience their best performance on shared-memory multiprocessors (SMPs). The programming model of these applications, however, contains inherent parallelism that is not only limited to SMPs but can readily be exploited on a distributed system with shared virtual memory, such as DSM-Threads.

The advantages of utilizing a distributed system are their existing availability as networks of workstations and their potential to scale well. The disadvantages of distributed systems are rooted in the absence of global state that prevents the reuse of well-understood shared-memory algorithms. Instead, distributed algorithms have to be utilized that perform explicit communication, either using a centralized approach (a potential bottleneck) or more complex decentralized methods (*e.g., via* timestamps). DSM-Threads, on the other side, facilitates the migration from a concurrent programming model using shared memory to a distributed model with minimal changes of the application code. Programs built atop POSIX threads for uniprocessors or SMPs may continue to use the shared-memory algorithms *and* exploit the processing power of a distributed system *via* DSM-Threads.

This paper discusses the design and implementation of DSM-Threads. First, the main design issues and their realization are sketched. Then, a number of issues are highlighted: multi-threading within each node, the choice of consistency models, priority support, an asynchronous access model and heterogeneous support. A first evaluation of the current implementation is discussed, including the support for Ada95 language features. Finally, related work and conclusions are presented.

2 Design Issues

The design of DSM-Threads was driven by a number of design goals. In the following, the major design goals are motivated and the corresponding implications are considered.

Portability: The system depends on established and emerging standards (*e.g.*, POSIX) and widely ported tools (*e.g.*, gcc and other Gnu utilities).

Compatibility: The interface of DSM-Threads resembles closely the interface of POSIX Threads, both in syntax and semantics. Thus, minimal modifications to source programs commonly suffice when migrating from Pthreads to DSM-Threads (toward "backwards compatibility").

Communication Issues: Communication between nodes requires a considerable overhead. Thus, consistency and synchronization protocols have been chosen such that they reduce the amount of communication as much as possible. Furthermore, the protocols are based on decentralized approaches to avoid bottlenecks.

Responsiveness: The local resources of a node should be used when possible. The multi-threaded approach of DSM-Threads exploits the local processing capabilities by switching from one user thread to another on blocking operations, *e.g.* when communication with other nodes becomes necessary. Furthermore, the protocols utilize asynchronous communication and assign workers to handle messages, which enhances the responsiveness of the system.

Extensibility and Tuning: A set of protocols offers users the option to design simple distributed programs whose performance can be enhanced step-by-step by advancing to more sophisticated protocols on demand. In addition to the existing protocols, the design facilitates the introduction of new protocols for future enhancements.

Heterogeneous Support: The processing power even of architectures with different data representations should be exploited. The communication layer of DSM-Threads makes such issues transparent to the user.

Modular Design: DSM-Threads consists of several independent subsystems, each of which is dedicated to a widely isolated aspect of the system. For example, communication is realized in hierarchic layers from a low-level communication interface over a common DSM-message protocol layer to the communication aspects of various consistency protocols. This design facilitates the exchange and addition of modules to adapt the system, in this case, to new communication standards or future consistency models.

Multi-Paradigm Support: Conventional high-level languages as well as object-oriented languages should be supported. A choice between different consistency models provides this property.

3 DSM System Architecture

On each node that is utilized for distributed thread execution in the system, a process is created that contains DSM-system threads, a Pthreads system (currently FSU Pthreads [2]) and user threads (see Figure 1). The DSM runtime system consists of worker threads and a communication server, a separate thread of control with the highest priority of all threads within this process. The task of the communication server is to open a communication channel and wait for incoming messages. When a message is received, the protocol header is decoded to determine the message type. Depending on the type, a message may be acted upon by the communication server if little work has to be done (*e.g.*, for forwarding requests), or the processing of the message may be delegated to a worker thread.

Worker threads can also be activated via the DSM runtime system to function as user

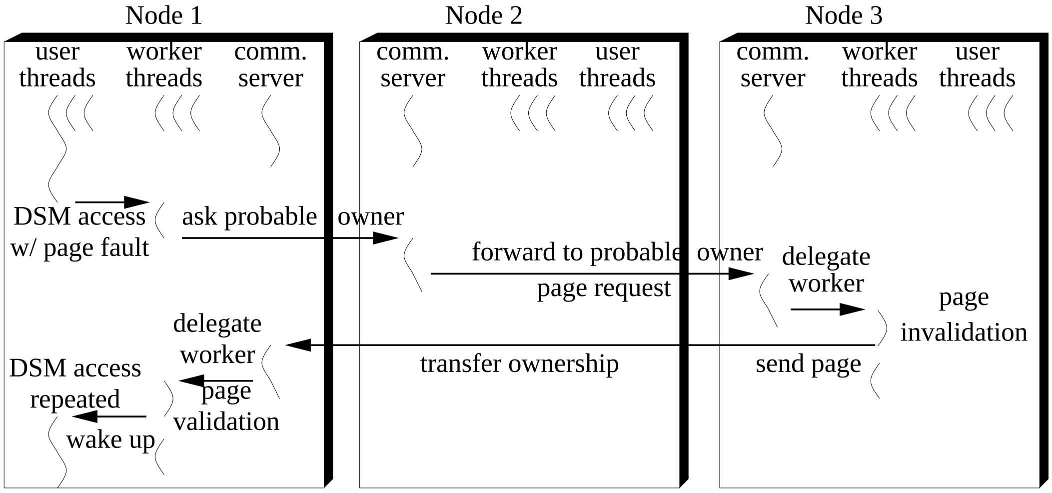


Figure 1: Operational Model for Page Fault Handling

threads (`dsm_thread_create`) or when a page fault occurs, *i.e.* when DSM data is referenced with insufficient access rights on the node. The page fault handler then activates a worker thread to perform the task of obtaining the proper access rights to the page, which is an asynchronous signal-unsafe operation [1] that cannot be safely executed within the page fault handler. The worker thread may then send a request to the communication server of another node (the probable owner). Upon response, the communication server of the current node delegates a worker to validate the page and change the access rights. The worker then reactivates the user thread.

4 Data Consistency and Access Protocols

Keeping DSM data consistent can be regarded as a trade-off between user-friendliness and efficiency. The user would prefer to keep a globally consistent view of data changes at all times; after all, the DSM data is said to be *shared* from the user's view. However, the communication overhead in distributing a change of DSM data to other nodes imposes a limit on the frequency of synchronization to ensure efficiency, responsiveness, and user acceptance of the system.

We decided to provide the user with a number of consistency models:

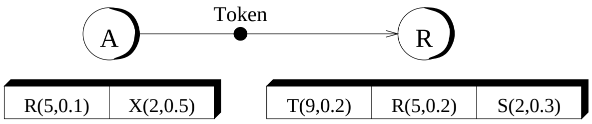
- Single Reader, Single Writer (SRSW): optional interface for the user when no distinction between read and write locks exists;
- Multiple Reader, Single Writer (MRSW): default for the user, implemented via an improved dynamic distributed manager algorithm supporting page migration with page invalidation [3];
- Multiple Reader, Multiple Writer (MRMW): advanced user-models, including Entry Consistency (EC) [4], open for future extensions.

The operational model of handling page faults is depicted in Figure 1 to illustrate the internal architecture of DSM-Threads. The pages currently not owned by a node are protected, *i.e.* an access to the page causes a memory fault, for example on node 1. The memory fault is reported to a worker, who determines the (probable) owner of the page and sends a request to the remote owner (node 2). This request is received by the communication server of node 2, where the local page table is consulted. If the node does not own the page anymore, the request is forwarded to its probable owner (node 3). If the communication server on node 3 determines that it still owns the page, it delegates a worker to act upon the page request. The worker invalidates (protects) the page on node 3 and sends the page content directly to the requesting node 1. The communication server of the requesting node 1 then delegates a (possibly different) worker to act upon the sent page. The worker validates the page on node 1 and enables access to the page. Had there been more intermediate nodes, the original page request would have been forwarded along a chain of probable owners. But in the end, the request would be handled by the actual owner (node 3 in the example) as described before. For entry consistency, the operational model differs only with respect to the initiation of the request and reception of a page. These tasks can be performed by the user thread as part of the lock operations instead of delegating worker threads. Through the association of DSM data with locks, page faults and the corresponding signal handlers can be avoided so that signal safety is no longer an issue.

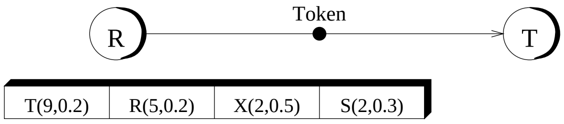
5 Priority Support

Conventional parallel applications do not distinguish between the urgency of subtasks. Such a distinction may, however, be useful to ensure that certain actions are performed promptly if they threaten to create a bottleneck or if they have to be executed by a certain deadline, *e.g.* for real-time applications. POSIX threads provide priority scheduling for this purpose. DSM-Threads is aimed at carrying this priority model into a distributed environment.

The memory consistency protocols described in the last section can be extended to provide priority support. A token-based protocol for mutual exclusion has been developed that orders requests according to priorities. In particular, if a current requester R of a DSM object receives a lower priority request from S, it will store this request after $R's$ in a local queue. Figure 2(a) depicts this situation for priorities 5 and 2 for nodes R and S, respectively. Should R receive a higher priority request from T, then this request would be queued locally in front of $R's$. The request from T would also be forwarded to the probable owner.



(a) before merging queues



(b) after merging queues

Figure 2: Example of Distributed Priority Queuing for Node(Prio, Wait Time)

On token forwarding, *i.e.* when the token owner forwards the token to the first requester in its local queue and piggybacks the queue onto the message, the token receiver merges the piggybacked queue with his local queue. If the head of the merged queue is the current node, the token is used for mutual exclusion; otherwise, the token is forwarded to the requester at the head of the queue (as in this case to T). The merging of queues collapses multiple entries of the same request. For example, in Figure 2(b), request R has one entry in the merged queue with a wait time determined as the maximum wait time of the different entries before merging. Merging of queues follows a strict priority policy and orders requests of the same priority according to the accumulated wait time, exemplified by the ordering of requests X and S. This wait time can be approximated *via* local time facilities for each node, *i.e.* the algorithm does not require the notion of a global logical clock and timestamps (see [5] for more details).

This priority model can be used to tune applications by identifying time-critical portions of the code and ensuring that their resource requirements are handled with a higher priority to bypass other requests. The model is also applicable to real-time applications where priorities represent the importance of tasks. The priority inversion problem can furthermore be addressed in a distributed environment by using the stack resource policy (SRP) [6] to ensure that high priority requesters will not be stalled by low-priority owners of DSM object. Finally, the priority model can be extended to user threads in the distributed system and may be used to guide task distribution such that more important tasks are placed on nodes with better performance.

6 Asynchronous Access Model

The system architecture of DSM-Threads provides the means to implement an asynchronous access model in order to enhance the concurrency of multi-threading within each node. The communication server receives messages but delegates workers to act upon these messages. A worker then manipulates the local image of the DSM space or sends messages to other nodes. However, a worker may never wait for any messages since message reception would result in suspension of the worker. Thus, message exchanges are handled asynchronously. This property increases the availability of workers and thereby the responsiveness of the system.

For example, in Figure 1 a request message is sent from a worker in node 1. The response to this message is later received by the communication server on the same node and handed down to another worker that may differ from the worker who initiated the request.

This asynchronous access model differs from traditional synchronous models [3]. In synchronous models, the initiator of the request suspends after sending the request and resumes execution only after receiving a response. As discussed earlier, a request may not be issued from the page fault handler but only from another thread due to conformance with the POSIX restrictions. The synchronous access model would thus block a worker thread till it received a response. The asynchronous access model, on the other hand, utilizes the worker resources better by handling request and response separately. The model also allows multiple concurrent requests by threads on one node to be mapped onto a single internode request. This local multiplexing of requests in a multi-threaded system applies to page-based protocols as well as object-based protocols, such as entry consistency where the requesting user thread blocks.

The asynchronous access model results in a different protocol due to the enhanced level of concurrency that arises when multiple threads may be active on one node. This protocol can be described by states of a DSM object (page or data structure). Figure 3 depicts the states and transitions for the MRSW protocol (also applicable to EC). For example, consider an object with "no access". When a thread attempts to access the object by writing to it, a request for the object is issued so that the object's state becomes "write pending". Once the object has been received for writing, it changes to the state "write access". When a read request is received for the object, the object is sent to the requester (for read access) and is then in the "read access" state on the local node. A write request in this state results in sending invalidations to members of the nonempty copy set and a transition to "read access, acknowledgment pending". Once the last acknowledgment has been received, the object has "no access" again.

Besides responsiveness, this asynchronous model has a number of additional advantages. For instance, the object remains read-accessible while acknowledgments are pending. Most of all, a request may be forwarded or stored as the next request in the distributed queue of requesters while a write request is pending. Notice that the synchronous model would not have let such requests be forwarded right away; instead, they would have been blocked till validation. Furthermore, some optimizations for transitions between read and write access are possible when the requester already has access to the object. Lowering access from write to read on read requests was

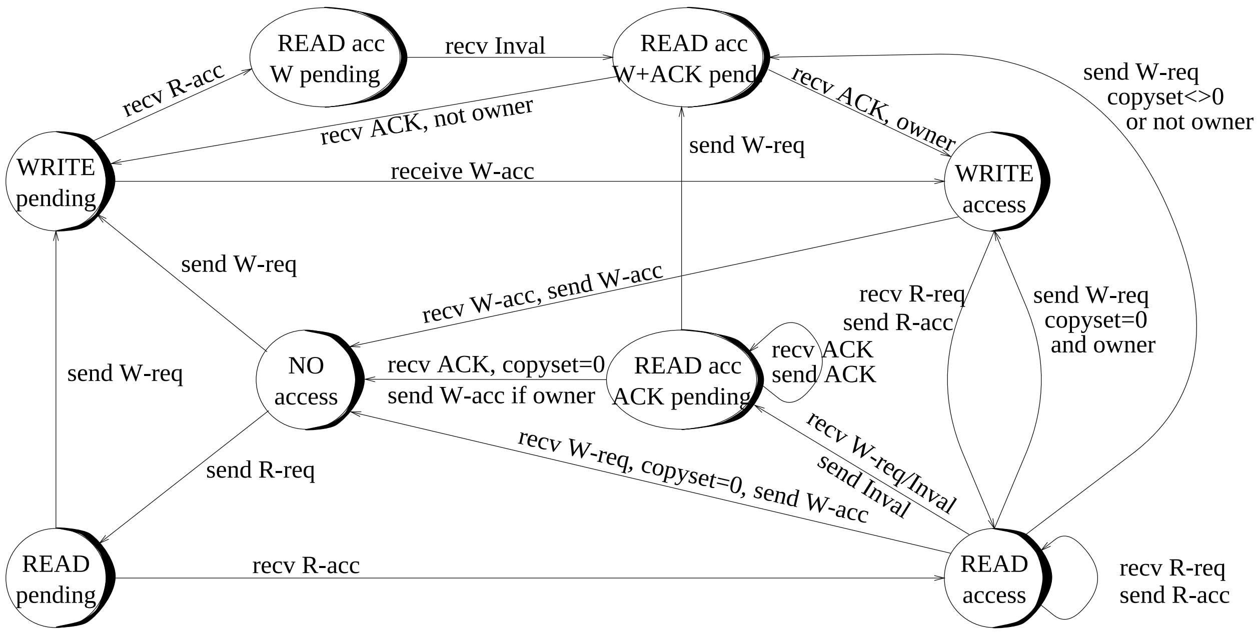


Figure 3: States and Transitions for DSM Objects

already discussed. Conversely, access can be raised locally from read to write, either right away for an empty copy set or after invalidating all copies. Invalidation, however, can only be initiated by the owner, *i.e.* the root of the copy set; otherwise, race conditions would be possible for multiple invalidations initiated from different nodes of the copy set. In Figure 3, transitions to the same state are omitted when read or write requests result in forwarding or registration as next request in the distributed queue, respectively. Table 1 depicts these cases. Neither forwarding nor registration in the distributed queue causes a state transition.

The responsiveness of a node is furthermore ensured by restricting the allocation of worker threads to tasks. Consider n page faults that occupy all n worker threads, such that the communication server cannot delegate page requests to a worker anymore. If all nodes were in such a state, the system would deadlock. Thus, one has to require that the pool of worker threads may at no time be engaged in handling tasks of only one type. We actually go one step further: First, we require that the size of the worker pool be at least the number of delegated message types (*i.e.*, task types assigned to workers). Second, we restrict the allocation of worker threads such that a worker may only be allocated a task if no worker is engaged in handling this message type yet or if the number of idle workers exceeds the number of currently unhandled message types.

7 Heterogeneous Support

DSM-Threads supports heterogeneous environments by data encoding. We take advantage of the consecutive DSM data space and extract type information about DSM data prior to program execution. Typically, two nodes engage in a bidirectional communication, for example when node A requests data (a page) that node B owns and node B responds with this data to A. On a request, information about the data representation is embedded in the the header of the communication protocol. The receiving node compares this information with its data representation. If the representation matches, raw data is sent in the response including another header to indicate once more that the same data representation is used. If, on the other hand, the representation differs,

Table 1: State Transition Table

State	Acc	Acc pend	Ack pend	send R-req	send W-req	recv R-acc	recv W-acc	recv Inval	recv Ack	recv R-req	recv W-req
0				1	2					0 forward	0 forward
1		R			2	3				1 next/fwd	1 forward
2		W				5	4			2 next/fwd	2 next/fwd
3	R				4/7			6		3	0/6
4	W									3	0
5	R	W						7		4 next/fwd	5 next/fwd
6	R		+		7				0/6	5 forward	6 forward
7	R	W	+						2/4	6 next/fwd	7 next/fwd

DSM data to be sent is encoded in the external data representation format (XDR [7]) and, conversely, decoded at the receiver. This approach ensures portability while supporting heterogeneous environments and avoids the encoding overhead when possible.

In a heterogeneous environment, the layout of DSM data also has to be considered. It is imperative that DSM data reside within the same DSM object for each node of the distributed system. This ensures that an access fault for some data refers to the same DSM object identity within the entire system. For example, DSM data (smaller than a page) resides on page n on each node but may have completely different absolute addresses, or even different offsets from the beginning of the page.

DSM objects with sizes larger than the page size (*e.g.,* large arrays) require special treatment. In a heterogeneous environment, an item (*e.g.,* element of an array) of this object may be located at the end of a page on one system while it may reside at the beginning of the next page on another system. The different offsets are caused by distinct data alignment rules. In this case, the DSM object is considered to occupy a page range. Access faults to any address within this range are linked to the DSM object, even if they occurred on different pages. Consequently, page-based protocols will treat this page range as an indivisible object, *i.e.,* the entire page range will be provided after a page fault for any of these pages.

Large DSM objects are identified by considering the data layout and alignment rules for each architecture within the heterogeneous environment. Each architecture in the distributed system is probed for its representation of basic data types and alignment rules by a generic test program. The results are merged to determine the greatest common size and alignment for each basic data type. A preprocessor parses DSM object declarations and detects whether or not an object will still fit on the current DSM page. Should the object be placed on a new page for at least *one* architecture, a page feed is generated for *all* architectures. This page feed may come prior to filling the entire page on other architectures and may thus result in a modest waste of space but on the "worst-case" architecture a page feed will be required.

In a previous design, modifications to the compiler were considered to use the greatest common size and alignment rules on each architecture. However, this approach would limit the reusability of existing object code, in particular library code with unknown sources. Furthermore, the approach may not be easily portable for different compilers and source languages. Thus, multi-page DSM objects were introduced instead of modifying the compiler.

8 Implementation

DSM-Threads has been implemented on SunOS 4.1.x for the SPARC architecture and on Linux for the ix86 architecture. This provides a heterogeneous test bed for the system. The implementation is based on FSU Pthreads [2] for both architectures. The current implementation supports page-based SRSW and MRSW protocols. Entry consistency is currently being implemented and release consistency will be supported in the future. Threads may be distributed either explicitly or implicitly by DSM-Threads. The current implementation is based on a communication layer that utilizes TCP socket streams. Table 2 depicts the overhead involved in requesting a DSM page between SPARC classics connected by a 10 Mbps Ethernet. We chose this platform as a base to interpolate the speedup for current architectures with both clock and network speeds that are an order of magnitude higher.

Table 2: Page Exchange Measurements

Description	Time
Entire Page Exchange	54 ms
Page Fault and Signal Handling	2 ms
Invalidation	4 ms
Validation	2 ms
2 TCP Messages	46 ms

The measurements indicate that the overhead of handling requests amounts to about 15% of the entire overhead of a page exchange. The main overhead was caused by the TCP socket communication. Optimizations of TCP socket connections may allow to reduce the communication overhead by about one fourth. Further improvements could be gained by using UDP connections whose raw speed is about 3.5 times faster than TCP connections. However, the actual savings are difficult to quantify at this time since additional overhead would be induced to deal with the unreliable properties of the UDP protocol. Figure 4 depicts an interpolation of our raw speed for a 275 MHz architecture with 155 Mbps network throughput. The interpolation indicates a page exchange overhead of about 2.08 ms for UDP and 3.65 ms for TCP, respectively. Keleher and Tseng report at best 1.388 ms for their CVM system in an Alpha cluster for UDP with the above characteristics [8]. Overall, a speedup of an order of a magnitude for current architectures and a faster communication protocol such as UDP should make DSM-Threads a viable option for distributed computing.

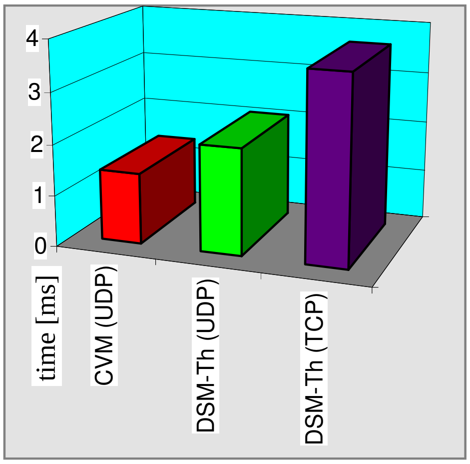


Figure 4: Interpolation for Page Exchange

DSM-Threads has also been used in conjunction with distributed Ada95 applications to support "shared passive library units" that contain data accessible by different nodes. The DSM abstraction provides an efficient way to support this Ada95 features. Regular data declarations may be mapped onto the page-based MRSW protocol. So-called protected objects can be mapped onto entry consistency to provide distributed mutual exclusion in conjunction with data migration. We have combined the runtime system of the Gnat compiler [9, 10] to demonstrate the feasibility of this approach. DSM-Threads works in conjunction with the Gnat Library for Ada Distributed Execution (GLADE). Glade provides a configuration tool, Gnatdist [11], and a runtime library, Garlic [12], for distributed execution. In addition, DSM-Threads supports the distributed access of shared data. Regular data accesses can be readily supported without changes to the Gnat system. Distributed protected objects require runtime calls to distributed mutual exclusion interfaces. These calls have to be mapped onto the corresponding DSM-Threads services [13].

9 Future Work

Entry consistency and the priority model are currently being implemented. Based on this, a comparison with other systems by evaluating common benchmarks is the foremost goal combined with an effort to tune the system, for example through the use of different communication protocols. The support of Ada95 protected objects requires work on the Gnat compiler and the Glade runtime system to integrate our work. A Solaris 2.x implementation using native threads is being planned to experiment with shared-memory multi-processor nodes within the system and identify portability problems with other Pthreads libraries due to the laxity of the POSIX standard. Furthermore, support for dynamically allocated DSM objects is being added.

10 Related Work

The decentralized dynamic manager algorithm by Li and Hudak [3] was redesigned and extended to serve as an asynchronous model. The token-based distributed mutual exclusion protocol by Naimi *at. al.* [14] inspired the prioritized protocol that differs from the original work by the usage of multiple local queues and local time facilities. Ariadne implements migratable threads for shared-memory and distributed system such that the thread migrates to the data location on an access fault [15]. The Millipede system also supports multithreading, migration and page-based DSM but uses Windows-NT as a base rather than POSIX [16]. Entry consistency was adopted based on the Midway system [4]. Filaments employs a set of lightweight worker threads to act upon remote processing requests quickly but is based on compiler modifications to optimize runtime data placement [17]. Chant [18] and Nexus [19] support distributed threads. These systems can be enhanced by distributed shared memory but this feature is not explicitly supported. Nexus' global pointers (GPs) are similar to our support for recognizing heterogeneous connections built into the communication protocol.

11 Conclusions

DSM-Threads provides an environment to support concurrent programming with shared-memory in a distributed environment *via* distributed shared memory. This paper relates the design of DSM-Threads to the implementation choices and raises several issues unique to the system. The DSM system architecture exploits multi-threading for each node in the distributed environment by allocating runtime tasks to worker threads. This approach is enhanced by utilizing an asynchronous communication model. This model facilitates the multiplexing of requests of a multi-threaded node. In addition, it prevents workers from engaging in blocking operations and thus increases the responsiveness of each node. Priority support for a set of consistency models represents a novel feature for DSM systems that are based on decentralized distributed protocols. The choice of consistency models and the ability to define priorities facilitate the migration from POSIX threads to DSM threads and can be used to tune applications. Furthermore, heterogeneous environments are supported without compiler modifications to enhance the portability of the system. A first implementation shows that the DSM system architecture contributes about 15% of the overhead involved in DSM data synchronization while the remaining overhead is due to the underlying communication protocols. The implementation has been integrated with an Ada95 runtime system to demonstrate its ability to support higher-level languages. DSM-Threads is, to our knowledge, the first runtime system to support distributed threads on top of POSIX Threads via distributed virtual shared memory.

References

[1] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[2] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[3] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions of Computer Systems*, 7(4):321–359, November 1989.

[4] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.

[5] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *Workshop on Parallel and Distributed Real-Time Systems*, April 1997.

[6] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[7] Sun Microsystems, Inc., Mountain View, California. *External Data Representation Protocol Specification*, February 1986. Part Number 800-1324-03.

[8] P. Keleher and C. Tseng. Enhancing software dsm for compiler-parallelized applications. In *International Parallel Processing Symposium*, pages 490–499, 1997.

[9] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[10] E.W. Giering III and T.P. Baker. Features of the GNU Ada runtime library. In *Proceedings of TRI-Ada '94*, November 1994.

[11] Y. Kermarrec and L. Pautet. Gnatdist: A configuration language for distributed Ada 95. In *TRI-Ada*, 1996.

[12] Y. Kermarrec, L. Pautet, and S. Tardieu. Garlic: Generic Ada reusable library for interpartition communication. In *TRI-Ada*, pages 263–269, 1995.

[13] F. Mueller. Supporting Ada 95 passive partitions in a distributed environment. In *Ada Europe*, June 1997.

[14] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing*, 34(1):1–13, April 1996.

[15] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting threads migration. *Software Practice & Experience*, 26(3):327–356, March 1996.

[16] A. Itzkovitz, A. Schuster, and L. Shalev. Supporting multiple parallel programming paradigms on top of the millipede virual machine. In *Workshop on High-Level Programming Models and Supportive Environments*, pages 25–34, April 1997.

[17] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, August 1996.

[18] M. Haines, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Supercomputing*, pages 350–359, November 1994.

[19] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.