Local Relaxed Consistency Schemes on Shared-Memory Clusters

Martin Schulz*, Jie Tao†, and Sally A. McKee*

*School Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853, USA
Email: {schulz,sam}@csl.cornell.edu

†Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München, Germany
Email: tao@in.tum.edu

***Abstract*— Shared Memory is an attractive and convenient programming abstraction, and Shared Memory Clusters are a straighforward and efficient way to provide it. Unfortunately, the overhead of enforcing consistency to implement shared memory on such architectures can be prohibitive. Ensuring caches remain coherent throughout an application's entire execution, as done on CC-NUMA architectures, is particularly expensive in terms of hardware complexity, and scales poorly with system size. For this reason, most High Performance Compute Clusters are implemented as No-Remote Memory Access (NORMA) architectures supporting message-passing APIs, thereby avoiding the need for global memory coherence schemes. Rendering Shared Memory Clusters more competitive thus requires reducing consistency overheads.**

In this paper, we use the Splash-2 benchmarks to quantify the potential for reducing these overheads. We evaluate the costs of several different *local consistency schemes*, which require neither global update nor invalidation operations and hence exhibit greater scalability. Each is compared against a conventional CC–NUMA and an optimal, false–sharing free model. On average, we find that invalidating all cache lines and flushing write buffers to enforce release consistency performs slightly better than the conventional CC–NUMA scheme, but is burdened by the overhead caused by extensive cache invalidations. Two novel local schemes based on selective invalidations, which are introduced in this work, target this problem and are capable of achieving a significant speedup within 10% of the optimal coherence behavior for most applications.

I. Motivation

Clusters built from commodity components and interconnected with System Area Networks (SANs) are becoming increasingly popular, largely due to their attractive price/performance ratio. Advances in System Area Network technology have introduced native support for shared memory abstractions. These *Shared Memory Clusters* [1] combine the cost efficiency and scalability of clusters with the programmability and fast communication support of shared memory multiprocessors. They thereby form a competitive alternative to tightly coupled large-scale SMPs or CC-NUMA machines.

These cluster architectures face the challenge of maintaining memory consistency in a loosely coupled SAN environment. The I/O bus-based architectures of such SANs prevent access to the coherence traffic from the SAN NICs. To overcome this problem, Heinrich et al. [1] maintain consistency via special hardware coherence controllers on top of an Infiniband-based scalable SAN fabric. Their scheme supports the same programming abstraction as the successful CC-NUMA architectures (e.g., the Origin 2000 [2]), but at a much lower system cost. Nonetheless, their coherence controller requires complex hardware additions to current cluster nodes, and the design assumes access to coherence information, though such access is not supported in current node architectures. Developing alternative coherence mechanisms would allow us to exploit current, commodity architectures to build high-performance Shared Memory Clusters.

One alternative approach relaxes the consistency of the system's shared memory abstraction and restricts coherence operations to specific application synchronization points. At these points—and only these points—the system guarantees that it will maintain a predictable memory state in the presence of consistency enforcing operations like cache invalidations or buffer flushes. Since coherence operations execute locally on a single node, implementing global coherence does not limit the scalability of the underlying architecture.

We have previously implemented a prototype relaxed consistency system within the SMiLE (Shared Memory in a LAN-like Environment) project [3]. This SCI-VM [4], [5] system was built with Dolphin's Scalable Coherent Interface [6], [7] on top of a commodity x86 cluster. One difficulty in implementing this approach is the inability of current processors to selectively invalidate cache blocks. This means that the entire cache has to be invalidated at every synchronization point, which leads to additional overhead caused by subsequent cache line loads.

To reduce these invalidation-induced coherence overheads, we propose two cache controller extensions. These schemes have to ability to support the efficient implementation of local coherence schemes resulting in less coherence overhead. Note that we advocate adding only small hardware additions of isolated complexity on the individual nodes; no global, system-wide component will be added. These extensions allow cache controllers to transparently and independently track relevant portions of the cache and selectively invalidate only the necessary blocks. Experiments with five numerical benchmarks taken from the Splash-2 suite [8] show that local schemes based on these additions can outperform the conventional CC-NUMA schemes and can reduce execution time overhead to within 3% of an optimal consistency scheme in some cases.

The remainder of this paper is organized as follows. Section II provides the basis for the discussion of local coherence protocols for Shared Memory Clusters, and Section III discusses the three schemes studied in this work. Section IV provides simulation results contrasting the three schemes to a CC-NUMA and an oracle implementation, Section V discusses some related work, and Section VI concludes the paper with some final remarks.

II. Relaxing Consistency

Consistency models provide the shared-memory programmer with a safe abstraction with which to reason about the behavior of the underlying memory system. Models differ from one another with respect to the rules that govern when updates become visible by other processors. Existing SMP and CC-NUMA machines [2], [9], as well as proposed Shared Memory Clusters [1], provide strong consistency in which write operations trigger global updates and thereby become immediately visible to all nodes. These approaches require many global operations and complex, system-wide hardware mechanisms. Alternatively, the consistency model can be relaxed by distinguishing between regular and consistency-enforcing memory operations. While the former are executed without global coordination, the latter directly impact the visibility of all writes previously issued to the memory system.

A. Design Guidelines

Consistency operations are critical to system performance, and hence need to be designed carefully such that they adhere to the following guidelines:

Request new data only when required. The system should only perform invalidations when new data are required by the application. Current global coherence mechanisms perform updates or invalidations on every write, and hence communicate information that may never be accessed by remote nodes.

Avoid global consistency operations. Scalability is a traditional problem of large shared memory machines. One source of this problem is the frequency of global coherence operations. By keeping these operations local to a single node, larger systems become possible.

Limit impact on system design. No new scheme should require drastic modifications to current processors. While SMP–based systems can use the standard MESI–like mechanisms in the processors, CC–NUMA systems require a significant amount of special hardware in the form of either hardware maintained directories [10], [2] or sharing lists [6], [11].

Low hardware complexity. To be feasible, new hardware designs should not significantly deviate from current architectures. For example, the substantial custom hardware required to realize CC–NUMA machines is a barrier to more widespread adoption.

B. Relaxed Consistency Models

In order to take effect, applications need to explicitly make use of consistency enforcing operations. In order to minimize the obvious impact on the programming model, it is useful to leverage the concept of relaxed consistency models [12], which combine synchronization constructs with consistency enforcing mechanisms. They build on the observation that all shared memory codes require synchronization and that these events coincide with consistency requirements. Therefore, relaxed consistency models can often be introduced without any or with only minor code changes.

Relaxed consistency models have been extensively researched in the realm of Software Distributed Shared Memory (SW–DSM) [13], [14], [15] and implemented in the form of relaxed consistency protocols [16], [17]. While latter ones are not applicable or necessary in HW–DSM scenarios, the concept of relaxed consistency models can be applied in a rather straightforward way. The result for the programmer is a secure, global memory abstraction.

III. Local Consistency Schemes

Based on these design guidelines, we have developed a family of three local consistency schemes: Full Invalidation, Cache Footprint Invalidation, and Dual Scope Invalidation. While the first one is capable of running on and has been implemented for today's commodity hardware, the other two require minimal changes to the individual local cache controller on each node. All three leverage strictly hardware-based implementations and hence require neither special software handlers nor specific network protocols, in contrast to previous work in SW-DSM systems.

A. Full Invalidation

The first approach investigates mechanisms currently available in commodity systems, namely *Full Cache Invalidation*, to control and enforce consistent cache state. In a typical Shared Memory Cluster, inconsistent state can appear in any component along the memory path, including various read and write buffers, as well as all processor caches. To control these components, two independent operations can be defined: a) a *write flush* of all complete operations in the local write pipeline including all write buffers, and b) a *read invalidation*, which cleans all local read buffers and caches.

Both operations are generally available in some form on any current system. Cache invalidations are mostly implemented as full cache invalidations only and are present as special instructions in the Instruction Set Architecture (ISA). A typical example of this occurs on the Intel x86 architecture [18]. Write flushes, which are required not only for the schemes discussed here, but for any multiprocessor environment, are normally present less explicitly in CPUs. They are often implicitly triggered by specific fence operations. Shared memory networks also have to be equipped with such barriers, though often more explicitly. In summary, these properties make it feasible to implement such an architecture on commodity components or clusters and thereby to profit not only from the improved consistency mechanisms and greater scalability, but also from the excellent price/performance ratio of such systems.

It should be noted that in most current CPUs, cache invalidations are total operations in the sense that the entire cache is invalidated. This is true for both *Acquire* and *Barrier* operations. Both therefore incur additional local cache misses and performance degradation due to the invalidation of data that must be reloaded before future reuse.

As discussed above, these consistency enforcing operations can be combined with synchronization constructs to form consistency models. An example of such a programming interface is provided by merging the above hardware mechanisms with lock, unlock, and barrier invocations as follows:

- *Acquire*
  - Perform lock operation
  - Perform read invalidation
- *Release*
  - Perform write flush
  - Perform unlock operation
- *Barrier*
  - Perform write flush
  - Perform barrier operation
  - Perform read invalidation

This kind of coherence scheme adheres to the guidelines above and provides a system with Release Consistency [16], [19]. Informally, it ensures that data structures are kept consistent as long as they are correctly protected by locks. This is generally the case, since virtually all shared memory programming models, whether they are based on hardware or software mechanisms, contain some notion of relaxing the memory system. This is even true for the POSIX memory model governing the use of POSIX threads on most commodity operating systems [20].

This scheme has the distinct advantage that it relies solely on commodity components, and can hence be implemented on today's hardware. We have done this within the SCI-VM [4], [5] project on commodity clusters interconnected with SCI [7], [6]. It has yielded good performance on many benchmarks. In the following, however, we will not further address this prototype implementation, but instead use pure simulation to facilitate the comparison against other schemes.

B. Cache Footprint Invalidation

To reduce cache miss overhead, it is necessary to avoid full cache invalidations and instead use a more selective scheme. Most current processors, including the Intel x86 architecture [18], offer no or only very limited support for partial cache invalidations. Even if processors did offer such a feature, an implementation with explicitly managed partial invalidations would lead to high management overhead in upper layers, similar to that of Entry Consistency models [21]. We therefore propose simple additions to the cache controller to enable implicit tracking of required updates. This, in turn, allows *Cache Footprint Invalidation*– invalidating only the cache footprint contributed by a particular application's consistency-constrained remote read accesses, rather than the entire cache.

	Read	Write
Local	Cached, under control of local MESI	Write back into cache
Remote	Selective invalidation	Write through into main memory

TABLE I
CLASSIFICATION OF MEMORY ACCESSES AND RESPECTIVE ACTIONS.

For this purpose, we distinguish read and write operations as well as their destination nodes (Table I). This selective invalidation protocol focuses on remote memory read accesses, since these are responsible for potential inconsistencies. The other three classes of accesses can be performed in the same way as in standard systems since they do not directly introduce inconsistencies.

During any cache hit on an attempted read of remote data, the protocol must determine whether the data are potentially stale and must be invalidated. For this purpose, we introduce an additional one-bit flag for each cache line indicating whether it is valid. In brief, this flag is cleared whenever the application requires up–to–date data and is set when the corresponding cache line is freshly loaded.

More specifically, during each remote read access resulting in a cache hit, this flag is read and if not set, the cache line is immediately invalidated and the read access is treated as a cache miss. This leads to the request of an up–to–date cache line from the next level of the memory hierarchy.

These flags are set each time the corresponding cache line is loaded with the most current value, such as after a cache line fill following a miss and after a write operation. The reset of all flags, on the other hand, is carried out each time the application requires that new data has to be visible at a node. Like above, this is connected to synchronization constructs such as barrier and lock operations. At these points, all flags in all caches on the local processor are reset to zero and hence trigger a reload on subsequent accesses.

To ensure correctness, we also require write flushes at unlock operations. Otherwise, new data may be kept in intermediate buffers, while read operations (potentially after an invalidation triggered by a zero flag) return the old data still present in the physical memory cell.

In summary, this scheme ensures that any remote memory access delivers a value at least as recent as the preceding barrier and lock operation. Its consistency model is therefore equivalent to the full invalidation scheme described above and only differs in the timing of the invalidation (at the access instead of at the barrier/lock). Further, this scheme retains the programming ease and code portability of the full invalidation approach.

C. Dual Scope Invalidation

The Cache Footprint Invalidation scheme fails to exploit the semantic differences implied by barriers and locks. Whereas barriers are used to control access to larger memory regions and are intended for a loose cooperation between them, locks usually protect specific data structures during short time intervals. *Dual Scope Invalidation* leverages the fact that these two constructs usually protect different types of resources by separating memory updates triggered by barriers from those triggered for critical regions protected by locks.

This approach partitions cache block invalidations with respect to the associated synchronization constructs controlling them. Under Dual Scope Invalidation, memory operations that would need to be invalidated under a lock, might not need an update when accessed outside a critical region, but rather between two barrier operations. Depending on a code's synchronization pattern, this can lead to fewer invalidations and improved performance.

For the programmer, the result is a slightly more relaxed consistency model with two consistency scopes [17]. Due to the inherent separation between locks and barriers in most shared memory codes, however, it can be expected that the impact on the programming model will be negligible for most codes.

This new scheme can be implemented by adding a second consistency flag to each cache line that mirrors the behavior of the bit under Cache Footprint Invalidation. Whereas the same flag under Cache Footprint Invalidation is set and cleared by both lock and barrier constructs, its use is disambiguated under Dual Scope Invalidation– one flag governs memory accesses protected by locks, the second controls accesses within critical regions.

Such separation allows the coherency controller to enforce different consistency policies based on the corresponding synchronization mechanism; Dual Scope Invalidation enforces lock protected consistency only within critical regions, whereas access outside these regions may happen in an inconsistent manner. Similar to above, cache line loads set the flags and barrier and lock operations again reset them. While a cache line load always sets both flags (since the cache line is valid for both scopes following the load), each synchronization operation only resets one set of flags and thereby achieves the separation of consistency information between locks and barriers.

The complete protocol for a remote read operation is shown in Figure 1. During a remote memory read, the coherency controller knows whether the current process or thread is executing within a critical section. The consistency flag associated with locks is evaluated and used to guard memory accesses only within a critical section. As within the Cache Footprint Scheme, the consistency flag associated with barriers must always be set to enable a cache hit. If this flag is reset or

```
if not flag_barrier(cache_line) then
    invalidate line
else
    if not flag_lock(cache_line) and in_mutex then
        invalidate line
    fi
fi
```

Fig. 1. Dual–Scope: Consistency protocol during a remote read.

the lock flag is reset while the application holds a lock, the cache invalidates the line and simulates a cache miss.

One of the prerequisites for this scheme is the existence of a special lock flag within the processor. This flag is set by the OS when entering a critical region and is zeroed when performing the corresponding unlock. This flag can then be used to determine the appropriate action in case of a read hit. In order to minimize the impact of this flag and to avoid additional system calls, this lock flag should be part of the user accessible register set. This will allow each user process to set or reset this flag directly using user-level synchronization constructs [22], and will only incur the cost of a single register access.

The resulting consistency model is slightly more relaxed than its two predecessors. This stems from the fact that it separates two different groups of consistency enforcing operations, whereas all other schemes so far have been based on a single, global mechanism. These two areas correspond to consistency scopes [17]. They were originally implemented in Scope Consistency, a straightforward extension of Release Consistency utilizing the implicit relationship between barriers on one side and lock variables on the other. The Dual–Scope scheme discussed here is therefore similar to Scope Consistency, but restricts the number of scopes to two.

IV. Experimental Evaluation

In this section we present performance numbers comparing all three schemes in terms of number of invalidations and number of executed cycles. To contrast these numbers to existing systems, we have included the performance of a processor consistent CC-NUMA approach and of an oracle which determines whether accesses to given addresses are shared, i.e., will be reused by another processor in the near future, or will not lead to true communication. Only the former accesses will lead to remote invalidations, while others are ignored, resulting in the minimal necessary network traffic.

A. Simulation Setup

All simulations in this paper have been done using the SIMT framework [23], a detailed simulator for shared memory multiprocessors, which itself is based on Augmint [24], [25]. This system simulates architectures with an arbitrary number of processors, each with its own multilevel cache hierarchy, and allows the implementation and evaluation of several cache consistency schemes.

The machine we model for this work is constructed of single-processor building blocks based on Intel Pentium II^{TM}systems. It employs simple write buffers, separate instruction and data L1 caches of 32 KBytes, a unified L2 cache of 512 KBytes, and a local MESI protocol between the CPU and the network interface. Remote memory accesses are modeled with 2000 cycles, while local accesses are assumed to consume 100 cycles. These values have been measured in the prototype system based on the full invalidation scheme and hence provide a realistic evaluation scenario.

Due to the hardware implementation of all three schemes, modeling software components is not required. The only software mechanisms used by the three approaches are the manipulation of consistency flags. These are generally assumed to be one cycle, since they usually just require a flag to be set, while the ensuing cache invalidation is executed outside of the critical path. In addition, we model an OS that distributes all memory pages transparently in a round-robin fashion. During the SCI-VM experiments this has proven to be useful as the general distribution.

B. Benchmarks

All benchmarks were taken from the SPLASH-2 parallel benchmark suite [8] as they provide a large range of different memory access patterns. To reduce simulation time, we reduce the input dataset for Barnes from 32K to 4K bodies. We increase the work for FFT from 2^{10} to 2^{18} complex numbers. For all other benchmarks we use the standard inputs as given in the original SPLASH-2 distribution.

C. Results

All codes have been simulated with a configuration of 8 and 32 nodes using the three consistency schemes introduced above. Figure 2 shows these results in terms of executed cycles in relation to the optimal oracle scheme. In all cases, the new local coherence schemes outperform the conventional CC-NUMA scheme, as they are able to reduce the number required remote memory accesses due to less invalidations and less coherence traffic. Note, however, that this is done at the expense of a more relaxed consistency model. It hence forms a tradeoff between ease of use for the programming and performance. This is, however, often not a problem, since most programming models do already have a relaxed consistency model as the base system (even if the host architecture would support more).

Comparing the three local schemes, the full invalidation is outperformed by the two other schemes and the Dual Scope Invalidation further improves performance in all codes, which include both barrier and lock constructs. In codes that just deploy barriers for synchronization, however, Dual Scope and Cache Footprint Invalidation behave equally, since the second scope remains unused.

Figure 3 shows the number of cache line invalidations incurred during the execution of the various schemes. This is an indicator of the amount of traffic caused by coherence operations in the system. The graph shows that the two partial schemes, Cache Footprint and Dual Scope Invalidation, can significantly reduce the number of invalidations compared to the Full Invalidation, but that even those schemes still perform a large number of wasted invalidations compared to the oracle scheme. This leaves room for further optimizations.

V. Related Work

Software controlled cache coherence in HW-DSM architectures with NUMA characteristics has already been investigated in a few projects. Within Platinum [26] a coherent memory abstraction for a NUMA architecture is created using the virtual memory management system. Using this mechanism in coordination with global page state information, read or write accesses, which cause potential cache inconsistencies, can be detected and handled. Similar approaches are taken in [27] and [28]. All three, however, are limited by the granularity of the virtual memory system.

A different approach to deal with potential cache inconsistencies has been undertaken within the Shared Regions project [29], which defines a high–level abstraction to group and manipulate memory regions. Based on this abstraction, it provides mechanisms to invalidate or flush remote memory regions and uses these mechanisms in coordination with synchronization primitives to guarantee a safe and reliable memory abstraction [30]. This approach avoids additional overhead for maintenance and hence represents a lean and easy–to–implement solution. This scheme is very similar to Full Invalidation, but has not been put in the context of Relaxed Consistency Models since it predated the formal introduction of relaxed memory consistency models to SW-DSM systems.

Several research projects have investigated opportunities of reducing coherence related overhead without influencing the consistency model of the end-user application. Examples of such work are the Cashier system [31], which is capable of introducing coherence annotations using a compiler framework, and the use of speculative self–invalidations of unused data in DSM systems [32]. Both of these approaches, even though they initially target a different domain due to their premise of maintaining consistency transparently, might be able to complement our schemes. The use of an annotating compiler framework can alleviate some burden from the programmer, while speculative (non-)invalidations have the potential to further increase the performance. In future work we will look in these directions.

VI. Conclusions

Shared Memory Clusters offer a cost efficient alternative to current shared memory multiprocessors. However, maintaining memory coherence in such architectures is challenging. Current approaches use coherence mechanisms similar to those in modern CC-NUMA machines. This has the drawback of relying on global coherence mechanisms, which restrict scalability, and requiring access to intra–node coherence traffic, which is not possible in current node architectures.

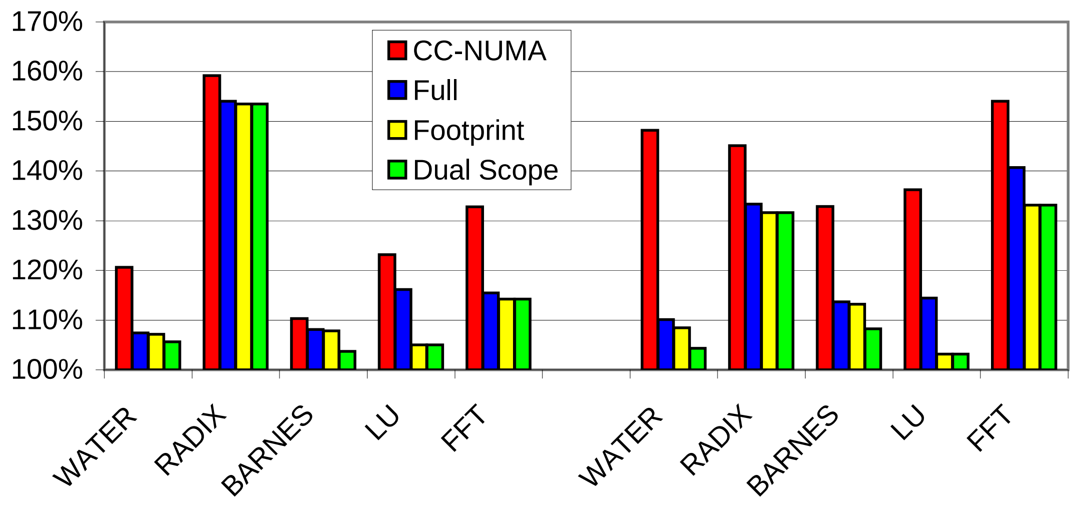

Fig. 2. Execution times (in cycles) of all benchmarks under different consistency schemes relative to optimal scheme (left set: 8 nodes, right set: 32 nodes).

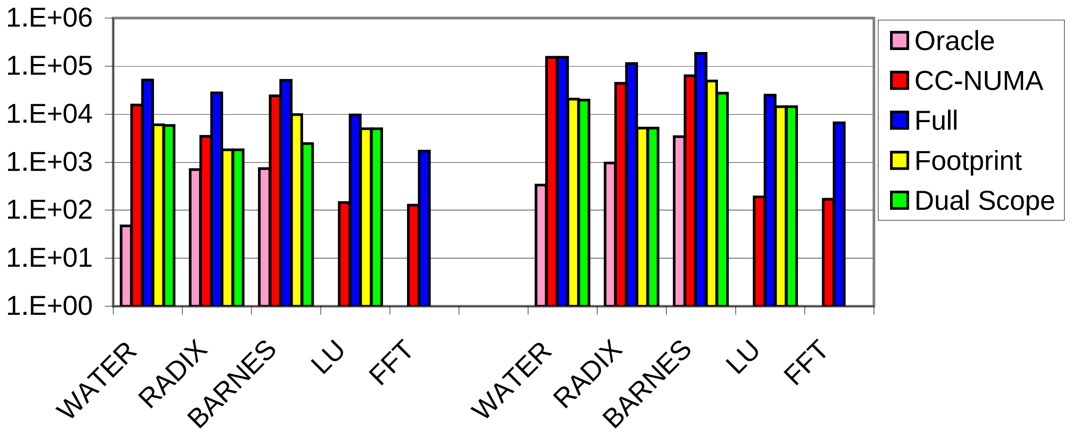

Fig. 3. Number of invalidations of all benchmarks under different consistency schemes (left set: 8 nodes, right set: 32 nodes).

As an alternative, we have proposed the use of local consistency schemes and relaxed consistency models. We have introduced three different schemes, one which can be implemented using current SAN technology, and two which are built on top of small, local extensions in the cache controller. None of the schemes rely on any global hardware component and hence each maintains the scalability of cluster architectures. Experiments with several numerical benchmarks have shown that these schemes can outperform conventional CC-NUMA machines and can reduce the overhead to within 3% of an optimal oracle scheme in some cases.

Acknowledgment

We gratefully thank Brian White for providing us with a great deal of feedback and editorial assistance with this paper, and with helping in the discussion of the results and their validation.

References

[1] M. Heinrich, E. Speight, and M. Chaudhuri, "Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters," in *Proceedings of the Fourth International Symposium on High-Performance Computing, Lecture Notes in Computer Science*, May 2002, pp. 78–92, vol. 2327, Springer-Verlag.

[2] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *In Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, June 1997, pp. 241–251.

[3] M. Schulz, J. Tao, C. Trinitis, and W. Karl, "SMiLE: An Integrated, Multi-paradigm Software Infrastructure for SCI-based Clusters," in *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, Berlin, Germany, May 2002, pp. 247–254.

[4] M. Schulz, *True shared memory programming on SCI-based clusters*, ser. LNCS State-of-the-Art Survey. Springer Verlag, Oct. 1999, vol. 1734, ch. 17, pp. 291–311, iSBN 3-540-66696-6.

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

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

[7] H. Hellwagner and A. Reinefeld, Eds., *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters*, ser. LNCS State-of-the-Art Survey. Springer Verlag, Oct. 1999, vol. 1734,

iSBN 3-540-66696-6.

[8] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH–2 Programs: Characterization and Methodological Considerations,” in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 24–36.

[9] WWW: Sun Servers (Hardware, SUN Enterprise (TM) servers), http://www.sun.com/servers, Feb. 2001.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Stanford FLASH Multiprocessor,” in *In the Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, Apr. 1994, pp. 302–313.

[11] G. Astfalk, T. Breweh, and G. Palmeh, “Cache coherency in the convex mpp,” *Convex Computer Corporation*, Feb. 1994.

[12] S. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” Department of Electrical and Computer Engineering, Rice University and Western Research Laboratory, DEC,” Rice University ECE Technical Report 9512 and Western Research Laboratory Research Report 95/7, Sept. 1995.

[13] K. Li, “Shared Virtual Memory on Loosely Coupled Multiprocessors,” Ph.D. dissertation, Yale University, Sept. 1986, available as TR492.

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

[15] M. Eskicioglu, “A Comprehensive Bibliography of Distributed Shared Memory,” Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA, Tech. Rep. TR–95–01, Oct. 1995.

[16] P. Keleher, “Lazy Release Consistency for Distributed Shared Memory,” Ph.D. dissertation, Rice University, Jan., 1995.

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

[18] Intel Corporation, *Intel Architecture Software Developer’s Manual for the PentiumII*. published on Intel’s developer website, 1998, vol. 1–3.

[19] M. Schulz, J. Tao, and W. Karl, “Improving the Scalability of Shared Memory Systems Through Relaxed Consistency,” in *Proceedings of the Second Workshop on Caching, Coherence, and Consistency (WC3 ’02)*, June 2002.

[20] Technical Committee on Operating Systems and Application Environments of the IEEE, *Portable Operating Systems Interface (POSIX) — Part 1: System Application Interface (API)*, 1995th ed. IEEE, 1996, ch. Secton 2.3.8, General concepts/memory synchronization, ANSI/IEEE Std. 1003.1.

[21] B. Breshad and M. Zekauskas, “Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors,” School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Tech. Rep. CMU-CS-91-170, Sept. 1991.

[22] M. Schulz, “Efficient Coherency and Synchronization Management in SCI based DSM systems,” in *Proceedings of SCI-Europe 2000, The 3rd international conference on SCI–based technology and research*, G. Horn and W. Karl, Eds. SINTEF Electronics and Cybernetics, Aug. 2000, pp. 31–36, iSBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[23] J. Tao, W. Karl, and M. Schulz, “Using Simulation to Understand the Data Layout of Programs,” in *Proceedings of the IASTED International Conference on Applied Simulation and Modelling (ASM 2001)*, September 2001, pp. 349–354. [Online]. Available: http://wwwbode.in.tum.de/archiv/artikel/asm01/TSK2001.ps.gz

[24] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas, “The augmint multiprocessor simulation toolkit for intel x86 ar chitectures,” in *Proceedings of 1996 International Conference on Computer Desig n*, October 1996.

[25] Augmint Homepage, http://iacoma.cs.uiuc.edu/augmint/, 2002.

[26] A. Cox and R. Fowler, “The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINIUM,” in *In Proceedings of the 12th Symposium on Operating Systems Principles (SOSP)*, Dec. 1989, pp. 32–44.

[27] K. Petersen and K. Li, “Multiprocessor Cache Coherence Based on Virtual Memory Support,” *Journal of Parallel and Distributed Computing, Special Issue Scalable Shared Memory Multiprocessors*, vol. 29, Oct. 1995.

[28] L. Kontothanassis and M. Scott, “High Performance Software Coherence for Current and Future Architectures,” *Journal for Parallel and Distributed Computing*, 1995.

[29] H. Sandhu, B. Gamsa, and S. Zhou, “The Shared Regions Approach to Software Cache Coherence on Multiprocessors,” in *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993.

[30] H. Sandhu, “Integrating Applications with Cache and Memory Management on a Shared Memory Multiprocessor,” in *Proceedings of CASCON*, 1992.

[31] T. Chilimbi and J. Larus, “Cashier: A Tool for Automatically Inserting CICO Annotations,” in *Proceedings of the International Conference of Parallel Processing (ICPP)*, Aug. 1994.

[32] A. Lai and B. Falsafi, “Selective, Accurate, and Timely Self-Invalidation using Last Touch Prediction,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.