A Framework for Portable Shared Memory Programming

Martin Schulz and Sally A. McKee
School of Electrical and Computer Engineering
Cornell University
Ithaca, NY, 14853
{schulz,sam}@csl.cornell.edu

Abstract

Widespread adaptation of shared memory programming for High Performance Computing has been inhibited by a lack of standardization and the resulting portability problems between platforms and APIs. In this paper we present the HAMSTER framework, which helps overcome these problems via cross-platform support and easy retargetability to a wide range of programming models. HAMSTER currently supports models ranging from thread APIs to one-sided put/get interfaces, all on top of a single, core middleware architecture. The HAMSTER framework allows programmers to use any of these models — without modification — on top of SMPs, NUMA-like clusters, and Beowulf systems. In addition, our experiments show that HAMSTER achieves this flexibility and portability without sacrificing performance.

1 Motivation

Shared memory provides parallel application programmers with a clean, natural programming model close to that for sequential programs. By abstracting away all details of low-level interprocessor communication, shared memory eases the programmer's burden, at least for initial implementations. Despite these apparent advantages, shared memory programming still plays a minor role in High Performance Computing. Performance and religious issues aside, shared memory still lags behind message passing with respect to standardization. There exists a diversity of shared memory programming models and APIs, many of which are targeted towards different architectures, user groups, and application domains. In addition, many APIs are only available on top of specific research systems unsuitable for production use. All these factors greatly reduce portability and compatibility of the existing code base. This is in contrast to the message paradigm, which achieves high portability by having largely converged to one standard, the Message Passing Interface (MPI) [32]. Very little standardization exists for shared memory (OpenMP [22] constitutes the most notable effort), and existing approaches have always targeted a specific class of machine, mainly SMPs.

This work presents HAMSTER, a comprehensive framework that helps overcome these problems. It runs on top of a set of very different architectures, ranging from tightly coupled shared memory multiprocessors to Beowulf-style clusters, and can be easily retargeted to any shared memory API with little effort. HAMSTER effectively decouples the programming model from the base architecture, enabling users to leverage a range of programming models on any of the supported architectures. The framework currently includes programming model modules for thread APIs, various DSM APIs, and one-sided communication libraries. Experimental results show that this list can be easily extended, and hence our experience validates the claim that retargetability can be achieved with minimal complexity. In addition, the overhead induced by this framework is negligible, and in some cases, the performance of the base system actually increases.

Section 2 introduces the HAMSTER framework, which the first author designed to overcome these problems. Section 3 and Section 4 present a detailed discussion of HAMSTER's two main innovations: a) support for multiple architectures, and b) retargetability to multiple programming models. Section 5 presents data on the complexity of porting the framework to new APIs, along with experimental results. Section 6 provides an overview of ongoing research on both the framework and the novel hardware on which it was developed.

2 Software Framework

Overcoming the current limitations of shared memory programming requires a general and adaptable framework that bridges the gap between different target architectures and the vast range of existing programming models. We present one such framework: HAMSTER. Originally, HAMSTER provided DSM support for SCI-based [10, 8] clusters [17]. Here we extend HAMSTER to other parallel

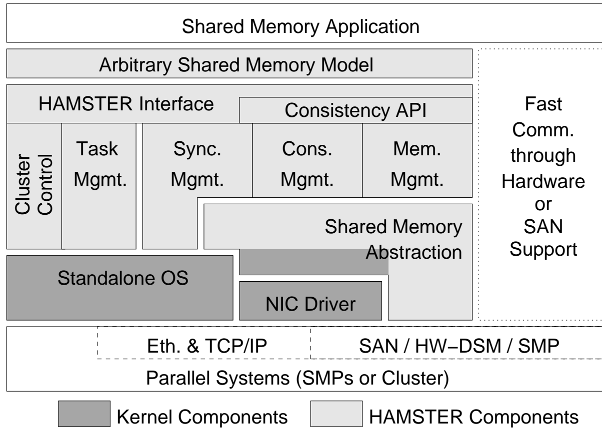

Figure 1. The HAMSTER Framework

architectures, as we implement a wider range of programming models on top of it. These additions make HAMSTER a generalized shared memory programming framework capable of unifying shared memory programming on parallel architectures.

An overview of the HAMSTER framework is shown in Figure 1. Shown at the top of the figure, applications execute transparently within their native programming model and are generally unaware of the underlying HAMSTER system. This transparency is created by a thin programming model layer implemented via services encapsulated within the HAMSTER API. The approach ensures lean programming model implementations, making it feasible to include a large range of models concurrently on top of a single core. Currently, models from thread APIs to remote put/get APIs have been successfully implemented within this framework.

Several management modules provide the versatile services enabling these programming models, with each module responsible for a subset of services. Clean design and usability requires that each of these modules be implemented orthogonally, without cross-module side effects. The modules rely on native operating system support and require a global shared memory abstraction. The implementation of the global memory abstraction depends strongly on the targeted architecture, which can range from straightforward SMPs to Beowulf-type clusters connected with Ethernet.

The majority of the framework is implemented at user level, easing development and deployment. For functionality that requires privileged commands or access to protected resources, HAMSTER relies on the respective native OS and its associated drivers. The only exception is a kernel-level component to establish the global memory abstraction in NUMA clusters. This kernel driver module controls the local application process's access to physical memory on all nodes, and hence the module must implement protection mechanisms.

In summary, HAMSTER's design achieves two main goals: a) it supports very different target platforms while hiding their specifics from the applications, and b) it retargets easily to a range of programming models that are then transparently exported to the user. Both contributions are further described in the following sections.

3 Support for Multiple Platforms

The success of HAMSTER, or any similar framework, depends on the ability to target many machine architectures. HAMSTER is therefore designed to run on parallel architectures ranging from Symmetric MultiProcessors (SMPs) with Uniform Memory Access (UMA) to Distributed Memory cluster architectures with NO Remote Memory Access (NORMA) capabilities.

3.1 Requirements

The main requirement of any base architecture for the HAMSTER system support for a global memory abstraction. This means it must be possible to allocate memory globally, and that any processor be able to transparently issue reads and writes to global memory regions. In addition, the architecture must include adequate synchronization mechanisms (e.g., in the form of locks, conditions, or barriers), and must provide sufficient information about the memory consistency model and its control mechanisms.

These functionalities need to be exported to the upper layers by appropriate software layers (either the OS or a special parallel run-time system). Due to the very different natures of the targeted architectures, HAMSTER does not enforce a common, standardized interface (which would be difficult to achieve), but rather it integrates the various native APIs into its core. This allows both high performance and easy adaptability, while keeping the impact on portability reasonable. Note that we assume ports between architectures will be much more rare than software ports providing new APIs on top of a given hardware architecture.

3.2 Mapping HAMSTER onto Existing Systems

HAMSTER currently supports hardware shared memory systems from SMPs, to more scalable NUMA (Non Uniform Memory Architectures) systems, and Beowulf-type clusters connected with both System Area Networks (SANs) and Ethernet.

Tightly Coupled Implementations. Hardware shared memory systems naturally fulfill the HAMSTER porting requirements. On these systems, the required functionality for memory allocation and synchronization is provided by the OS, mostly in the form of process control and/or native

thread APIs. In addition, those systems come with hardware coherence, and hence do not require explicit consistency control.

Loosely Coupled Implementations. At the other end of the spectrum lie Beowulf-type clusters with straightforward Ethernet interconnections. These systems contain no hardware support for shared memory, and hence they require extra software components to artificially create a global memory abstraction that is transparent for the user. Such systems, known as Software Distributed Shared Memory, or SW-DSM [21], have been the focus of research for almost twenty years, beginning with Li's Ivy [16]. Many different systems have been developed since, among them TreadMarks [1], Shasta [24], the Coherent Virtual Machines [14], and HLRC [23]. Most of the work in this area focuses on consistency, giving rise to many relaxed models, including Release Consistency [13], Scope Consistency [11], and Entry Consistency [4].

A SW-DSM system is required to enable shared memory programming on top of loosely coupled architectures within the HAMSTER framework. To avoid duplicating the work of others (and creating yet another DSM system), we integrate an existing, using it transparently to implement HAMSTER services on Beowulf architectures. We choose *JiaJia* [9], developed at the Chinese Academy of Science, because it boasts the advantages of open source, ready availability, and reasonably widespread use[1]. To our knowledge, JiaJia is the only existing implementation of Scope Consistency that is freely available, making it well suited for the fine-grain consistency mechanisms of HAMSTER services. JiaJia performance has been validated and shown to be competitive with DSM systems like the CVM [14].

Hybrid Systems. Many intermediate architectures lie in between the design extremes of tightly coupled multiprocessors and independent, networked nodes. These range from tightly coupled NUMA systems, like the SGI Origin series, to clusters with System Area Networks, like Dolphin's SCI [10, 8], Myrinet [5], or Infiniband. Inherent OS support for the former allows such machines to be managed much like SMPs. The latter can be used directly to speed communication within traditional software DSM frameworks, as in HLRC [23], PM2 [20, 2], or NOA [18]. Such clusters may also provide specialized hardware mechanisms for improved consistency protocols, as in Shrimp [3] or Cashmere [30]. The structure of these systems resembles that of software DSM systems like JiaJia, and hence we expect their integration into HAMSTER to be similar.

Cluster SANs represent an interesting design point in this spectrum: their remote memory read and write capabilities map any communication directly onto the hardware (without software protocol overhead), effectively yielding a NUMA system. Memory management still needs to be distributed, and must be handled in software to retain the possibility of executing on the more scalable cluster architectures. The resulting hybrid DSM system leverages both software and hardware shared memory techniques. SCI represents the most dominant SAN interconnect with the these properties, and two approaches have been implemented on top of it: SciOS [15] and the SCI-VM [25]. This work is based on the latter. Due to its software-like DSM memory management, the integration of the SCI-VM can be handled similarly to JiaJia's. The main difference lies in the fact that models like the SCI-VM recognize and handle remote data by extending the local memory management scheme in the OS, and thus they require an additional kernel component. The SCI-VM module within HAMSTER implements tight integration with the NIC driver, along with its proper initialization and configuration.

3.3 Integration Issues

The HAMSTER system integrates these different base architectures into a single framework. Special attention must be paid to cleanly integrating these independent and potentially conflicting systems. The critical issues for this are a unified startup operation and the integration of the different communication frameworks.

The task model and system initialization differ significantly among the three base architectures: shared memory multiprocessors rely on the OS to perform the required tasks; JiaJia contains internal mechanisms for remote job starts; and the SCI-VM uses external, script-based remote job execution. The latter two are integrated by removing JiaJia's internal startup mechanisms and replacing them with the configuration mechanisms of the SCI-VM, including the unification of the different node configuration files. The multiprocessor platform is integrated two ways: a) multithreaded programming models directly leverage the native operating system APIs, or b) models oriented towards process parallelism treat these systems as separate nodes. The latter is accomplished by using the startup and memory management mechanisms of the SCI-VM library, but appying them to UMA instead of NUMA systems. The appropriate startup handling can then be chosen according to the requirements of the target programming model.

Within the communication framework, conflicts can occur if two base systems use the same interconnection resources without coordinating. In the multiprocessor subsystem, the OS handles conflicts, but the problem is evident when integrating software and hybrid DSM. Both systems use socket-like communication for their internal messaging[2], and hence they compete for access to the network and to signaling mechanisms. HAMSTER cleanly solves

[1] JiaJia has been downloaded to over 120 sites [33].

[2] In the SCI-VM, only configuration data is initially passed using socket communication; all application data is transferred through hardware DSM.

this integration issue is by coalescing the two separate interconnection structures into one, which then forms the basis for both models and is also exposed to the user for external messaging.

4 Programming Models

Portable programming models can be implemented with ease on top of HAMSTER's integrated, cross-architectural platform. HAMSTER facilitates this portability by providing a set of services on top of the architectural abstraction discussed above. These services provide the necessary mechanisms to support shared-memory programming, hiding architectural differences to bridge between programming models.

4.1 Design Criteria

The main design goal of these services is to provide the necessary functionality to implement arbitrary programming models. Their design and implementation is guided by the following four criteria.

Comprehensiveness. The services must be sufficiently comprehensive to meet prerequisites for any shared memory programming model. Mechanisms must be generic, avoiding bias towards any particular programming model, while retaining enough power and expressiveness to recreate any shared memory functionality.

Flexibility. Individual services must be flexible and highly parameterizable. Each service can then be custom-tailored to a specific use, allowing a direct mapping of the many calls in target APIs to these parameterized versions of the underlying services.

Ease of Use. The services must be easy to apply and must behave predictably, without side effects. Their design must minimize the implementation complexity of a target programming model.

Low-Overhead Implementation. HAMSTER services must guarantee high efficiency. Overheads are sufficiently low that performance may improve in some cases, due to tight integration between the API and the core system (for instance, the integration of the two messaging layers can yield improved message performance).

4.2 Supported Services

HAMSTER services can be classified into five types, with each set of services implemented as a separate, orthogonal module, without cross-modules dependencies. This implementation philosophy keeps the framework's learning curve low.

Memory Management Module. Services dealing with memory allocation and distribution reside in the Memory Management module. Users may specify coherence constraints and distribution annotations for any memory subsystem (as long as the subsystem can accommodate the given parameters). A capability test routine lets the user probe the underlying shared memory system to discover supported coherence schemes.

Consistency Management Module. Relaxed coherence provides opportunities for improved application performance, but requires appropriate control mechanisms. In conjunction with the Synchronization Management module's constructs, the services in this module support creation of appropriate relaxed consistency models.

Synchronization Management Module. Synchronization primitives coordinate individual processes or threads in any shared memory model. This module contains lock and barrier implementations optimized for the appropriate base architecture, and it provides mechanisms to implement programming-model specific constructs. Whenever possible, these are either implemented on top of native synchronization mechanisms offered by the base operating system and its thread API or by using user-level communication mechanisms in the underlying hardware.

Task Management Module. This module contains services required to implement the correct task model. Note that this module does not include specific thread routines for creation or joins, but rather provides mechanisms for the integration of such thread services into the programming model. This design maintains the HAMSTER's generality and allows programming model implementations to benefit from platform-dependent, native thread services, and to maintain platform-specific semantics (by not introducing a new thread API with different semantics).

Cluster Control Module. This module implements the functionality for managing cluster configuration and provides services for identifying nodes and querying node parameters. It includes a simple messaging layer used for initialization. This module differs in composition from the others in that it provides services used by other modules in addition to those used to implement programming models.

4.3 Performance Monitoring

Performance monitoring is crucial to tuning shared memory applications and systems for efficient operation. Shared memory communication is implicit, out of the direct control of the programmer. Some shared memory APIs or models therefore include mechanisms to generate performance statistics useful for application optimization, as in the performance monitoring system on the SGI Origin series [6] and the performance statistics in TreadMarks [1] and JiaJia [9]. These routines are very specific to the pro-

gramming model, are often badly documented, and in some cases have been added in an ad hoc manner.

As a programming-model independent framework, HAMSTER is designed with generalized monitoring mechanisms within its individual modules. From the user's perspective, each module provides independent services to query and reset its statistics counters. The implementation of these services maintains these statistics independently of what the underlying architecture provides[3].

These monitoring routines are useful in several scenarios: the application may directly access these services (and thereby circumvent the transparency of the programming model implementation); a run-time system may use them for dynamic optimization; or an independent monitoring system may attach externally. The monitoring capabilities therefore enable architecture-independent and programming model-independent tool support, making it possible to leverage toolsets across platforms.

4.4 Programming Model Implementation

The services described above constitute the HAMSTER interface that provides the basis for the implementation of shared memory programming models. Additional services independent of the parallel programming environment (e.g., platform-independent support for application timing measurements) augment the usability of the framework. Implementing a concrete shared memory API on top of HAMSTER first requires an analysis of the individual API calls, many of which can be directly mapped onto a corresponding HAMSTER service. The calls not having a direct counterpart in the HAMSTER interface must be decomposed and then implemented using a set of HAMSTER services.

When implementing a new API, the following three components require special attention, for they may vary largely between shared memory models: memory consistency model, task structure, and initialization. The first can be achieved by a proper combination of consistency-enforcing routines and synchronization constructs. If the task structure requires more than what is provided by HAMSTER's inherent SPMD model, it must be based on the native architecture's thread API. Initialization operations are split into internal initialization of the share-memory model's support mechanisms and initialization of the external cluster configuration (and accompanying startup procedures). HAMSTER provides a set of standard templates for both, and these can be reused when implementing new models.

We have already implemented several fully functional programming models on top of HAMSTER. A high-level analysis of these models is included in the next section, and more details about the implementation of DSM APIs [27] and threading APIs [26] can be found elsewhere.

[3]The amount of information provided may depend on the base architecture capabilities.

4.5 Notes on Consistency

Efficient, flexible support for consistency is the most critical requirement for a framework like HAMSTER. Base architectures and target programming models can differ radically in how they enable or enforce consistency, and proper mapping between software and hardware is crucial to providing full functionality without sacrificing performance.

A weaker software model may always be mapped onto a stronger hardware model, as consistency models simply define a lower bound on the expected memory coherence. A good example is an implementation of relaxed consistency (e.g., Release Consistency [13]) on top of Shared Memory Multiprocessors with hardware cache coherence. The SMP system provides a stronger model in hardware (usually Processor Consistency [7]), but can still execute applications written for the weaker model. In fact, most commonly used thread libraries for SMPs, including POSIX threads [31] and Win32 threads [19], are themselves based on weak consistency models, despite the available hardware support for stronger consistency models.

Distributed memory architectures whose interconnects are LANs or SANs, however, *require* weak consistency models for efficient execution. In almost all cases, the hardware's relaxed consistency scheme matches the programming model's or API's. The exact semantics of the target architecture's underlying model must be matched during programming model implementation. For this purpose, the HAMSTER framework includes a separate consistency API containing optimized implementations of all widely used models, including Release Consistency [13] and Scope Consistency [11]. Other models can be implemented based on the HAMSTER services alone, but this might yield degraded performance.

Ultimately, researchers and practitioners seek a generic API for high-performance consistency control (which would be part of HAMSTER's consistency module). A first step in this direction exists for NCC-NUMA systems [29]; the results are encouraging, but not yet easily transferable to other DSM types. Transferability requires a general, formal model for DSM consistency, along with significant changes to and integration efforts for the target DSM systems.

5 Evaluation

We evaluate HAMSTER according to two main criteria: a) a quantitative assessment of the ease with which HAMSTER is retargeted, and b) a discussion of the HAMSTER overheads compared to those for native execution. The raw

Benchmark	Working Set
Matrix Multiplication	1024x1024 matrix
Computation of π	-
Successive Over Relaxation (SOR)	1024x1024 matrix
LU Decomposition	1024x1024 matrix
WATER (Molecular Simulation)	288 / 343 molecules

Table 1. Benchmarks and Their Working Sets

Programming Model	#Lines	#API calls	Lines/call
SPMD model	502	23	21.8
SMP/SPMD model	581	25	23.2
ANL macros	146	20	7.3
TreadMarks API	326	13	25.1
HLRC API[4]	137	25	5.5
JiaJia API (subset)[4]	43	7	6.1
POSIX threads	725	51	14.2
WIN32 threads	988	42	23.5
Cray put/get (shmem) API	505	29	17.4

Table 2. Implementation Complexity of Programming Models Using HAMSTER

performance of the underlying DSM systems is not a property of the HAMSTER system itself, and so we focus on retargetability and overhead. Wall-clock execution times are given elsewhere [28, 9], but we also include some of these results here to provide perspective on the performance of the particular DSM examples discussed herein.

5.1 Experimental Setup

For all following experiments, we use a four-node Linux cluster equipped with both SCI and a switched Fast Ethernet. Each dual-SMP node contains Intel 450 MHz Xeon processors with 512 MB main memories. For all DSM experiments, we use only one CPU per node; for execution on top of hardware DSM, we use one node alone.

The benchmarks are taken from the collection of codes included in the JiaJia distribution. They are all well known benchmarks that have been adapted and optimized to the JiaJia API. Table 1 lists all benchmarks and their respective working set sizes.

5.2 Implementation Effort

A framework like HAMSTER must a) support a range of programming models, b) require only modest effort to port each model, and c) extend easily to support new programming models. Together with broad support for a variety of target architectures, this flexibility is essential to the success of the framework.

Table 2 lists the models that we have already ported to HAMSTER. They span the spectrum of shared memory APIs, ranging from distributed thread APIs (for both Linux [31] and Windows [19]) to one-sided communication APIs with remote put/get capabilities, as in Cray's *shmem* API [12]. In addition, HAMSTER includes APIs from several SW-DSM systems, as well as a custom SPMD-style library. SPMD implements a more user-friendly abstraction for most HAMSTER services: its ease of use provides a good basis for run-time systems supporting high-level, shared memory programming.

[4] These results are based on a subset of the SPMD programming model.

Table 2 indicates the programming effort required for each model, giving total lines of code used to implement the model with respect to the size of the API. To ensure fair comparison, each count is computed by a simple script that first removes comments and empty lines, and then (to a certain degree) standardizes the coding style. These results show that each programming model can be implemented with limited complexity: on average, we invest fewer than 25 lines of code for each API call.

The HAMSTER DSM implementations represent one end of the programming-model spectrum. Note that the SPMD models require relatively many lines of code to export the underlying (harder-to-use) HAMSTER services to the user, yielding API calls with broader functionality at the cost of increased implementation complexity. This model is the first implemented within the project [25], forming the basis for similar programming models, including JiaJia [9] or HLRC [23]. Like the SPMD model, these DSM APIs use synchronous allocation routines involving all nodes. In contrast, TreadMarks [1] uses single-node allocation, and hence requires an additional, special routine to distribute allocation data. The TreadMarks API thus lends itself to a low-complexity implementation, since almost all routines can be mapped directly to HAMSTER services (attesting to the completeness of the HAMSTER design). Only the one routine required to distribute data in the TreadMarks single-node allocation scheme must be implemented fully by hand: single-node allocation requires that the data retrieved from this additional allocation routine be delivered to the other nodes (the other APIs implement global allocation with an implicit barrier, paying overhead costs for a consistency model that is not always required).

At the other end of the spectrum lie the more complex POSIX and Win32 thread APIs. Both contain a forwarding mechanism that enables the application to execute threading routines either on the node on which the target thread runs, or (as in the case of the thread-creation routine) on the node on which the respective routine should execute. Again, for flexibility's sake the HAMSTER services intentionally omit providing such a forwarding framework. Instead, forwarding can easily be built on top of the HAMSTER messaging primitives: all communication uses some form of active message present within the HAMSTER modules. The manycalls supported by these APIs means that the forwarding facility's implementation will be frequently used, offsetting the higher complexity of implementing the threading models (especially for the Win32 thread API [19]).

Our experience indicates that simpler programming models (like most SW-DSM APIs) can usually be imple-

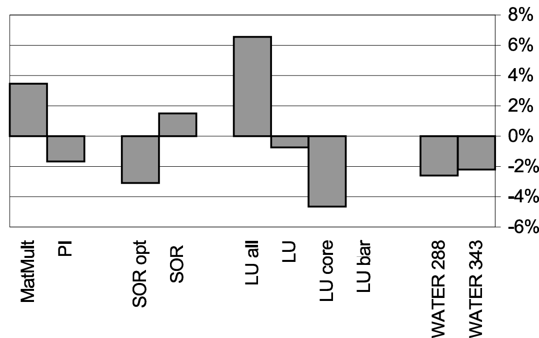

Figure 2. Overhead of Execution with HAMSTER Compared to Native Execution on JiaJia (4 Nodes)

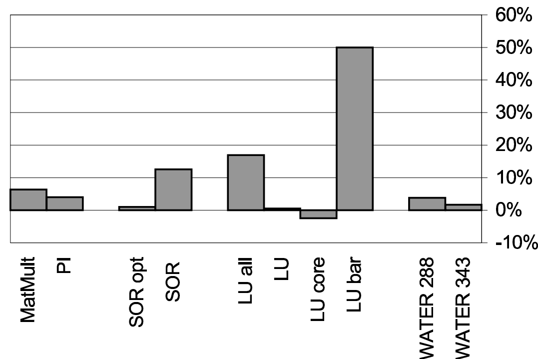

Figure 3. Performance of Hybrid-DSM with SW-DSM as Baseline (4 Nodes)

mented and tested within half a day. More complex models (like the thread APIs) require more code, due to their complex command-forwarding mechanisms. However, even those APIs can usually be implemented within a day. Implementing a wide range of programming models on top of HAMSTER is not only feasible but straightforward: the system meets its retargetability design goal.

5.3 Overhead

To evaluate the overhead induced by HAMSTER, we run the above mentioned benchmarks on top of the same system architecture, using both the standard distribution of JiaJia without modifications, and a JiaJia API implemented on top of HAMSTER. The latter is executed on top of the integrated and modified JiaJia within HAMSTER.

The results of these experiments are shown in Figure 2. Positive bars indicate a performance degradation when run on top of HAMSTER, whereas negative bars indicate a performance gain. For SOR, the results of both a transparent run and an optimized run with locality optimizations are shown. All other codes are used with locality optimizations. In addition, LU results have been split into an overall time, a time without initialization, the actual computational core without synchronization, and the time spent in barriers.

In summary, this data shows a very small influence on overall performance behavior: in single-digit percentages. In many cases, we even observe slight performance increases. This is most likely due to HAMSTER's tight integration, especially of the communication subsystem.

5.4 Raw Performance

Using the same setup, we execute the benchmarks on different target platforms supported by HAMSTER. For this, only the configuration of HAMSTER (in the form of a configuration file) is changed between experiments; the actual codes are not modified, and in fact we use the identical binaries.

Figure 3 shows the performance ratio between running on the Hybrid-DSM vs. the Software-DSM system on all four nodes. Positive bars indicate an advantage for the Hybrid-DSM, whereas negative bars indicate faster performance on the Software-DSM system. As expected, the Hybrid-DSM outperforms its Software counterpart, as applications can take advantage of the special hardware features. The difference, however, is not as high as expected, due to small data set sizes used in these experiments and the fact that these codes (as part of the JiaJia suite) have been tweaked for optimal performance under Software-DSM. Note, for instance, that the optimized SOR code exhibits only a small performance difference, while the unoptimized version is significantly better on the Hybrid-DSM system. This indicates that the Software-DSM relies more heavily on locality optimizations for good performance. As for the LU code, the overall performance is significantly better, as the typical write-only initialization is very expensive in Software-DSM systems, while the actual core computation is faster. However, the Hybrid-DSM system balances the overhead more evenly across the complete application, as can be seen in the significantly lower barrier times.

Figure 4 shows the results of the same experiments on only two nodes in relation to the performance of the same codes run with the same SMP system as the target architecture. As expected, the tight coupling of the SMP outperforms the other two DSM systems in most cases. The only exception is the matrix multiplication, which is faster on the DSM systems. This is due to the fact that this code is memory bound, and hence can profit from the use of separate memory buses on the two nodes. As for a comparison between Hybrid-DSM and Software-DSM, no clear trend can be deduced from the results given here, most likely due to the small cluster size.

This paper represents but a preliminary study, and our

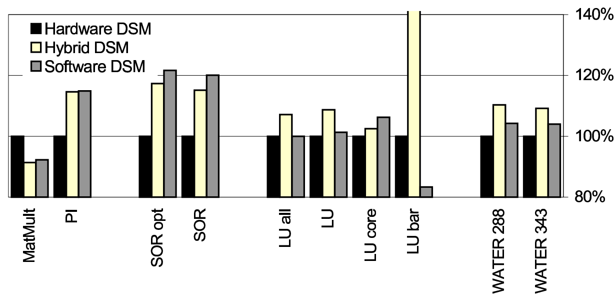

Figure 4. Performance of Hardware–, Hybrid–, and Software–DSM (2 Nodes)

goal is to show the capabilities of HAMSTER and to demonstrate (and quantify, to the extent possible) the flexibility of the overall framework. Experiments with more and larger codes, as well as with different and larger system setups, are all part of ongoing work.

6 Future Research

HAMSTER provides the means to efficiently support all major relaxed consistency models. Its implementation, however, is currently oriented towards the model of the underlying system architecture, and is aided by a specific consistency API. While this approach provides the necessary functionality, it would be preferable to include a fully generic and user-centric consistency API that includes a more formal mechanism for reasoning about memory consistency. Such an API is already included in the SCI-VM, the original base system from which HAMSTER [25] grew, but it needs to be adapted to the more complex consistency structures of modern software DSM protocols. This will allow memory consistency implementations to be more easily verified, and will enable experiments with new, potentially application-specific consistency models.

HAMSTER's ability to concurrently support multiple DSM systems within one framework offers the opportunity for a direct and fair comparison among such systems. To our knowledge, such a comparison between hybrid DSMs and multiple software DSMs has not yet been performed, largely due to the difficulties of comparing different software systems on top of very different hardware architectures. We expect that such a study would not demonstrate the dominance of a single approach, but rather would reveal that individual system performances are dependent upon application characteristics (particularly memory and I/O access patterns). If this turns out to be the case, HAMSTER makes it possible to combine several different DSM mechanisms within the execution of a single application, resulting in custom-tailored, shared memory solutions for individual applications.

7 Conclusions

Here we have described the HAMSTER framework to support the shared memory paradigm. In principle, HAMSTER allows programmers to use any shared memory API without modification, running on top of SMPs, NUMA-like clusters, and Beowulf systems. The framework currently supports shared memory models ranging from thread APIs to one-sided put/get interfaces, all on top of a single, core middleware architecture. Our experience demonstrates that HAMSTER achieves this flexibility and portability without sacrificing performance. Compared to native execution on a four-node JiaJia system, HAMSTER suffers less than 6.5% overhead, and in other cases HAMSTER performance overheads range from about a 2% slowdown to almost a 4.5% speedup.

Acknowledgments

Much of this work was performed at the Technische Universität München under the support of the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR).

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1995.

[2] G. Antoniu, L. Bouge, and R. Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. In *Parallel and Distributed Processing, Proceedings of IPDPS workshops including RTSPP*, volume 1586 of *LNCS*, pages 496–510. Springer Verlag, Berlin, Apr. 1999.

[3] A. Belias, L. Iftode, and J. Singh. Shared Virtual Memory across SMP Nodes Using Automatic Update: Protocols and Performance. In *Proceedings of the 6th workshop on Scalable Shared-Memory Multiprocessors*, Oct. 1996. Also as Princeton Technical Report TR-517-96.

[4] B. Bershad and M. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Sept. 1991.

[5] N. Boden, D. Cohen, R. Felderman, J. S. A. Kulawik, C. Seitz, and W.-K. Su. Myrinet: A Gigabit–per–Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[6] D. Cortesi and J. Fier. Origin2000 and Onyx2 Performance Tuning and Optimization Guide. Technical Report 007-3430-002, Silicon Graphics, Inc., 1998. Available at http://techpubs.sgi.com/.

[7] J. Goodman, M. Vernon, and P. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–73, 1989.

[8] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *LNCS State-of-the-Art Survey*. Springer Verlag, Oct. 1999. ISBN 3-540-66696-6.

[9] W. Hu, W. Shi, and Z. Tang. JiaJia: An SVM System based on a New Cache Coherence Protocol. In *Proceedings of High Performance Computing and Networking (HPCN-Europe)*, volume 1593 of *LNCS*, pages 463–472, Apr. 1999.

[10] IEEE Computer Society. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.

[11] L. Iftode, J. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computer Systems*, 31:451–473, 1998.

[12] C. Inc. *SHMEM, in CRAY T3E C and C++ Optimization Guide*, chapter 3. SG–2178 3.0.1, Available at http://www.cray.com/.

[13] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, Jan., 1995.

[14] P. Keleher. *CVM: The Coherent Virtual Machine*. University of Maryland, cvm version 1.0 edition, Nov. 1996. http://www.cd.umd.edu/projects/cvm/.

[15] P. Koch, J. Hansen, E. Cecchet, and X. R. de Pina. SciOS: An SCI-based Software Distributed Shared Memory. In *Proceedings of the First International Workshop on Software Distributed Shared Memory (WSDSM)*, June 1999. Available at http://www.cs.umd.edu/˜keleher/wsdsm99/.

[16] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Sept. 1986. Available as TR492.

[17] M. Schulz. Efficient deployment of shared memory models on clusters of PCs using the SMiLEing HAMSTER approach. In A. Goscinski, H. Ip, W. Jia, and W. Zhou, editors, *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 2–14. World Scientific Publishing, Dec. 2000.

[18] D. Mentre and T. Priol. NOA: A Shared Virtual Memory over a SCI cluster. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*, pages 43–50. Cheshire Henbury, Sept. 1998. ISBN: 1-901864-02-02.

[19] Microsoft Cooperation. *Microsoft Platform Software Development Kit*, chapter About Processes and Threads. Microsoft, 1997. available with Microsoft's SDK.

[20] R. Namyst and J.-F. Mehaut. PM2: Parallel Multithreaded Machine, a Computing Environment for Distributed Architectures. In *Proceedings of ParCo 1995*, pages 279–285, Sept. 1995.

[21] B. Nitzberg and V. LO. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–59, Aug. 1991.

[22] OpenMP Architecture Review Board. *OpenMP C and C++ Application, Program Interface*, Oct. 1998. Version 1.0, Document Number 004-2229-01, Available from http://www.openmp.org/.

[23] M. Rangarajan, S. Divakaran, T. Nguyen, and L. Iftode. Multi-threaded Home-based LRC Distributed Shared Memory. In *Proceedings of the 8th Workshop on Scalable Shared Memory Multiprocessors (held in conjunction with ISCA)*, May 1999.

[24] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low Overhead, Software–Only Approach for Supporting Fine–Grain Shared Memory. Technical Report WRL Research Report 96/2, Digital Western Research Laboratory, Nov. 1996.

[25] M. Schulz. *True shared memory programming on SCI-based clusters*, chapter 17, pages 291–311. Volume 1734 of Hellwagner and Reinefeld [8], Oct. 1999. ISBN 3-540-66696-6.

[26] M. Schulz. Multithreaded Programming of PC clusters. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT) 2000, Philadelphia, PA, USA*, pages 271–278. IEEE, Oct. 2000.

[27] M. Schulz. Overcoming the Problems Associated with the Existence of Too many DSM APIs. In H. Bal, K. Löhr, and A. Reinefeld, editors, *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid — DSM2002: Distributed Shared Memory on Clusters*, pages 319–324. IEEE, May 2002.

[28] M. Schulz and W. Karl. Hybrid-DSM: An Efficient Alternative to Pure Software DSM Systems on NUMA Architectures. In L. Iftode and P. Keleher, editors, *Proceedings of the Second International Workshop on Software Distributed Shared Memory (WSDSM)*, May 2000. Available at http://www.cs.rutgers.edu/˜wsdsm00/.

[29] M. Schulz, J. Tao, and W. Karl. Improving the Scalability of Shared Memory Systems through Relaxed Consistency. In R. Bianchini and L. Iftode, editors, *WC3'02, Second Workshop on Caching, Coherency, and Consistency*, June 2002.

[30] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

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

[32] WWW:. MPI – The Message Passing Interface Standard . http://www.mcs.anl.gov/mpi/index.html, Dec. 1999.

[33] WWW:. Thanks for your interest in JIAJIA Software Distributed Shared Memory System . http://www.cs.wayne.edu/ weisong/jiajia.html, Oct. 2002.