

Improving the Scalability of Shared Memory Systems through Relaxed Consistency

Martin Schulz, Jie Tao, and Wolfgang Karl

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)

Institut für Informatik der Technischen Universität München, Germany

{schulzm,tao,karlw}@in.tum.de.

Abstract

Shared Memory Systems with a globally consistent memory abstraction are currently very successful. The main reason for this can be seen in their ease-of-use and the convenient programming model which is close to the sequential one. Especially the memory coherency mechanisms contribute to this, as the programmer does not have to take care of any data conflicts or memory update operations. This convenience, however, comes with the price of a restricted scalability of such systems since consistency enforcing mechanisms are of global nature and therefore infer machine global data traffic. This is true for both UMA architectures with their snooping protocols and NUMA systems, which mostly deploy some kind of directory scheme. One way to solve this scalability problem, while maintaining the abstraction of a global memory, is to omit any hardware coherency mechanism and to replace it with appropriate software mechanisms. This can be done in a way that the impact on the programming model is minimized by relying on a formalized relaxed consistency model, a formalism well established in SW-DSM systems. The result are architectures with similar programmability properties compared to their coherent counterparts, but with significantly higher scalability properties.

1 Motivation

Currently, a trend towards shared memory systems with a globally consistent memory abstraction can be observed; in recent years several of such systems have been built and successfully marketed using both Uniform Memory Access (UMA) architectures, like the SUN Enterprise servers [33] or the HP Superdome systems [31], or Non Uniform Memory Access (NUMA) architectures, like the SGI Origin 2000/3000 [14]. In addition, several new systems of this class are planned or announced for the near future.

These systems maintain the coherency between the individual processors and caches in hardware using a cache coherency protocol which guarantees a consistent memory view at any time on any processor. While this is a very convenient abstraction for the user of such systems, it often provides a consistency which is too strong with respect to the actual

application requirements and hence infers unnecessary overhead. In addition, these cache coherency mechanisms are a major source of the scalability limitations of current shared memory multiprocessors. They all rely, independent of their internal organization, on the automatic invalidation of shared cache lines in the case of updates from a remote processor and hence involve some kind of global communication procedure on every shared write operation. The necessary hardware mechanisms also add complexity to the overall system design and mostly rely on complex proprietary or custom hardware components in order to support a competitive system size.

Shared memory systems based on HardWare Distributed Shared Memory (HW-DSM) mechanisms, but without coherency mechanisms, on the other hand, are simpler and more straightforward to build. They can be based on commodity processor/cache modules, which do not need to be aware of the fact that they are used in a multiprocessor system, and on simplified interconnection fabrics without the need to support special coherency mechanisms. This leads to a significant reduction in system development and production cost, and additionally to performance improvements due to reduced communication/update traffic. This forms the base for a higher system scalability.

The main disadvantage of this approach is the loss of the consistent view on the memory, which needs to be compensated in order to reach a stable and reliable environment for the user. This, however, can be easily accomplished by applying the formalism of relaxed consistency models, which are well known from the SW-DSM domain [16, 18, 5], and research in this area (e.g. [13, 10, 21]) has shown that relaxed consistency models only lead to a minimal impact on the programmability. Hence, the ease-of-use is maintained despite the missing hardware coherency mechanisms. Non-coherent systems, in coordination with an appropriate hardware coherency management, are therefore competitive alternatives to their coherent counterparts and have the potential to break the scalability barrier for this class of machines.

This paper investigates the architectural benefits gained from dropping hardware consistency mechanisms and describes the necessary software consistency techniques required to establish a reliable programming platform on top of them. This also includes a detailed description on how

to implement *Release* and *Scope Consistency* in such an environment. In addition, the paper presents a set of simulation results validating the claim for increased performance and scalability on such systems. They show that these are capable of outperforming their hardware coherent counterparts in most cases and especially in larger system configurations.

The remainder of this paper is structured as follows. Section 2 features a brief introduction into current consistency mechanisms for both UMA and NUMA machines. In Section 3 the relaxation of memory consistency and its impact on both the architecture and the programming of such systems is discussed, followed by a detailed description on how to implement two sample relaxed consistency models on such a platform in Section 4. In Section 5, the results of a study is shown comparing the performance of non-cache-coherent shared memory machines with their coherent counterparts. The paper is rounded up by some conclusions and a brief outlook on future work in Section 6.

2 Consistency in Shared Memory Systems

Current shared memory systems are mostly implemented in a hardware coherent fashion. This means that the coherency between the individual caches, which are distributed in the system, is maintained automatically and transparently by the system. Depending on the base architecture, two principle approaches for this can be distinguished: passive approaches based on snooping protocols and active approaches based on some kind of directory information about the contents of remote caches.

The best known representatives of the former group is the MESI or Illinois protocol [19]. It is typically applied in bus-based SMP systems as it depends on all processors or cache controllers listening to all memory traffic on the bus. Based on this snooped information the status of the local cache tags is accordingly adjusted using a simple state machine. By relying on this kind of broadcast property of a bus, however, it also inherits the scalability limitations of bus systems. In addition, it requires that all traffic is globally propagated within a bus cycle limiting the bus clock frequency and the physical system size.

The second group, the active coherency protocols, aims at avoiding the necessity of relying on the observation of coherency events on a common bus and hence enable non-bus-based shared memory multiprocessors. They maintain distributed information about shared replicated data on remote processors. Several different ways have been proposed and implemented to manage this information, including directories as it is done in the Stanford DASH [15] or the SGI Origin O2000/3000 [14], or hardware maintained linked lists as it is done in the Scalable Coherent Interface (SCI) [9]. However, also here all updates need to be propagated to processors holding replicas of that data, again leading to global communication. In addition, the maintenance of either directories or linked-lists requires complex custom hardware with direct access to the processor/memory bus. In most cases this prohibits the use of low-cost commodity components.

3 Relaxed Consistency and its Implications

As discussed above, the presence of an implementation of coherent shared memory in hardware limits the scalability of the underlying system. In addition, it increases the hardware complexity of such systems and prevents their implementation from commodity components. To avoid these problems, this work therefore investigates the use of shared memory architectures without hardware support for coherency and illustrates both the benefits and challenges encountered in this class of systems.

The modifications examined in this work are, however, solely directed to the coherency mechanisms of shared memory systems. The base architecture providing a global memory abstraction in hardware at either a physical or virtual address space is always maintained. This has to be seen in contrast to the well examined distributed memory architectures in combination with Software DSM (SW-DSM) systems, which establish a global memory abstraction in software. These are able to work on any parallel architecture as they can be based on top of any communication subsystem, but have to cope with an increased software complexity. By assuming a hardware DSM system, as it is done here, these overheads are avoided [25]. In addition, this kind of hardware support is easy and straightforward to achieve, does involve only minimal hardware complexity, and provides an efficient support for both message passing and shared memory [12]. Therefore, this kind of support, either in a coherent or non-coherent fashion, is on the rise with more and more UMA and NUMA being introduced by the major systems vendors and can therefore be assumed to be present in a large percentage of future parallel architectures.

3.1 Architectural Advantages

The most significant advantage of non-coherent architectures, compared to their coherent counterparts, is that they can rely on fully independent processor/cache modules. The need for any global component or infrastructure as well as any system wide dependency is removed. The only hardware requirement is an interconnection fabric capable of establishing a global memory abstraction, which can be reduced to the ability to perform remote memory accesses directly from the local processor. As this does not impose a significant hardware complexity, it leads to a straightforward system design with similarly good scalability properties as distributed memory machines, but with the advantage of a system global memory abstraction.

In addition, the requirements for the interconnection fabric within the multiprocessor will be significantly reduced, as global coherency with its complex multi- or broadcast structures can be avoided. In addition, the interconnection fabric no longer needs to be integrated into a coherency controller or connected directly to the system bus. Altogether, they enable the use of commodity System Area Network (SAN) technology known from clustering instead of having to rely on complex custom interconnection fabrics.

In summary, the use of non-coherent shared memory systems will lead to simpler and less specialized systems. It will also open the door to develop state-of-the-art NUMA architectures based on commodity clustering techniques leading to a significant reduction in both hardware and development cost and to a higher system scalability. In addition, it will be possible to leverage on higher bus clock speeds at the various levels of the memory subsystem, as global dependencies are removed.

3.2 Control Consistency in Software

The price for this reduced hardware complexity is a missing consistency control for the overall system. This will lead to inconsistent cache content and stale data being returned on read accesses to shared memory areas. A sample scenario for this situation is shown in Figure 1. While all caches are consistent at time t_1 , the store operation at time t_2 only appears in the memory hierarchy of the issuing processor leaving the old value in the second cache.

Since in the model discussed in this paper all communication is solely handled in hardware, these kind of consistencies can not be avoided. In order to still ensure a safe and reliable execution environment, adequate software mechanisms need to be deployed which are capable of handling the consequences by directly controlling the various buffers which can potentially contain stale data. These can be found in any component of the memory hierarchy, including processor write buffers, streaming buffers in the network card, and prefetch buffers for both the network and the processor, but the main source of potential problems is found in the actual processor caches since these have the largest capacities. The problems and issues found in all of these components, however, is the same. Hence, without a loss of generality, they can be combined into two blocks: (1) buffers in the read pipeline keeping data duplicates including all hardware caches and (2) buffers in the write pipeline used to optimize stores. In the following, the former will simply be referred to as caches, while the latter will be denoted as write buffers.

Each of these two components can be controlled by a separate operation. Caches can be invalidated deleting all data, including any stale copy; write buffers can be flushed and the data thereby propagated through the complete store pipeline. The former enables the deletion of stale data from the local cache, guaranteeing that the next access will produce a miss and then lead to a fetch of the most current data from main memory, while the latter represents a memory barrier which ensures that the current status of the local processor is guaranteed to have propagated to main memory and hence is available to other processors. Within this work, both operations are thereby assumed to work globally, i.e. all caches or write buffers on the local node are invalidated or flushed without any restrictions with respect to address ranges. This is done to realistically model current processor architectures, which mostly do not provide the possibility for partial cache invalidations; this includes Intel's x86 architecture [11], which is used as the base architecture in this study.

It should be noted that the invalidation operations themselves are of purely local effect, as they only affect the locally cached data. They are normally implemented by calling the cache flush instruction of the underlying architecture and hence do not include any communication. Write propagations, on the other hand, cause communication by flushing data to other processors. However, only the data contained in current write buffers is affected. As this data is of very limited size, also the write flush has only an insignificant global impact.

Programs intended for non-coherent architectures can use these two operations at those points in the program in which consistent data is required and thereby ensure a correct program execution. As a consequence, updates are only performed when necessary for the application, enabling an optimized use of the memory system customized to the specific coherency requirements of the respective application.

3.3 Relaxed Consistency Models

This need to control the consistency behavior in software and the possibility of inconsistent data between flushes (obviously) has a direct impact on the programming model. The user is (in principle) responsible to mark the shared data and to identify when a consistent view on the data is required — a difficult, error-prone, and tedious task which would seriously restrict the acceptance of non-coherent machines.

Fortunately, this task can be hidden from the user by applying the concept of relaxed consistency models. They formalize the use of inconsistent memory and hence give the user a safe abstraction. They are combined with synchronization operations, like locks and barriers, which are contained in any shared memory code. As accesses to shared data normally needs to be synchronized anyway in order to prevent races, it thereby introduces a natural way to enforce a consistent access to shared data. This ensures a level of programmability on such systems which is close to that of hardware coherent architectures.

This claim has been sufficiently shown to be true in the context of so called Software Distributed Shared Memory systems (SW-DSM) [16, 18, 5]. These systems are designed to provide a global memory abstraction in distributed memory systems without any hardware support for remote memory accesses. Here, it was necessary to purposely introduce memory inconsistency in order to be able to delay update propagation as long as possible until the data is actually requested in order to reduce the communication overhead. The consequences for applications, however, are the same as in the context of non-coherent shared memory machines, where the inconsistencies stem from uncoordinated caches; hence the same formalism can be used.

This similarity, however, is restricted to the actual consistency model with the effect that, given any particular consistency model, any system adhering to it provides the same environment in terms of memory behavior to the application. The implementation, however, varies significantly between SW-DSM implementations and the hardware based systems with software consistency control discussed here. While the

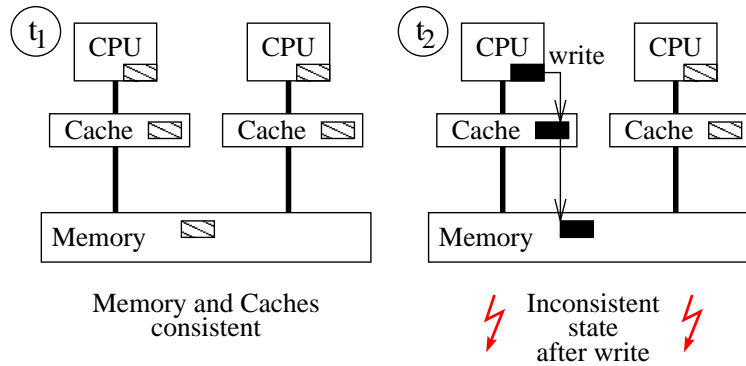


Figure 1. Potential cache inconsistency when caching remote memory

former rely on sophisticated and complex memory update protocols which are responsible for the actual data transfer between nodes, the approach discussed here relies on hardware to perform the actual data transfers and only provides means to insert the necessary flush and/or invalidation operations into the application codes. Unlike in traditional SW-DSM systems, no further protocols are required thereby significantly reducing the system software complexity.

This is also true compared to existing systems, which implement a global memory abstraction using a DSM system relying on modest hardware support for efficient update propagation. Examples for such systems are Shrimp [3] with its automatic update support enabling local writes to be propagated to a second remote node and Cashmere [27], which is built on top of a remote write network. In both cases, though, the underlying interconnection fabric misses support for reads from remote memory. Therefore, they still have to rely on additional software protocols to request and retrieve remote data, an operation unnecessary in the approach discussed in this paper due to the assumed HW-DSM capabilities.

4 Implementing Relaxed Consistency Models

The work on SW-DSM systems has lead to the definition and implementation of several different consistency models (e.g. [13, 10, 4]). Almost all of them, most prominently *Release Consistency* [13], can be implemented on non-coherent hardware by combining the use of synchronization operations in user applications, like locks and barriers, with cache invalidations and write buffer flushes as introduced above. In the following, this will be demonstrated using both *Release Consistency* and *Scope Consistency* [10], followed by a brief outlook on how to deal with other consistency models.

4.1 Example 1: Release Consistency (RC)

Release Consistency, as defined in [13], is one of the most commonly used relaxed consistency models. Especially in the realm of SW-DSM approaches, many systems have been built on top of this memory model, including TreadMarks [1], HLRC [20], Quarks [28], DSM-PM2 [2], and Shasta [22]. Even though, their implementation differs significantly,

as each system is built on top of a different memory update protocol, they all provide the same consistency model, i.e. provide the same memory guarantees to the application¹.

Consistency Conditions and Interpretation

When using *Release Consistency*, all memory operations are divided into synchronizing and non synchronizing operations. The first group is further split into so called *Acquire* and *Release* constructs. The former is used to gain permission to access shared data, while the latter is used to grant access to shared data to other processes.

Based on this division, *Release Consistency* is defined using the following three consistency conditions [13]:

1. Before a read or write access is allowed to perform with respect to any other processor, all previous *Acquire* processes must be performed.
2. Before a *Release* is allowed to perform with respect to any other process, all previous read and write accesses must be performed.
3. *Acquire* and *Release* accesses are sequentially consistent with respect to one another.

Informally, the two operations *Acquire* and *Release* resemble routines controlling the visibility of data on remote nodes defining a window of safe access to shared data. They are therefore enhanced with synchronization semantics, which are used to control the task structure, as only this grouping enables a combination of data and task control as required by application codes.

With regards to locks, an *Acquire* is typically combined with a lock operation, while a *Release* is combined with a corresponding unlock operation. This ensures that the data accessed during the critical section always represents the current global state (due to the *Acquire*) and that any modification is made available after the critical section (due to the *Release*).

¹The literature, especially [13], describes minor differences in Release Consistency models (Eager vs. Lazy RC). This is, however, of negligible importance for the application, and rather an important classification criterion for the actual RC protocol implementation. This is therefore not further considered in this work.

Implementation

Based on this informal description, it is clearly visible that the *Acquire* operation corresponds to the cache invalidation described above, as in both cases the data on the local nodes needs to be updated, i.e. old data needs to be invalidated. On the other side, the *Release* operations can be implemented with a write flush, as in both operations the locally modified data is pushed across the network and therefore made available.

More precisely, the implementation is as follows:

- *Acquire*
 - Perform lock operation
 - Perform cache invalidation
- *Release*
 - Flush write buffers
 - Perform unlock operation

This implementation approach satisfies all three conditions stated above: by combining the consistency enforcing operations with the respective synchronization operations the sequential consistency of *Acquire* and *Release* operations is enforced, as demanded in (3). In addition, any access to shared data is assumed to take place within the region of mutual exclusion, ensuring that is preceded by an *Acquire*, which itself is preceded by any previous *Release*. This ensures that all memory accesses to the shared region have been completed and their results are fully visible. The result is, therefore, a fully compliant *Release Consistency* implementation.

4.2 Example 2: Scope Consistency (ScC)

The *Release Consistency* model described above creates an implicit relation between *Release* and the following *Acquire*, as any data written before the *Release* must be made visible after the following *Acquire*. This relation can be loosened by introducing an explicit relationship between groups of *Acquire* and *Release* operations and restrict the visibility of data after an *Acquire* to the data written before a *Release* within the same group. This creates so called Consistency Scopes, first introduced by [10] and implemented in systems like Brazos [26] and JiaJia [7].

Consistency Conditions

As this concept of Scope Consistency is just a straightforward extension or generalization of the Release Consistency already discussed above, also its consistency conditions are very closely related. The key difference is the introduction of consistency scopes, which can be opened and closed by applications. Any read or write operation is then performed with respect to any open scope and write operations are assumed to be completed with respect to a scope on closing this scope. Using these concepts, the consistency conditions of Scope Consistency can be defined as in [10]:

1. Before a new session of a consistency scope is allowed to open at a process P, any write previously performed with respect to that consistency scope must be performed with respect to P.
2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

Informally, this definition leads to the memory behavior that memory accesses made during a scope are only guaranteed to be visible after the scope has been closed and also only within the same scope opened by another process. Any other access is not guaranteed to be visible. The important consistency enforcing mechanisms in this concept are the opening and closing of scopes, as these represent the points during a program's execution in which the visibility of data changes. Contrasted to Release Consistency, opening a scope is the same as an *Acquire* (only restricted to the scope) and closing a scope is the same as a *Release* (again restricted to the scope).

As Release Consistency, also this concept has to be used in conjunction with synchronization mechanisms in order to allow applications a useful assumption about the underlying memory behavior. In order to exploit the scope concept to a maximum, normally each lock is associated with its own scope, as it is assumed that each lock is responsible for the management of a distinct part of the overall data set. During the lock operation the respective scope is opened, i.e. the scoped *Acquire* is performed, and during the unlock operation the scope is closed again, i.e. the scoped *Release* is performed.

Using this inherent connection, the usability of Scope Consistency is drastically increased, as an explicit scope management would significantly increase the coding complexity of applications. Experiments [10] have actually shown that most codes written for Release Consistency can directly be used with Scope Consistency without any code modification, but at improved performance.

Implementation

The basic scheme of the *Scope Consistency* implementation on top of HW-DSM architectures is the same as for the *Release Consistency* implementation described above. The only difference lies in the execution of the write buffer flush and cache invalidation operations. These are enhanced by a mechanism to detect, whether a cache invalidation is necessary in relation to its scope, i.e. if a write buffer flush has been done with respect to the same scope since the last cache invalidation on the local node. Unnecessary cache invalidations can then be omitted, which not only results in reducing the execution time by the cost of this invalidation, but even more important avoids the negative impact of invalidated caches on the further application execution.

To implement this scheme it is necessary to provide a global time stamp identifying the temporal relation between cache invalidations and write buffer flushes. Such a time

stamp mechanisms can easily be implemented based on a global counter, in the following denoted as *Global Activity Counter (GAC)*. Each cache invalidation or write buffer flush is tagged with such a time stamp by reading this counter followed by an atomic increment. Before a cache invalidation, these time stamps are checked to detect unnecessary invalidations, which are then omitted.

Using this extended scheme, the write buffer flush is performed as follows:

- Perform actual write buffer flush
- Get new time stamp from the *GAC*
- Save the time stamp in the global variable *Last Global Flush*, which is distinct for each scope (*LGF-scoped*).

A scoped cache invalidation uses this information in the following manner:

- Check whether there has been a write buffer flush since the last invalidations by comparing the time stamps in *LGF-scoped* and *Last Local Invalidation (LLI)*, a node-local variable containing the time stamp of the last invalidation performed on the local node
- If *LGF-scoped* has an earlier time stamp than *LLI*, the invalidation is unnecessary and can be omitted
- Get new time stamp from the *GAC* and store it in *LLI*
- Perform actual cache invalidation

Again, this implementation satisfies the consistency constraints stated above, due to the same argumentation as above, and hence guarantees the global visibility of data within its scope.

Impact in Comparison to RC

In order to make a first attempt to quantify the difference between *Release* and *Scope Consistency*, two applications from the SPLASH-2 suite [32], namely the *RADIX* sorting kernel and the *WATER N-Squared* molecular dynamics code, have been chosen. These two codes both use a rather large number of locks and therefore provide a useful basis for an analysis of these two consistency models.

Both codes have been executed using the HAMSTER environment [24]², a framework for shared memory programming on top of loosely coupled NUMA architectures. Its current implementation is targeted towards non-coherent NUMA cluster based on the Scalable Coherent Interface (SCI) [6, 9] and implements the two relaxed consistency models as described above. It therefore also proves the feasibility and usability of the proposed approach in a real world scenario.

Table 1 summarizes the results of this set of experiments. It includes data for both codes gathered during two runs each: one using *Release* and one using *Scope Consistency*. In both

cases the number of lock, barrier, and consistency operations have been collected using HAMSTER's performance monitoring interface and are presented for two areas: the total code (including any operation needed by the NUMA-DSM framework for its own setup and initialization), and the computational core.

In both codes, the use of *Scope* consistency enables a significant reduction of the number of *Acquire* operations during the execution, mainly during the core phase. Up to 68 % of all *Acquires* were avoided and with them the related cache invalidation operations.

4.3 Capabilities and Limitations

It is important to note that the concepts presented in this work are not bound to an implementation of *Release* or *Scope Consistency* alone. By using the same basic mechanisms also other known consistency models can be implemented for non-coherent shared memory architectures relying on the invalidation and flush primitives introduced above. An important example for this is *Weak Consistency (WC)* [8], a less relaxed model which features only one type of synchronizing access (in contrast to two in RC) and is used e.g. in the POSIX thread standard [30]. In addition, it is also possible to customize memory consistency models to specific needs and application requirements leading to the wide field of application or application domain specific consistency models.

However, given certain restrictions of the underlying hardware architecture, it may not always be possible to fully exploit the advantages of specific memory models, due to deficits in certain architectures (both within the processor and or the interconnection fabric). One typical example is an implementation of *Entry Consistency (EC)*. This model is also derived from RC, but allows the user to explicitly specify the memory regions on which to apply synchronization routines. In order to make use of this additional application specific information, a corresponding non-coherent system implementation would have to be able to only invalidate this specific memory region. As stated above, however, many architectures, including any x86 architecture [11], are only capable of invalidating the complete cache, missing out on the performance chances offered by such models. In the future it might be an interesting approach to further investigate in this and to explore the performance advantages that can be expected by architectures capable of executing partial cache invalidations.

5 First Experimental Results

As already mentioned above, the technical feasibility of the proposed concepts has been shown before using a sample NCC-NUMA system based on a cluster of PCs connected via SCI [6, 9] (using SCI-PCI bridges without coherency support) and with a comprehensive software framework called HAMSTER [24] capable of supporting a large range of shared memory programming models on top of a single core. On top of this architecture both numerical ker-

²For more information, see also <http://hamster.in.tum.de/>.

	RADIX		WATER (N-Squared)	
	Release C.	Scope C.	Release C.	Scope C.
Data set size	262144 keys		1331 molecules	
Locks (total)	158	158	663	663
Locks (core)	12	12	89	89
Barriers (total)	330	330	1095	1095
Barriers (core)	24	24	24	24
Acquire (total)	621	615	2295	2183
Acquire (core)	24	18	113	36
Release (total)	621	621	2295	2295
Release (core)	24	24	113	113
Ops saved (total)	6 / 0.97 %		112 / 4.88 %	
Ops saved (core)	6 / 25 %		77 / 68.14 %	

Table 1. Properties of two applications running with both Release and Scope Consistency (on 4 nodes).

nels [25] and large scale applications from the area of medical imaging [23] have been executed successfully and efficiently.

In order to show the scalability advantages, however, and to discuss various tradeoffs associated with moving from a hardware coherent platform to a software-enforced coherent shared memory environment, it is necessary to deploy simulation. Only this will allow a fair comparison between software and hardware consistency mechanisms while keeping all other parameters identical. So far, however, this study does not account for additional architectural benefits that could be applied in the case of non-coherent architectures, which include increased processor speed and enhanced networks through simplified design requirements. This study is therefore a conservative first attempt to characterize the advantages of non-coherent architectures.

5.1 Simulation Setup and Target Applications

The simulations in the following sections have been done using the SIMT framework [29], a detailed simulator for shared memory multiprocessors, which itself is based on Augmint [17]. This system allows the simulation of architectures with an arbitrary number of processors, each with its own multilevel cache hierarchy. The coherence between the caches can be varied between a hardware coherent scheme (HCC) similar to MESI and a relaxed scheme based on the concepts of *Release Consistency* (NCC). The concrete parameters used in the following simulations are summarized in Table 2.

For this paper four applications from the SPLASH-2 suite [32], namely RADIX, WATER, OCEAN, and LU, as well as a self-coded Successive Over-Relaxation (SOR) have been used. Each code has been executed with a working set size suitable for the simulation system. In addition, the LU decomposition has been executed using three different working set sizes in order to also be able to study the impact of the working set size. The concrete working set sizes for each application is given in Table 3.

³The rather small cache sizes have been chosen to match the relatively small working set sizes of the test applications which are necessary to allow the simulation.

Number of Processors	2–32
Levels in Cache Hierarchy	2
Size of L1 cache	8 KB ³
L1 cache line size	32 Bytes
L1 associativity	2-way
L1 access latency	1 cycle
Size of L2 cache	64 KB ³
L2 cache line size	32 Bytes
L2 associativity	2-way
L2 access latency	10 cycles
Main memory access latency	100 cycles
Remote access latency (NUMA)	100–2000 cycles
NUMA memory distribution	Round-Robin

Table 2. Simulation parameters.

5.2 Application Scalability

The first set of experiments takes a look at the scalability of codes running on the two different shared memory architectures. For this purpose the execution of the benchmark suite described above was simulated with a varying number of processors ranging from 2 to 32. In addition the base architecture was varied with respect to the assumed latency of remote memory accesses. Both a UMA scenario with equal access times for local and remote memories and a loosely coupled NUMA environment were investigated. In the latter case, a difference in latency of a factor of 20 is assumed, which corresponds to the actual difference in loosely coupled NUMA environments as it is e.g. given in SCI-based clusters.

In order to investigate the difference between the HCC and the NCC case, Figure 2 shows the results of the experiments as a ratio between the execution times of the HCC and the NCC case. Values lower than 1 show a performance benefit for HCC, while values higher than 1 indicate a faster execution in an NCC environment.

As a general trend, the graphs show a rising advantage for the NCC scheme with increasing system sizes. Especially

Application	Working set
SOR	512x512 dense matrix
RADIX	262144 keys
WATER	512 molecules
WATER	66x66 ocean segments
LU-SM (small)	128x128 matrix
LU-MD (medium)	256x256 matrix
LU-LG (large)	512x512 matrix

Table 3. Simulated applications and their working set sizes.

the SOR code is able to benefit from the NCC environment with a performance improvement of more than a factor of 2 on 32 processors in both the UMA and the NUMA scenario. This can be attributed to its regular memory access behavior and low consistency requirements. In this case, the hardware scheme triggers a significant amount of unnecessary updates and invalidations, which are avoided in the NCC scheme.

Comparing the UMA and the NUMA results, it can be seen that the results in the latter scenario seem less regular than in the UMA system. This is especially obvious when looking at results for RADIX and LU, which do not provide a clear trend across all numbers of processors. This can be most likely attributed to the fact that in this case the global memory is distributed transparently without any NUMA data locality optimizations. The induced memory access pattern and with it the performance characteristics can therefore significantly vary between different process configurations. It is expected that this behavior will change after applying data layout optimizations on all codes.

Despite these irregularities, though, NCC outperforms its HCC counterpart in all of these cases. In addition, the results of LU show that the benefit increases for larger data set sizes, most likely due to increased pressure on the memory hierarchy, which can be more easily handled by the NCC approach.

5.3 System Scalability

Besides the scaling properties of applications on top these architectures, it is also useful to look at the scaling properties of the architectures themselves. A good indicator for this is the latency used for remote memory accesses in a NUMA scenario, as this latency depends on the physical distance between processors and on the requirements set forth for the interconnection fabric.

In order to investigate this, the number of processors has been fixed, while varying the remote access latency from 100 cycles (which is again providing the properties of an UMA architecture) to 2000 cycles, a value which is typical for SAN-based cluster interconnects.

As expected, with a rising remote memory access latency, the absolute execution time increases in all cases almost linearly. Interesting, however, is again the ratio between the HCC and NCC execution times, which is depicted in Figure 3 for both a 2 and a 32 processor system. For most codes, the ratio increases with rising remote access latencies and has

thereby shifted in favor of the NCC architectures.

The only notable exceptions can be observed in the 32 processor system for RADIX, which slightly degrades in larger systems despite a large increase of the ratio HCC:NCC in the two CPU scenario, and the LU decomposition with the small working set size. However, in the former case, the decrease is only very slightly and the results still show a significant advantage of NCC over HCC, and in the case of LU, the behavior is reverted for the larger data set sizes indicating benefits for NCC in large scale environments with respect to both system and data sizes.

5.4 Performance Summary

In summary, it can be noted that the NCC scheme is clearly beneficial in larger systems and in the by itself more scalable NUMA scenario, while it is mostly outperformed by the hardware coherent scheme in small UMA systems. This shows that hardware coherency mechanisms, as they are deployed in current systems, very well support modest sized SMPs, but hinder the scalability both in terms of number of CPUs and with respect to increasing NUMA latencies. Here, the NCC approach can provide an efficient alternative and can extend the scalability of the underlying architecture.

6 Conclusions and Future Work

One of the main factors limiting the scalability of current shared memory architectures is their support for a global coherent memory abstraction. This feature is motivated by the user's demand for a programming model which is close to the traditional sequential one and hence allows an easy conceptual transfer of existing codes to the shared memory platform. It can, however, easily be substituted by relaxed consistency models, which are known to have only a minor impact on the programmability. This has been shown by numerous projects in the SW-DSM area. As a result, it is possible to drop the hardware coherency mechanisms, which limit the scalability, resulting in more scalable and easy-to-build systems.

This claim has been validated in this work using a detailed simulation framework. Compared to their hardware coherent counterparts, non-coherent systems, combined with an appropriate software implementation of a relaxed consistency model based on cache invalidations and write buffer propagations, show a performance benefit in most cases. This advantage increases both with larger system size and with increased NUMA latency and hence potentially larger system configurations. This indicates the excellent application and system scalability properties of non-cache-coherent systems.

Based on these encouraging results, further research will be directed towards the investigation of architectural properties which allow a more efficient implementation of relaxed consistency models in such environments. Especially the support for partial cache invalidations is of interest in this respect, as this would allow to further reduce the impact of the necessary cache flushes. This would also open the door for further consistency enhancements, like they are e.g. done

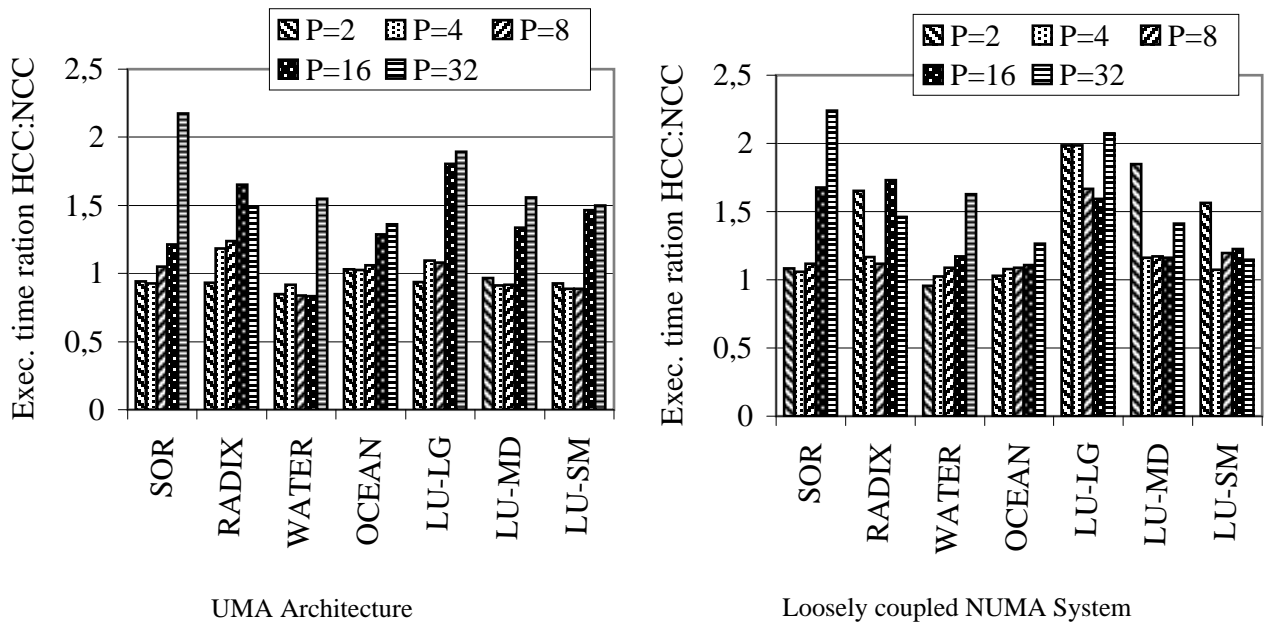


Figure 2. Execution time ratio HCC:NCC on several system sizes — left: remote latency 100 cycles (UMA/SMP), right: remote latency 2000 cycles (loosely coupled NUMA).

in *Entry Consistency* [4]. In addition, future work will focus on the impact of dropping consistency mechanisms from the overall system design, like the possibility for increased clock frequencies for processor/cache busses, as cache updates no longer need to be propagated system wide. An assessment of the concrete impact, however, requires a more detailed simulation of the implementation of the corresponding bus systems.

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. Appollonia, J. Méhaut, R. Namyst, and Y. Denneulin. SCI and distributed multithreading: the PM2 approach. In H. Hellwagner and A. Reinefeld, editors, *Proceedings of SCI-Europe '98, a conference stream of EMMSEC '98*. Cheshire Henbury, Sept. 1998. ISBN: 1-901864-02-02.
- [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. Breshad 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] M. Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. Technical Report TR-95-01, Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA, Oct. 1995.
- [6] 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.
- [7] W. Hu, W. Shi, and Z. Tang. JiaJia: An SVM System based on a New Cache Coherence Protocol. In *In the Proceedings of High Performance Computing and Networking (HPCN-Europe)*, volume 1593 of *LNCS*, pages 463–472, Apr. 1999.
- [8] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.
- [9] 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.
- [10] L. Iftode, J. Singh, and L. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory of Computer Systems*, 31:451–473, 1998.
- [11] Intel Corporation. *Intel Architecture Software Developer's Manual for the PentiumIII*, volume 1–3. published on Intel's developer website, 1998.
- [12] W. Karl, M. Leberrecht, and M. Schulz. Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE. In A. Sivasubramaniam and M. Lauria, editors, *Proceedings of Workshop on Communication and Architectural Support for Network based Parallel Computing (CANPC) (held in conjunction with HPCA)*, volume 1602 of *Lecture Notes in Computer Science (LNCS)*, pages 196–210, Berlin, 1999. Springer Verlag.
- [13] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, Jan., 1995.
- [14] 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)*, pages 241–251, June 1997.
- [15] D. Lenoski, J. Laudon, T. Joe, D. N. ira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and

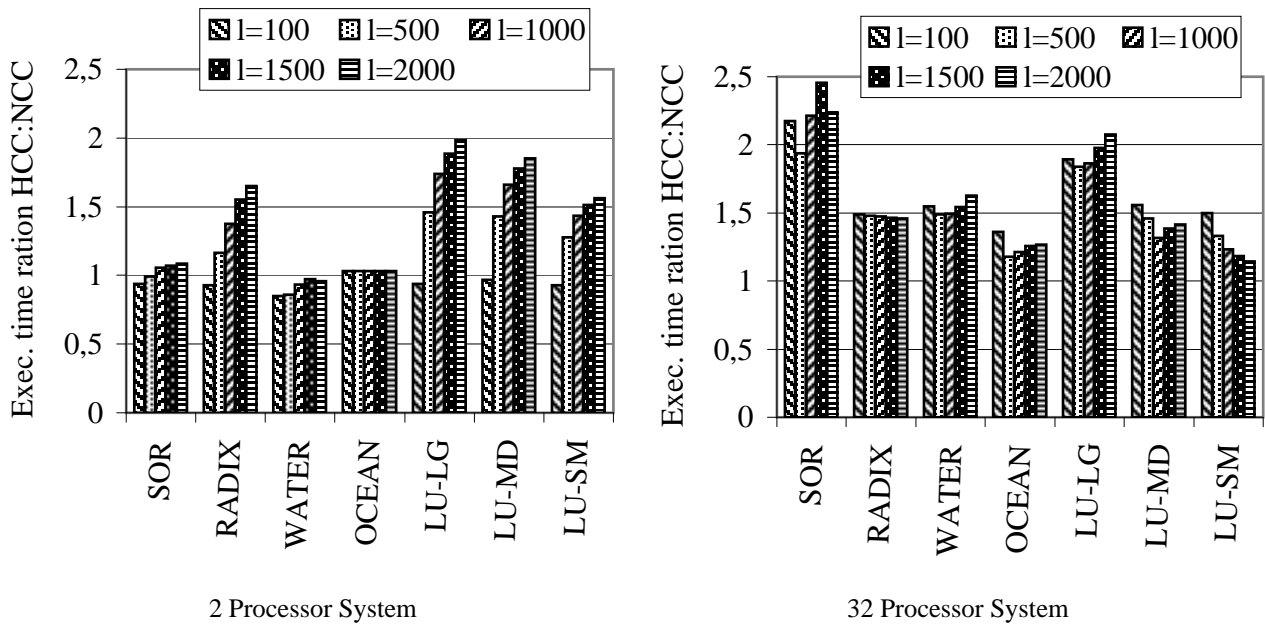


Figure 3. Execution time ratio HCC:NCC using various NUMA access latencies (100–2000 cycles) — left: on 2 processors, right: on 32 processors.

performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992.

- [16] K. Li. IVY: A shared virtual memory system for parallel computing. In *In Proceedings of the International Conference on Parallel Processing (ICPP)*, volume 2, pages 94–101, 1989.
- [17] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, October 1996.
- [18] B. Nitzberg and V. LO. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–59, Aug. 1991.
- [19] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 348–354, 1984.
- [20] M. Rangarajan, S. Divakaran, T. Nguyen, and L. Iftode. Multi-threaded Home-based LRC Distributed Shared Memory. In *In the Proceedings of the 8th Workshop on Scalable Shared Memory Multiprocessors (held in conjunction with ISCA)*, May 1999.
- [21] 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.
- [22] 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, 250 University Avenue, Palo Alto, California 94301, USA, Nov. 1996.
- [23] M. Schulz. Evaluating a Novel Hybrid-DSM Framework in Real-World Scenarios Using Medical Imaging Applications. In *Proceedings of the 18th PARS Workshop*, October 2001.
- [24] M. Schulz. *Shared Memory Programming on NUMA-based Clusters using a General and Open Hybrid Hardware/Software Approach*. PhD thesis, Technische Universität München, July 2001.
- [25] 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/>.
- [26] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Workshop*, pages 95–106, Aug. 1997.
- [27] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Konthanas, S. Parthasarathy, and M. Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.
- [28] M. Swanson, L. Stoller, and J. Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proceedings of the Workshop on High-Level Programming Models and Supportive Environments (HIPS) (held in conjunction with IPPS)*. IEEE, Apr. 1998.
- [29] 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)*, pages 349–354, September 2001.
- [30] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating Systems Interface (POSIX) — Part 1: System Application Interface (API)*, chapter Section 2.3.8, General concepts/memory synchronization. IEEE, 1995 edition, 1996. ANSI/IEEE Std. 1003.1.
- [31] C. Willard. Superdome: Hewlett-Packard Extends Its High-End Computing Capabilities. An IDC White Paper, 2000.
- [32] 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)*, pages 24–36, June 1995.
- [33] WWW:. Sun Servers (Hardware, SUN Enterprise (TM) servers) . <http://www.sun.com/servers>, Feb. 2001.