SHADOW STACKS – A HARDWARE-SUPPORTED DSM FOR OBJECTS OF ANY GRANULARITY

S. GROH, M. PIZKA, J. RUDOLPH
Technische Universität München,
80290 München, Germany
E-mail: {groh,pizka,rudolph}@informatik.tu-muenchen.de

This paper presents a new Distributed Shared Memory (DSM) management concept that is integrated into a scalable distributed virtual memory management technique and circumvents false sharing while still preserving simplicity to the application level. Objects defined as usual by variables in the declaration part of functions are made sharable among threads executing in the distributed environment. These objects of varying granularity and with different consistency requirements are managed separately to avoid false sharing. Consistency is enforced at runtime by a distributed manager-agent architecture, that supports automatic and dynamic selection of an adequate coherence protocol per object. To provide efficiency, the implementation of the *Shadow Stacks* concept is based on the exploitation of the page fault mechanism provided by of the shelf hardware.

1 Introduction

The relevancy of distributed computing in practice is still far behind the potentialities of nowadays available distributed computing power, provided by powerful workstations and high-speed interconnection networks. Obviously, the reason is tremendous complexity. Distributed computing either burdens the programmer with additional concepts and their effects or it demands tasks from the resource management system that a hard to fulfill. We state, that the distributed nature of the underlying hardware configuration has to be hidden as far as possible instead of aggravating distributed programming, which is obstructive for making distributed computing attractive beyond the scope of academic or research interest. In addition to the requirement of *simplicity*, distributed *performance* has to be convincing. Execution speed has to be reasonable compared to sequential and centralized software solutions, as well as it should provide speed-ups if additional computing resources are available.

Peak performance on distributed hardware platforms can be reached by using explicit message passing[1,2]. We argue, that writing parallel or even distributed programs with explicit message passing is a cumbersome and difficult task, contradicting the desired simplicity. The shared memory paradigm obviously provides an easier to use abstraction, since it moves the task of communication from the application-level to the system-level. The idea of a resource management system, that enforces the abstraction of a shared memory in a distributed environment (Distributed Shared Memory) is not very new, but since the first DSM implementation by Li[3] in 1986, the performance of most DSM systems is still unsatisfactory[4]. Performance problems of DSM implementations mainly arise from false sharing in page-based systems, missing hardware support to detect access violations on a fine-grained basis and communication overheads originating in inadequate coherence protocols.

False sharing can be prevented by reducing the size of DSM management units down to single application-level objects, such as simple integer variables. Communication costs can be reduced in several ways. Objects have to be grouped dynamically into larger units of transportation [5] and multiple coherence protocols have to be provided by the DSM to meet diverging consistency requirements. Furtheron, an implementation of a DSM concept has to consider mechanisms provided by the hardware. The common existing page-fault mechanism should be exploited to detect accesses to locally unavailable objects efficiently.

In this paper we present the concept and implementation of a decentralized distributed virtual memory management that provides simplicity as well as efficiency. The integrated DSM facility takes advantage of hardware properties to identify accesses to locally unmapped objects while still being able to manage objects of any size separately.

In the next section the underlying programming and resource management model will be presented. Section 3 presents our decentralized and scalable virtual address space (VA) management, that is based on a manager-agent architecture. In Section 4 we will elaborate on the concept and the implementation of Shadow Stacks in detail, before we compare it with other DSM systems in 5 and conclude with information on the state of the project in section 6.

2 System Model

The idea of Shadow Stacks is based on a new distributed system architecture[6], comprising language concepts and a scalable and adaptive resource management architecture. This section gives a brief overview about the main concepts as far as they are relevant for the subsequent sections.

2.1 Programming Model

Shadow Stacks are based on two major programming concepts. First is multithreading enhanced with a termination dependency, determining that no thread is deleted as long as child threads exist. Second is nesting of functions. Nesting together with termination dependent threads induces a hierarchy on the threads executing in the system. In addition to its own local data, a thread may also access non-local data, belonging to the stack frame of one of its static predecessors. It is important to notice, that there is no "global data". Objects declared in the main function executed by the first thread started, are not placed in a special global data section, but on the stack of the main thread just like other volatile objects. Therefore, globally visible objects do not require any special treatment. Accesses to non-local data are implemented using common techniques such as displays[7] or a static chain of frame pointers. Hence, sharing of data in the distributed system is expressable in a natural and simple way. Instead of employing sharable segments, a thread t allocates local objects o_i on its stack frame and may create further threads t_j executing nested functions. All objects o_i are accessible from t_j since they are visible from within the nested functions. Therefore, all objects on a thread stack are sharable among a family of child threads. Synchronization has to be done using common primitives, such as semaphores.

```
void main(void) {
    int a[3][100];                    /* a sharable object local to main       */
    void example(int space)  {        /* nested; initializes part of the array */
        int counter;                  /* object local to example               */
        for(counter = 0; counter < 100; counter ++)
          a[space][counter] = space;  /* access to shared non-local object "a" */
    }
    dist_thr_create(example, 0);      /* creation of child thread 1 */
    dist_thr_create(example, 1);      /* creation of child thread 2 */
    dist_thr_create(example, 2);      /* creation of child thread 3 */
}
```

Figure 1: Sample Program

The Shadow Stacks management environment has to fulfill two tasks. First, a mean to create termination dependent threads in the distributed environment has to be provided and second, accesses to non-local objects has to be enabled. Distribution is completely hidden to the programmer. He is provided with a virtual parallel shared memory machine.

With the extension of nesting as available in GNU-C[8] and additional libraries for distributed multi-threading, the programming language C meets the requirements of this programming model. In the example given in figure 1, a two-dimensional array is initialized with the number of the column. Each column is initialized in parallel by a separate thread. The `dist_thr_create` call starts a thread on a workstation selected by the load management system. The key features illustrated are the simple declaration of the sharable array and the nesting of the child thread function.

2.2 Manager Agent Concept

To enforce transparent, scalable and adaptable distributed resource management, we developed a multi-agent management architecture. To each thread, exactly one abstract manager is associated, responsible for performing thread-specific resource management, that is to fulfill all requirements arising from the computation of the thread. Besides fundamental tasks such as allocating memory for local objects, the abstract manager also has to maintain consistency of replicated objects, enforce access restrictions or perform load balancing. Conflicts such as stack overflows arising from different managers performing their tasks in parallel are solved by communication and cooperation between managers according to application-level structural dependencies between the threads, like for example the ancestor relationship between threads and child threads. In addition to cooperation, these abstract managers act autonomously as far as possible and may migrate at runtime. Autonomous acting, cooperation and mobility characterizes them as manager *agents* (m-agents). This management scheme is scalable, does not have a potential central bottleneck and is adaptable, since resource management is based on the requirements of application-level objects.

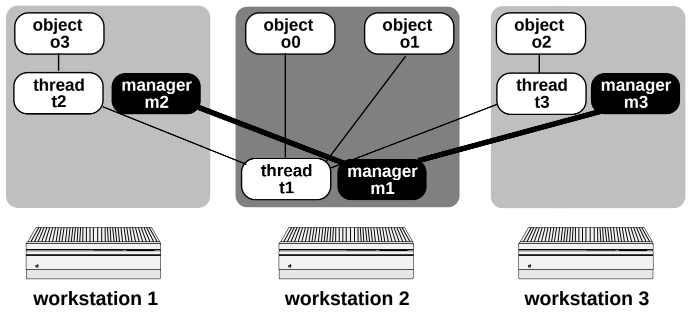

Figure 2: Manager Agents

The key design issue for the efficient implementation of this multi m-agent concept is to consider and integrate all levels of resource management. At first, management decisions are made by the compiler and must therefore be considered as m-agent functionality. After compilation, run-time management has to be performed. Run-time functionality of our m-agents is implemented in a non-uniform way, dependent on decisions made by the compile-time m-agent portion. To reduce management overhead, the run-time portion of a basic m-agent might solely consist of code generated by the compiler and might be as simple as only providing stack handling. More complex management capabilities as required for example by the Shadow Stacks concept is implemented by complex management objects, that might be executed by additional threads. Besides compile-time and run-time support, functionality provided by the micro-kernel is also tightly integrated as being a part of m-agents.

3 Distributed Virtual Address Space Management

An important issue in distributed computing is the organization of the virtual address space (VA). By providing a DSM the easy to use paradigm of a shared or even single VA can be applied to distributed processing. Besides others, a major problem arising from the single VA in a distributed environment is to find a scalable organization without a central bottleneck.

We propose a solution based on a combination of the programming model and the m-agent concept introduced above. The hierarchy of termination dependent threads establishes a hierarchy of m-agents. This hierarchy is used to split the task of VA management among the m-agents.

3.1 Requirements

In common operating systems, each application is provided with a separate VA. However, in order to share memory objects and libraries they have to be mapped into each address space, which produces additional management overhead. By employing a single VA for all applications this problem is evaded. Another advantage is the ability to identify each memory object by using its unique address within the single address space instead of having to separately generate unique object identifiers for this purpose.

With a single VA for all applications the issue of how to enforce access restrictions becomes important. Protection domains should be easy to handle for programmers while being powerful enough to protect applications from each other or even smaller units. Also the size of memory units that have to be managed can differ. For example, not every thread needs the same amount of stack space. To have the possibility to avoid wastage of memory due to fix sized management units, the VA management has to be able to flexibly handle memory regions of varying sizes. Additionally, the virtual address of a memory object should be independent from its physical location in the system, i.e. no information concerning the hardware used to hold the object (like e.g. the id of the workstation) should be coded in the virtual address. This is necessary in order to have the possibility to easily migrate objects from one node to another. Finally, it is obvious, that the VA management in a distributed system has to be decentralized. A single central organizing entity would be a bottleneck for the whole system, since every single memory request would have to be handled by this central unit.

3.2 Realization with Manager Agents

The task of managing the VA is divided among all m-agents in the system. Each m-agent is responsible for managing some regions of the VA and has at the same time a complete view on the distributed VA. The views of the m-agents on the VA differ in the classification of VA regions as one of four kinds:

own thread All address regions marked *own thread* are allocated and used by the m-agent itself or the associated thread.

free All address regions marked *free* are under control of the m-agent, but not used yet. This means, the m-agent may use these regions for its own purposes, for the needs of its thread or to transfer them to other m-agents.

delegated Address regions marked with a m-agent id are delegated to another m-agent. This means, the m-agent was originally responsible for these regions, but currently a different m-agent has control over them. As soon as one of these m-agent terminates, the region is changed from *delegated* to *free*.

no control All address regions marked with *no control* are not under the control of the m-agent.

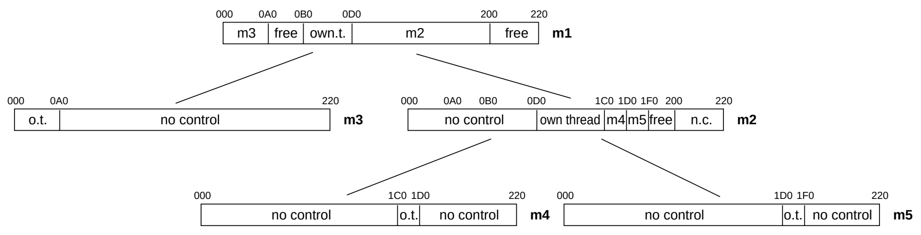

Figure 3: Distributed Virtual Address Space Management with M-Agents

Figure 3 is a brief example to explain the concept. M-agent *m1* has two child m-agents *m2* and *m3*, that in turn have two child m-agents *m4* and *m5*. M-agent *m1* is responsible for the whole address space. It has delegated the management of the first region (`000 - 0A0`) to the m-agent *m3*. The next region (`0A0 - 0B0`) is free for use. Region 3 (`0B0 - 0D0`) is used for the the data of the m-agent as well as for its corresponding thread. Region 4 (`0D0 - 200`) is delegated to m-agent *m2* and the last region (`200 - 220`) is free. This is the view of m-agent *m1* on the system. M-agent *m3* has a different view. For *m3* the whole address space is divided into two regions: one between `000 - 0A0` and the other between `0A0 - 220`. The first region is under its own control. In this case it uses the whole region for its own needs. The second region is not under its control. The same holds for m-agents *m2, m4* and *m5*. Each of these m-agents has its own view of the whole address space.

Each time a m-agent has to allocate additional memory and the size of its regions labeled *free* is not large enough, it can either ask its father up the hierarchy to delegate additional memory to it or its children to give back some of the memory it delegated to them.

Obviously the presented concept complies with the requirements of a single VA and a distributed management without bottlenecks. The need to efficiently store the different views of each m-agent and the possibility to allocate memory regions of any size seem to contradict. However, both requirements can be fulfilled by using a flexible data structure, that is based on *guarded page tables*[9]. The physical location of a memory object is not coded in its virtual address; its location is stored in the data region of the m-agent.

Note, that until now, the decentralized management scheme holds information about the allocation of virtual addresses but not about accessibility of regions. The additional information is provided by further dividing *no control* regions into *access* and *no access* regions. In analogy to *delegated* regions, the id of the controlling m-agent is also stored for *access* regions. A programmer may define *access* regions easily by employing the nesting facilities of the programming model proposed.

4 Shadow Stacks

To provide DSM functionality for objects of any granularity while still being able to exploit efficient hardware mechanisms, access to shared objects has to be indirect. In general, indirection gives us the possibility to check accesses to objects, which is essential to keep track of the location of the objects in the distributed system. In the Shadow Stacks DSM, indirection is used to activate the page fault mechanism of the hardware in case a referenced object is not locally available.

Objects placed on heap are usually addressed via pointers providing the required indirection in a natural way. In contrast to heap objects, for objects

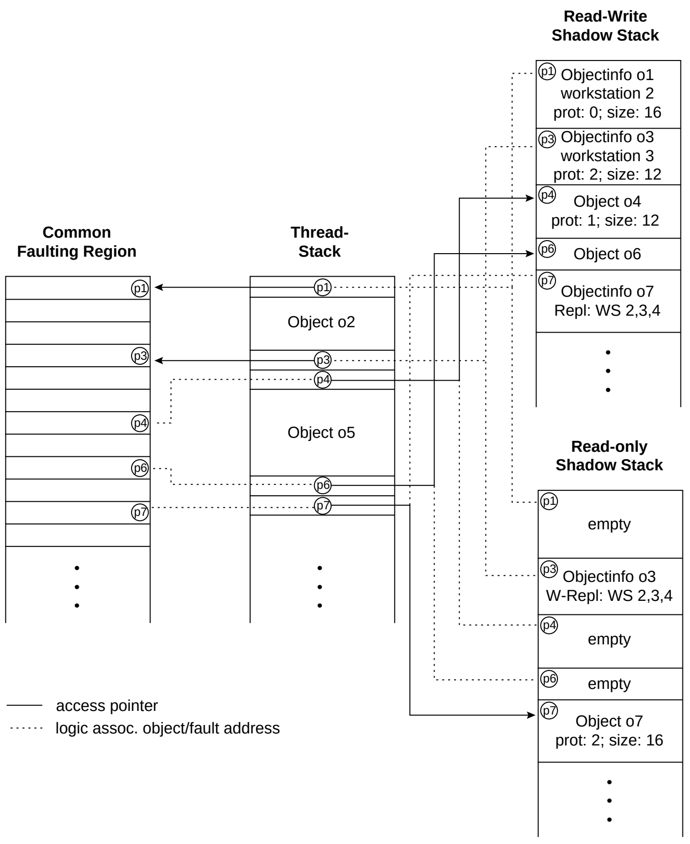
Read-Write Shadow Stack
p1 Objectinfo o1 workstation 2 prot: 0; size: 16
p3 Objectinfo o3 workstation 3 prot: 2; size: 12
p4 Object o4 prot: 1; size: 12
p6 Object o6
p7 Objectinfo o7 Repl: WS 2,3,4
Common Faulting Region
p1
p3
p4
p6
p7
Thread-Stack
p1
Object o2
p3
p4
Object o5
p6
p7
Read-only Shadow Stack
p1 empty
p3 Objectinfo o3 W-Repl: WS 2,3,4
p4 empty
p6 empty
p7 Object o7 prot: 2; size: 16
access pointer
logic assoc. object/fault address

Figure 4: Virtual Address Space Organization

declared in the declaration part of functions (stack objects), indirection is not automatically provided. Hence, providing DSM functionality for stack objects is a non-trivial task. With the Shadow Stacks technique we introduce support for sharing of objects placed on stack, that in turn circumvents the necessity to employ expensive heap management techniques for sharable data. It therefore provides better performance by exploiting simple and fast stack management in a distributed environment.

4.1 Concept

The basic idea behind Shadow Stacks is realizing the required indirection by moving shared objects from their original stack to a different memory region, called *shadow region* and providing a pointer on the stack for accesses to these objects. The redistribution of objects among different memory regions delivers the ability to exploit the page fault mechanism of the hardware. Thus, the DSM management only has to handle accesses to objects not locally available. In case of an object being present in local memory, no additional management functionality is necessary apart from dereferencing the pointer to the object. This mechanism avoids performance penalties as known from all in software implementations.

For each thread, three different VA regions are managed by the DSM: the *thread-stack*, a *read-only shadow* and a *read/write shadow* (see fig. 4). An additional range of virtual addresses, called *faulting region* is kept commonly for all threads. The first two regions are called *Shadow Stacks*, because they are organized stack-alike to hold copies of the objects, that would normally be found on the stack of the thread. All of these regions with the exception of the faulting region are mapped and contain the data of the objects together with additional management information, such as the size of the object and the coherence protocol to use. Any object is logically associated to exactly one of the four regions at a time, depending on several conditions:

- If the object is only used by its own thread, it can be placed on the stack of the thread to be accessible as usual without any indirection (e.g. the objects o2 and o5 in fig. 4). Otherwise a pointer variable is generated on the stack, that holds the pointer referencing the shared object in one of the other regions.
- If the object is currently represented on a remote workstation, then the object pointer on the stack points into the faulting region (e.g. the objects o1 and o3 in fig. 4). An access to this object triggers a page fault that is handled according to the strategy defined for the object (e.g. migration or replication).
- If the object is locally available, it can either be handled without restriction, i.e. read and write accesses are allowed (e.g. the objects o4 and o6 in fig. 4), or accesses are restricted to read only (e.g. the object o7 in fig. 4). Such a restriction is useful for example in case of replicates.

A programmer using the Shadow Stacks DSM does not have to worry about these details. If he uses a Shadow Stacks aware compiler, even the question whether an object is sharable, i.e. visible to more than one thread or not can be predetermined by the compiler[a] (see section 4.2). To the programmer,

[a] Depending on the programming language used.

variable handling is as easy as in sequential programming languages, which was one of our design goals (see section 1). The required level of consistency, such as strong, weak or sequential consistency[10] is also defined by the programmer, either explicitly by hints or directives to the compiler and the runtime system or implicitly by the utilization of special language concepts if provided as e.g. in INSEL[11]. A coherence protocol – adequate to enforce the consistency requirement – is automatically and dynamically chosen by the resource management system.

At a first glance, it might seem as if this approach simply transfers the problem of stack management into a heap management problem, but in fact the Shadow Stacks grow and shrink according to their corresponding thread stack. Notice, that no expensive heap management is needed, such as garbage collection or free-list handling.

4.2 Implementation Details

In the subsequent sections we discuss substantial implementation issues of the Shadow Stacks concept. The hardware configuration chosen currently consists of 14 SUN UltraSparc 1 workstations, interconnected with a 100MBit/s Fast Ethernet and running SUN Solaris 2.5.1.

Task of the Compiler One of the design goals was to make programming as easy as possible. Therefore the indirection needed for Shadow Stacks is not provided by the programmer, but automatically generated by a modified version of the GNU compiler *gcc*. The compiler analyzes potential accessibilities for each object. All objects, accessible by only one thread are stored on the (original) stack of the thread or in registers as usual. In all other cases the object is placed on one of the Shadow Stacks and a pointer refering to the object is placed on the thread stack. The selection between the read-write or read-only Shadow Stack depends on the accessibility of the object and the initial coherence protocol to use as decided by the compiler. Depending on potential access patterns - as far as they can be predicted - the object might be subdivided into smaller management units.
In addition to analyzes and generation of indirections, the compiler also inlines code that increments and decrements the stack pointers of the Shadow Stacks.

Reducing Memory Consumption Due to the three stacks and the faulting region, a straight-forward implementation of the Shadow Stacks concept would lead to a four time higher memory consumption as usual. This overhead is reduced by several sanctions.

No physical memory is ever allocated for the faulting region. Solely a range of virtual addresses is reserved as faulting region for all threads in common. The id of the thread that caused the page-fault by accessing the faulting region is retrieved as explained below. Furtheron, for each sharable object only a reference to the current location in one of the Shadow Regions has to remain on the original thread stack. This reduces the memory consumption for sharable objects on the thread stack to the size of a pointer. Additionally, sharable objects are always placed on at-most one Shadow Stack, either read-only or read-write. The corresponding empty slot on the other Shadow Stack is used to store further management information, such as the location of replicates. This information would otherwise have to be kept separately consuming additional memory.

Identifying the Requested Object After a Page-Fault Memory faults are only detectable per memory page by the hardware. This normally prevents efficient handling of objects smaller than the page size. The identification of sharable objects via indirections allows individual handling of objects of any size. If a sharable object is locally mapped it can be accessed without any additional checks by dereferencing the pointer, pointing into one of the Shadow Stacks. If the object is not mapped in local memory, the value of the pointer is modified to point into the unmapped logical faulting region. Hence, dereferencing the pointer leads to a hardware page-fault. The page-fault handler of Solaris delivers information that allows to compute the address of the object that was to be accessed by extracting the load or store operation, that triggered the page-fault and the number of the register holding the address of the memory cell to dereference. The content of this register together with the value of an offset register, that is as well extracted from the operation delivers the identifier of the thread causing the fault as well as the unique stack address of the object to access.

Retrieving an Unmapped Object Based on the unique stack address of the object, the *home thread*, that is the thread, that created the object, can easily be located by the m-agent. According to the coherence protocol determined for the object and the properties of the request (e.g. read or write), the m-agent of the home thread decides about the actions to take. Information needed for this decisions such as the current owner and the state of the object is stored in the Shadow Regions as extensions to the object data. As a consequence, the current owner of the object is notified to deliver the object to the requesting thread. After receiving the object, its data is placed in one of the Shadow Stacks and the indirection pointer is changed to this location. Finally, the content of the reference register is modified to point to the correct address and the faulting thread is reactivated.

Protocol Identifiers For each shadowed object, a protocol identifier (`p-id`) and the size of the object are hold as additional management information. The `p-id` indicates the consistency requirement and the coherence protocol to use for the object. With the `p-id` the coherence protocol can be changed dynamically, if runtime monitoring indicates the necessity due to for example a high ratio of invalidations. Obviously, a state allowing the switch to a different coherence protocol is the prerequisite condition to dynamically change the `p-id`. Such a situation can be forced by invalidating all replicates.

Network Traffic As a consequence of eliminating false sharing, the Shadow Stack concept delivers small units that have to be transported over the network. Hence, an efficient implementation has to reduce communication overhead due to a large number of small messages[12]. A straight forward implementation would cause at least two messages being sent for each access to a locally unmapped object, regardless of its size. This problem is solved with a piggypacking strategy. Small objects and protocol messages, such as invalidations or request messages are collected and sent in joint.

5 Related Work

Most software realizations of DSM are using conventional virtual memory management hardware and local area networks. Li's Ivy system[3] was the first implementation of a page-based DSM. Modern implementations are Mirage+[13], TreadMarks[14] or Odin[15]. All of these implementations have two things in common: the size of management blocks of the DSM are equal to hardware page sizes and one unique coherence protocol is used for all blocks. The consequences are false sharing and inefficient protocols for a large number of objects. The implementation of Shadow Stack omits false sharing by replacing the concept of management blocks of a fixed size with handling of objects of any granularity and supporting individual coherence protocols for each object.

Other implementations also try to avoid these problems. COMMOS[16] supports flexible coherence protocols but its block size is still a hardware page. The basic idea behind COMMOS is to create a separate coherence server for each supported protocol. An invalid access to a page results in a page fault, which is handled by the associated coherence server. The Shadow Stack implementation also supports different coherence protocols, but is not bound to hardware pages.

The implementation of Midway^{17} is an example for a DSM that is not bound to hardware pages. All store operations are done by the Midway library. The coherence protocol runs without triggering a hardware page-fault. The coherence protocol is chooseable by the programmer, but is common to all objects of the application. In contrast to this, Shadow Stack uses the page-fault mechanism to improve performance. In Midway the library has to be entered each time a write operation is necessary, in Shadow Stack writes can be done without additional management, if the object is locally mapped.

The implementation of Munin^{18} tries to solve both problems. The block size depends on the size of the objects and for each object a different coherence protocol can be chosen. In contrast to Shadow Stack, the objects are clustered on hardware pages and the management block are multiples of hardware pages. Therefore false-sharing is not prevented. In Munin each shared object has to be explicitly marked sharable by the programmer, in Shadow Stack this is unnecessary.

6 Conclusion

The desired simplicity of programming distributed applications concerning memory management is attained with Shadow Stacks by omitting the introduction of additional concepts to the application level. Although objects of any size are individually managed by the DSM, efficiency is reached by exploiting the page-based faulting mechanism provided by the hardware instead of choosing an all in software implementation.

Implementation of *Shadow Stacks* on the SUN UltraSparc Architecture is not yet finished. Most of all, the integration of multiple stack management into the compiler has to be done. Our first prototype supports sequential as well as weak consistency for individual memory objects and will be able to dynamically switch between read-only, migratory, invalidation and update replication protocol. After extensive performance testing, our next step will be to integrate Shadow Stacks functionality for objects placed on heap. Furthermore, we are going to integrate several more consistency and coherence protocols. In a second project we are investigating distributed load balancing techniques. The impacts of load management on memory management and vice versa will influence further development of the Shadow Stack.

References

1. A. Geist et al. PVM 3 user's guide and reference manual. Tech. rep., Engineering Physics and Mathematics Division, Oak Ridge Laboratory, Oak Ridge, Tennessee, Sep. 1994.

2. R. M. Butler and E. L. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. Tech. rep., Mathematics and Computer Science division, Argonne National Laboratory, Argonne, Illinnois, 1992.
3. K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Dissertation, Dept. of CS, Yale University, New Haven, CT, Oct. 1986.
4. H. Lu. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. Master's thesis, Dept. of CS, Rice University, May 1995.
5. W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. *Proc. of Fourth SEDMS*, Sep. 1993.
6. M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of HICSS'97*, pages 130–139, Maui, Hawai, Jan. 1997. IEEE CS Press.
7. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.
8. R. M. Stallman. *Using and Porting GNU CC.* FSF, Nov. 1995.
9. J. Liedtke. Page Table Structures for Fine-Grain Virtual Memory. Tech. Rep. 872, German National Research Center for Computer Science (GMD), Oct. 1994.
10. A. S. Tanenbaum. *Distributed Operating Systems.* Prentice–Hall, 1995.
11. H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In *Proc. of PDPTA'95*, Nov. 1995.
12. C. Amza et al. Trade-offs Between False Sharing and Aggregation in Software Distributed Shared Memory. submitted for publication, 1997.
13. 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):887–909, Oct. 1994.
14. C. Amza et al. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb. 1996.
15. A. N. Pears. Odin: Implications and Performance of a Novel DSM Design. In *Proc. of ICSE'96*, Jan. 1996.
16. F. Huang. Operating system support for flexible coherence in distributed object systems. In L.-F. Cabrera and M. Theimer, editors, *Proc. of Fourth IWOOOS*, pages 171–174, Sweden, Aug. 1995. IEEE CS Press.
17. B. N. et al. The Midway Distributed Shared Memory System. In *Proc. of the IEEE CompCon Conference*, 1993.
18. J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, Sep. 1995.