Heaps and Stacks in Distributed Shared Memory

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

Abstract

Software-based distributed shared memory (DSM) systems do usually not provide any means to use shared memory regions as stacks or via an efficient heap memory allocator. Instead DSM users are forced to work with very rudimentary and coarse grain memory (de-)allocation primitives. As a consequence most DSM applications have to "reinvent the wheel", that is to implement simple stack or heap semantics within the shared regions. Obviously, this has several disadvantages. It is error-prone, time-consuming and inefficient. This paper presents an all in software DSM that does not suffer from these drawbacks. Stack and heap organization is adapted to the changed requirements in DSM environments and both, stacks and heaps, are transparently placed in DSM space by the operating system.

1. Introduction

Software-based distributed shared memory (DSM) systems do usually not provide any means to use shared memory regions[1] as stack or heap memory. This situation is at least surprising, because first, stacks and heaps are fundamental and agreed concepts, and second, stack and heap handling has been highly optimized in virtually any operating system (OS), runtime environment and compiler. Disregarding these fundamental concepts DSM users have to work with rudimentary coarse grain memory allocation primitives, such as e. g. `Qk_create_region` in Quarks [Kha96]. Obviously, users of such DSM systems are forced to implement some allocation and de-allocation algorithms one way or the other to be able to use shared regions for application-level objects, such as a simple array of integer variables.

[1] the term memory "region" denotes a complete interval of virtual addresses

This recurring "reinvention of the wheel" has several disadvantages:

1. It is error-prone: implementing an efficient memory management allocation strategy is all but simple.
2. It is time-consuming: the DSM application developer has to spend a lot of time just for developing basic memory management concepts that would usually be provided by the OS.
3. Performance penalties: standard stack and heap handling strategies are highly sophisticated and optimized to achieve peak performance. An application integrated workaround will in general result in significant performance losses.

1.1. Are stacks and heaps within DSM needed?

We state, that a sound memory management support is mandatory for DSM to lift its relevance and applicability beyond the scope of academic interest.

One could argue, that stacks and heaps are actually not needed within DSM space. There are at least two possible reasons for this consideration. First, the amount of shared memory is usually very small and only used for a certain purpose, such as storing the elements of two matrices that are to be multiplied in parallel. Second, the placement of shared objects within the DSM is crucial for the performance of DSM based systems. As the correct placement depends on the application, it must be performed by the application itself, anyway and not by a general stack or heap algorithm.

The first argument does not hold in general but is based on observations of rather artificial experiments that are used to demonstrate speed-ups in scientific papers. We are confident, that distributed applications beyond traveling salesman and matrix multiplication will have more complex access patterns to shared memory and therefore expose a

stronger demand for support for dynamic memory allocation.

The second point addresses indeed a very hard problem, that still remains unsolved, although several projects, such as Orca [BK93] and TreadMarks [ea96, ea99], successfully investigated different ways to automate the placement of objects within shared space. We also investigated different OS level strategies to automatically find suitable placements for shared objects by combining runtime and compile-time techniques [EP99]. The techniques developed alleviate the problem but further advances in this field are certainly needed. Anyway, pushing the problem to the application level is no solution.

1.2. Reasoning

The are presumably three reasons for the absence of stack and heap alike managed regions in DSM systems:

1. As argued in section 1.1, the automatic placement of objects in shared space is non-trivial.
2. Heap memory is usually allocated via system calls such as `brk` (UNIX) and dynamic stack space adaption is managed by the OS kernel. Therefore, DSM systems implemented as runtime libraries can not easily be placed underneath the level of stack and heap management so that all stack and heap operations are automatically performed under control of the DSM system.
3. The conventional memory model of single threaded systems only distinguishes a single heap and a single stack within one address space. Accordingly, conventional heap and stack management techniques are tied to this memory layout model, which is unsuitable for distributed and parallel computations. Hence it is not possible to reuse e. g. a heap management library without significant modifications.

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].

Ivy [Li86] was the first page-based DSM, followed by improvements such as Mirage+ [ea94], TreadMarks [ea96] or Odin [Pea96]. Due to the possibility 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 a library; no page-faults are triggered. False sharing is circumvented but frequent writes degrade system performance. Munin [Car95] respects the size of individual objects, too and a different coherence protocol can be chosen for each object.

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 via awkward additional DSM services. Thus stack pointers and heaps have to be re-implemented 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. Similarly, object-based systems, e. g. Munin [Car95], require the programmer to explicitly mark sharable objects. Again, there is no dynamic transition from private to shared. The only exception so far is the DSM system Shadow [GPR97], which was built to efficiently manage shared objects that are located on thread stacks.

Thus, the memory models of software-based DSMs can be regarded as immature, because of a lack of support for dynamic allocation and the absence of an internal stack and/or heap alike organization of shared regions. Notice, hardware-based DSMs such as [SH98] do usually not suffer from this problem because DSM handling is performed mostly below the OS level.

3. Murks' Basics

The remainder of this paper presents the memory organization of our software DSM system *Murks* that does not suffer from the drawbacks depicted above. The architecture of Murks, some implementation details, and performance issues have already been discussed in [PR01]. In this paper we focus on its stack and heap management which has been adapted do DSM environments. Hence, this section will only provide basic information about Murks and our design philosophy that is needed to understand the remainder of this paper.

Murks is a page-based, sequentially consistent [Lam79] and POSIX threads compatible DSM system providing full transparency concerning physical distribution. 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 OS. In order to simplify coherence maintenance and to allow simultaneous reads at multiple hosts, Murks uses a multiple reader/single writer policy. The MMU page-fault mechanism is used to detect accesses to locally non-available data. Due to the sharing of rather coarse grained memory pages, the coherence protocol relies on invalidation instead of update propagation. Murks is im-

plemented as a number of cooperating server processes, a client runtime library and a Linux kernel module.

4. Memory Management

In centralized OSs heap memory is usually allocated via system calls (`brk`) and library wrappers. Stack growth and contraction is handled by the OS kernel. Clearly, parallel tasks within a single address space (i.e. threads[2]) need separate stacks. In contrast to single threaded applications where a collision of heap and stack occurs only due to memory exhaustion as shown in figure 1 (a), thread stacks and the heap may collide although a large amount of memory is still available (figure 1 (b)).

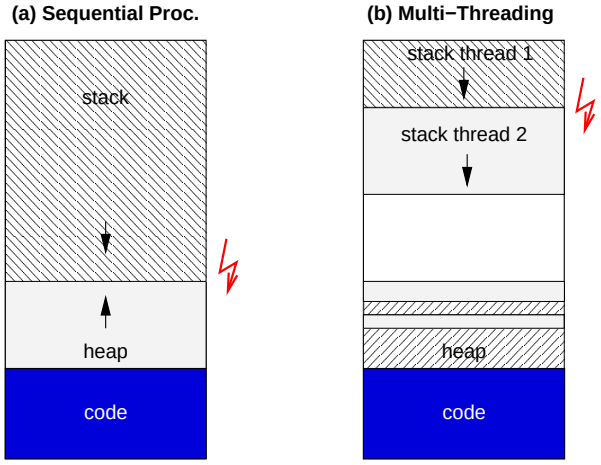

Figure 1. VA partitioning

For non-distributed, multi-threaded systems, this problem has been solved in [Piz99]. Each thread has its own stack and heap organized as sets of memory segments (see figure 2).

With a few extensions, this technique can similarly be used to introduce shared stacks and heaps in distributed environments.

4.1 Dynamic Mapping

In order to allocate a memory region to be shared among multiple distributed threads, the region must solely be mapped by a thread of the distributed system by calling `murks_mmap`. This function executes a local `mmap` for the region, registers it at the Murks server and thereby marks it as globally accessible.

Therefore, to establish stacks and heaps in shared space we have to assure that all requests for memory are based on

[2] including distributed threads

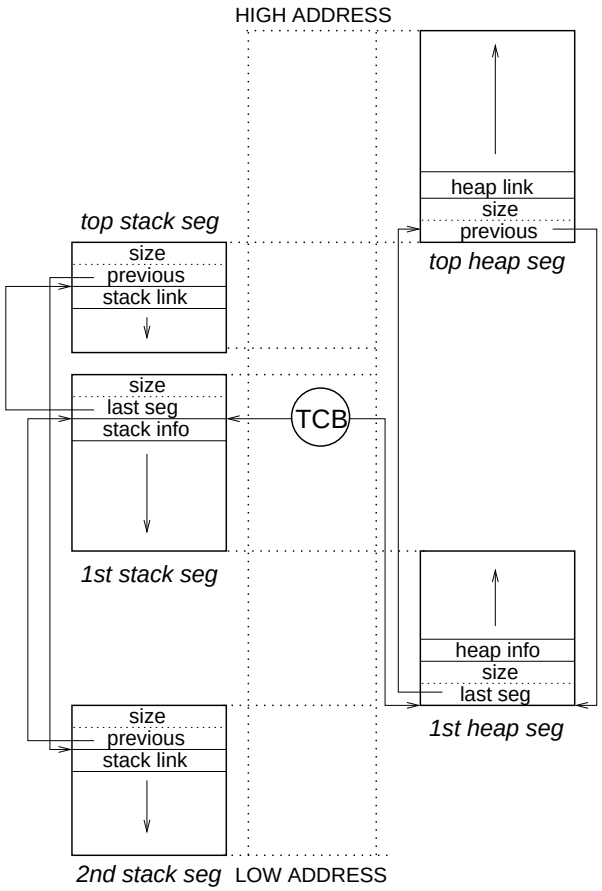

Figure 2. Memory Layout

`murks_mmap`. First, the call to `brk` within the heap library has to be replaced with a call to `murks_mmap`. Second, stack growth management has to be withdrawn from the OS kernel. This is accomplished by allocating stack space for each thread via `murks_mmap` guided by runtime checking for stack over- and underflows at entry and exit points of call frames (see section 4.3).

A shared memory region may be unmapped via `murks_unmap` by any thread of the distributed system at any time. Even sub-intervals or multiple consecutive regions may be unmapped with a single request. The unmap request is sent to the server which splits or coalesces regions if necessary, and distributes invalidations to all nodes within the copy set.

4.2. Segmentation

A *segment* is a complete interval of virtual addresses encompassing at least one memory region. A *segment stack* is a stack of dynamically pushed and popped segments. Additionally, the top most segment may grow and shrink.

Notice, addresses within a segment stack need neither be monotonous nor linear.

Every segment has a header specifying its size and a link. The header is placed at the highest address in case of stack, respectively the lowest address in case of heap to enable linear segment extensions for downward growing stacks and upward growing heaps. In case of an overflow of the top segment, it is first tried to extend the top segment by requesting a connecting region from `murks_mmap`. Otherwise, a *non-linear extension* is performed by pushing the region received as the new top segment. An underflow occurs, if the stack pointer or the heap limit drop below (above for heap) the top segment. *Reductions* triggered by underflows can as well be linear (shrinking top) or non-linear (top is popped).

Figure 2 illustrates stack and heap space based on segment stacks. Management data usually kept in a static data part, e.g. heap library variables, are placed in the information part of the bottom segments.

4.3. Stack Adaptation

Segment stacks allow to lazily adapt memory consumption without a rigid limit. Each thread is started with a single stack segment whose size is determined at compile time. At runtime, segment crossings are monitored and the usually linear stack space becomes eventually split to fit on separate segments. Only three possibilities for segment crossings must be considered. First, when *a call level is entered*, the stack pointer (SP) is decremented (downward growing stacks). Second, *dynamic stack objects*, such as fields with statically unknown ranges, are allocated by decrementing SP. While these two operations may cause overflows, *leaving a call level* is the source of underflows.

A hardware integrated compare logic checking SP against segment limits would be desirable but is not available. Hence, to be efficient, monitoring must be prepared by the compiler with inlining code into the prologue and epilogue of subprograms. Additionally, stack addressing had to be changed. Usually, a single frame pointer points in between two frames. Negative offsets reference local objects, while arguments are found via positive offsets. Now, the size of the possible gap between arguments and locals is statically unknown. This requires an explicit argument pointer. Furthermore, the activation frame layout was extended with a flag determining whether the frame has caused a non-linear extension to be able to detect underflows. While overflows are checked against the current stack limit recorded in the *thread control block* (TCB), a thread specific datastructure, underflows are detected with help of this extension flag. The modified stackframe layout for a non-linear stack is shown in figure 3.

Correcting an overflow requires calls of subprograms

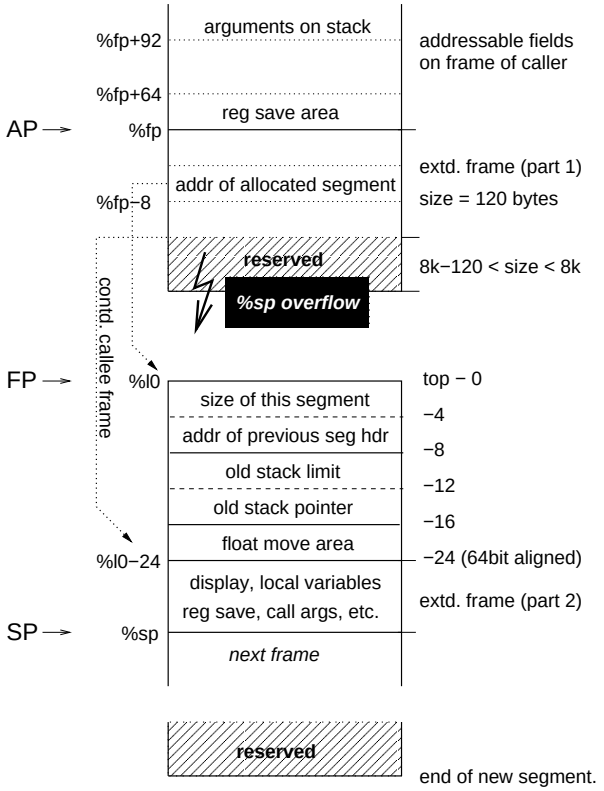

Figure 3. Non-linear stack extension

consuming further stack space. Therefore a small *reserved area* at the end of the current stack segment is needed for the stackframes of the subprograms. Our implementation ensures, that at least the size of the reserved portion (currently 8k) minus the minimal frame is available for the overflow handler. In case of non-linear extensions, the reserved area is temporarily lost. Linear extensions simply move the reserved area to the new end of the segment without losses.

All of these modifications were made to the low-level back-end of the GNU `gcc` compiler. Among the benefits are support for many languages (C, C++, INSEL, etc.) at once and compatibility with all compiler optimizations such as function inlining or leaf functions.

4.4. Heap Adaptation

As a starting point for the implementation of the DSM heap we selected D. Lea's freely available memory allocator `G++ malloc` [Lea96]. It structures heap space into free and allocated *chunks*. A special free chunk, called *top chunk* (TC), is used to grow and shrink the heap. It is split and coalesced as chunks are (de-)allocated at the top end of the heap while being increased and decreased at the upper end with calls to the `murks_mmap` and `murks_munmap`

functions. In contrast to stacks, the separate management

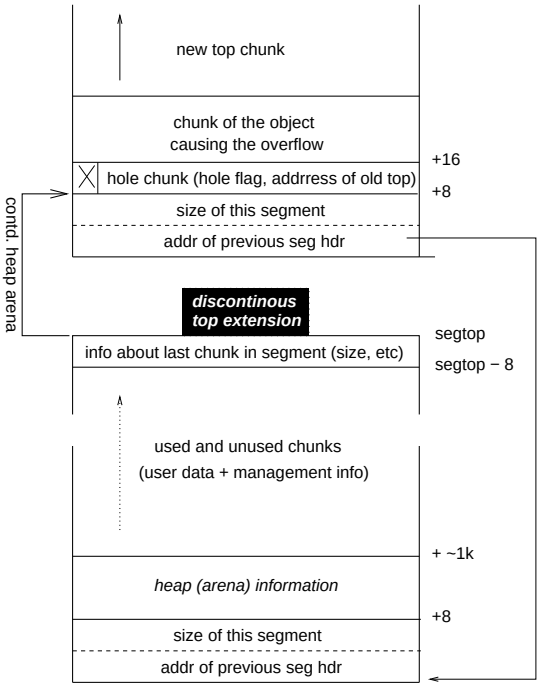

Figure 4. Non-Linear Heaps

of each application-level object in a chunk allows to easily spread a heap across segments, because splitting can be performed between arbitrary chunks.

Several modifications were made to support positive or negative *holes* caused by non-linear extensions (see figure 4). If TC is non-linearly extended, the current TC is converted into an ordinary free chunk. Its chunk information is placed at the highest address of the old top segment. Above the segment header of the new segment, a special *hole chunk* is installed and the allocation causing the overflow is performed. The remainder of the segment is used as the new TC. The hole chunk serves two purposes. It stores the information about the old TC and it has a flag set, that prevents this chunk from being coalesced with other chunks than the TC.

5. Performance

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. As we rely on common-off-the-shelf hardware Murks' DSM performance is similar to other page-based, sequentially consistent DSMs and their performance has already often been measured.

Here, we concentrate on the impact of the changes made to the stack and heap organization. Obviously, the sophisticated memory allocator `G++ malloc` and compiler supported stack optimization deliver superior performance compared to application integrated handling of data objects. We therefore only discuss potential degradations stemming from our modifications.

5.1. Speed

Concerning heap operations (`malloc`, `free`) using `murks_mmap` instead of `brk` does hardly have any impact on the performance of the memory allocator. This is especially true, because (de-)allocations of heap regions are rather rare. Most of the dynamics happens within heap regions without being affected by our changes.

Dynamic stack growth is more interesting, because stack growth must be checked with each entry and exit from a call frame. Still, the computational costs for monitoring stack overflows [Piz99] induced by our implementation based on inlined code is low. In the average case, only 8 additional machine instructions (see figure 5) are needed per call frame. Tests with a simple parallel prime generator indicate an insignificant overhead (40.3 versus 40.5 seconds).

Figure 5 lists the stack checking code used on Sparc V9. In this example, the frame size is 384 bytes. Line (1) of the prologue clears the extension flag, FP is assigned the value of AP (2), and the effectual limit is fetched from the TCB (3). If the SP is below the limit, nothing is left to do (4,5). Otherwise, the stack limit is cleared ($\equiv$ maximum) to avoid recursion (6) and the overflow handler is called (9) after shrinking to the minimal frame (10). The handler returns zero in case of linear extensions which is checked in (11). If linear, then only the SP is reset to the value before the handler was called (14,8). If non-linear, the stack limit and SP are written to the new segment (15,16) and the segment address is written to the extension flag (18), before the frame space is moved to the new segment by setting FP and SP (18,19). Lines 1–3 of the epilogue check whether the current frame caused a non-linear extension by comparing the extension flag with zero. If yes, then the current limit is set ineffective (5), and SP is reset (7), before the underflow handler is called (6), and the stack limit becomes reset (8).

5.2. Space

Non-linear extensions lead to internal fragmentation. Let f be the average frame size, r the size of the reserved area, and s the average segment size. Following formula approximates the internal stack fragmentation, if every extension was non-linear:

$$F_{avg} = \frac{r + ((s - r)\ mod\ f)}{s}$$

```
    save %sp,-384,%sp
 1) clr [%fp-8]
 2) mov %fp,%l0
 3) ld [%g3+12],%l1
 4) cmp %sp,%l1
 5) bgeu .prolog_end
    nop
 6) clr [%g3+12]
 7) mov 384,%o0
 8) mov %sp,%l2
 9) call OVERFLOW
10) add %fp,-120,%sp
11) cmp %o0,%g0
12) bne .non_linear
    nop
13) b .prolog_end
14) mov %l2,%sp
.non_linear:
15) st  %l1,[%o0-12]
16) st  %sp,[%o0-16]
17) st  %o0,[%fp-8]
18) add %o0,-8,%l0
19) add %l0,-384,%sp
1) ld [%fp-8],%o0
2) cmp %g0,%o0
3) be .epilog_end
   nop
4) ld  [%o0-12],%l1
5) clr [%g3+12]
6) call UNDERFLOW
7) ld  [%o0-16],%sp
8) st  %l1,[%g3+12]
```

Figure 5. Stack check prologue and epilogue

$$8k - 120 < r < 8k$$

For $f = 256, r = 8192$, and $s = 32k$ internal fragmentation would be 25%. Non-linear extensions are problematical in two ways. First, they may cause noticeable fragmentation, which can be optimized by choosing adequate segment sizes. Second, in contrast to linear extensions, non-linearly extended segments become freed as soon as the call-level causing the extension is left and might already be reallocated with the next call leading to unfavorable *thrashing*. This situation is avoided by exploiting the region allocator to provide regions at preferred addresses.

6. Conclusion

We state, that DSM research should pay more attention to the programming model from a software engineering point of view instead of solely concentrating on performance issues. The transparent integration of the well-known stack concept and a heap memory allocator into our software DSM Murks described in this paper aims at this direction. The benefit of this strategy is, that the user of the DSM is freed from the error-prone and time-consuming task to organize application level objects within coarse grain memory regions. We sketched the implementation of the stack and heap concept in DSM space and we showed, that the modifications needed to adapt stacks and heaps to DSM environments do not necessarily result in performance degradations. The techniques described in this paper can easily be transfered to other DSM systems to improve their usability as well as maintainability of DSM based distributed applications.

References

[BK93] H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *OOPSLA'93*, pages 162–177, September 1993.

[BK98] B. Buck and P. Keleher. Locality and performance of page- and object-based dsms. In *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.

[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.

[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.

[GPR97] S. Groh, M. Pizka, and J. Rudolph. Shadow stacks—A hardware-supported DSM for objects of any granularity. In *ICA3PP'97*, December 1997.

[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.

[Lea96] Doug Lea. A memory allocator. World Wide Web, December 1996. http://g.oswego.edu/dl/html/malloc.html.

[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.

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

[Piz99] Markus Pizka. Thread segment stacks. In *Proc. of PDPTA '99*, Las Vegas, NV, June 1999.

[PR01] M. Pizka and C. Rehn. Murks – a POSIX threads based dsm system. In *PDCS'01*, pages 642–648, Anaheim, CA, August 2001.

[SH98] M. Schulz and H. Hellwagner. Extending NT virtual memory by SCI-based hardware DSM. In *Proc. of the 2nd USENIX Windows NT Symposium (WIN_NT-98)*, pages 169–169, Berkeley, August 3–5 1998. USENIX Association.

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