Towards Compliant Distributed Shared Memory

Katrina E. Falkner, Henry Detmold, David S. Munro & Travis Olds
Department of Computer Science
The University of Adelaide
Adelaide, S.A., 5005, Australia
{katrina,henry,dave,trav}@cs.adelaide.edu.au

Abstract

There exists a wide spectrum of coherency models for use in distributed shared memory (DSM) systems. The choice of model for an application should ideally be based on the application's data access patterns and phase changes. However, in current systems, most, if not all of the parameters of the coherency model are fixed in the underlying DSM system. This forces the application either to structure its computations to suit the underlying model or to endure an inefficient coherency model.

This paper introduces a unique approach to the provision of DSM based on the idea of compliance. Compliance allows an application to specify how the system should most effectively operate through a separation between mechanism, provided by the underlying system, and policy, provided by the application. This is in direct contrast with the traditional view that an application must mold itself to the hard-wired choices that its operating platform has made. The contribution of this work is the definition and implementation of an architecture for compliant distributed coherency management. The efficacy of this architecture is illustrated through a worked example.

1. Introduction

The principal motivations behind distributed systems design are the promises of resource sharing, computational speed-up and increased reliability. However, delivering these benefits typically involves a trade-off between application programming complexity and underlying system complexity. In DSM systems, the aim is to simplify the programmer's task by presenting a single logical address space to the application. The logical address space is an abstraction over memory where each node's physical memory acts as a cache of this single space.

To improve performance, DSM systems allow data to be replicated across local caches. The application's requests to read and update data in the logical address space result in accesses and modifications to data in these caches. To avoid problems related to concurrent multiple updates and out of date information, a DSM system will use a coherency protocol that ensures the application has a consistent view of memory.

A large number of DSM models have been proposed, designed and implemented. Primarily these differ in their coherency model. (See [2] for a detailed introduction). These effectively form a spectrum trading off simplicity against efficiency through increased programming complexity. At one end of the spectrum, a simple sequential consistency model [12] effectively serialises concurrent update requests across nodes. Conversely, the relaxed consistency models [9] vary the extent to which programmer controlled inconsistencies are allowed to occur in an effort to increase performance through increased concurrency. In addition to the level of relaxation, DSM models can differ in the level of coherency grain, the DSM program interface and requirement for application cooperation, and memory update and writer model. This extensive range of sophisticated coherency models has complicated the otherwise elegant abstraction of DSM.

The choice of an appropriate coherency model is undoubtably dependent on the application's data access behaviour and phase changes in that behaviour [1, 3]. An application exhibiting localised write access patterns may benefit from a relaxed model where in message complexity on update is reduced. On the other hand, a loosely coupled system with a dispersed write access pattern may benefit from the management simplicity of a broadcast or eager update model even with the overhead of increased message complexity.

In order to accommodate application specific requirements many systems have defined adaptive policies [3, 6, 8, 14, 17] typically adapting along one dimension, such as choices between single and multiple writer policies or through aggregation of data to reduce message frequency. These systems present a fixed model of adaption, thereby

Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)
0-7695-1582-7/02 $17.00 © 2002 IEEE

limiting their appropriateness to a subclass of applications. In addition, it is the DSM system itself that makes the choice for the application based on its past behaviour and using a continuous function, precluding the ability to adapt across phase changes. This is in contrast to providing the application the opportunity to make its own decisions based on what it knows will be its future behaviour. It is by providing this opportunity that compliance moves system development from a one size fits all approach to an optimal application fit.

In a compliant system architecture, system components can evolve according to the current state of their environment [15]. Compliance dictates a separation of policy from mechanism, and thereby enables the development of new policy from run-time information. In conventional software architectures, the operational abstractions are constructed to meet the average predicted needs of the majority of cases. Policies in these architectures are derived from simulations and benchmarking a cross-section of applications that are intended to be run on the system. These policies are then rigidly fixed into the system. Such architectures are consequently structured to perform well under the benchmark conditions. The corollary is that unless optimisations for particular cases are accommodated by this average case then individual applications only occasionally run optimally, and even then do so only by accident. Architectural compliance allows the support architecture to be varied dynamically to meet the changing demands of the application. Supporting compliance at the language level facilitates the development of applications that are capable of exploiting and adapting the run-time system to support their needs.

A compliant system or language is not a panacea; the application must be aware, at composition time, of the supplied mechanisms and derive and apply appropriate policies, both statically and dynamically (e.g. via run-time reflection), at points in the computation where it ascertains behaviour change. This may complicate application development, but achieves the goal of providing a more suitable support environment.

Compliance in a DSM system allows an application to specify its own policies with full knowledge of its actual access patterns and data layout. Thus, it is possible for an application to utilise a more relaxed model for localised data, a different model for other data, and, in addition, have the ability to modify its policies based on what phase of its computation it is in. In an extreme case, the coherency policy can be unique for each shared object. Pragmatically, compliance does not require the application to make decisions on every memory access, it may, for example, provide the mechanism for an application to select what decisions it has an interest in making.

We have constructed a compliant architecture for a DSM system [5], which supports compliant scheduling, coherency management, locking and thread placement. This architecture currently supports experimentation into appropriate compliant policies for distributed systems, and the coordination of policies such as memory management [16] and concurrency. This paper describes that compliant architecture and the design and implementation of compliant coherency. This architecture, a distributed variant of the ProcessBase [15] environment, is designed to support compliance in a distributed setting, introducing a flexibility unseen in existing distributed languages. The efficacy of the compliant coherency model is demonstrated through a worked example, showing how an application can exploit compliance to specify a coherency model on top of primitive mechanisms, and, further, how it can dynamically change its policies according to its data layout and access patterns.

2. Distributed Compliant Architectures

ProcessBase [15] is a compliant language that utilises the modern practice of an abstract machine model. Designed to support process modelling, ProcessBase has the properties of strong typing, first class procedures and types, hyperprogramming [11] and persistence [15]. ProcessBase supports the development of libraries (including thread, synchronisation and I/O extension) which enables the production of a compliant library.

ProcessBase differs from most programming languages in that it supports flexible policy specification; services such as concurrency and distribution are not generalised but can be optimised to fit an application's specific needs. The principal benefits here are flexible modelling, simpler semantics and a potential for improved performance. This view is predicated on the belief that an application understands its own behaviour better than the support environment, and is therefore more capable of adaptations.

The distributed ProcessBase environment (DPB) consists of four distinct architectural layers.

- The application layer, which views the remaining layers as a unified system through a single logical address space.
- A layer of interpreter nodes that perform the computation.
- A communications layer that facilitates interpreter-to-interpreter communication.
- An object store layer, providing orthogonally persistent storage.

The unit of distribution in DPB is the thread, provided through a compliant threading library. Each interpreter node has a unique site ID that is used to direct communications.

Figure 1 illustrates the DPB architecture. Core mechanism exists at each level; policy to control the invocation of a particular mechanism may exist at any higher level. Mechanism is invoked through a *downcall*[1], allowing information about policy to be shared with a lower layer. Policy information can be requested through an *upcall* which exists as a form of interrupt, triggering the execution of policy code written in the ProcessBase language. Upcalls allow information about mechanism to be shared with higher levels. Information may be passed between components at the same level using lateral calls (conventional function calls).

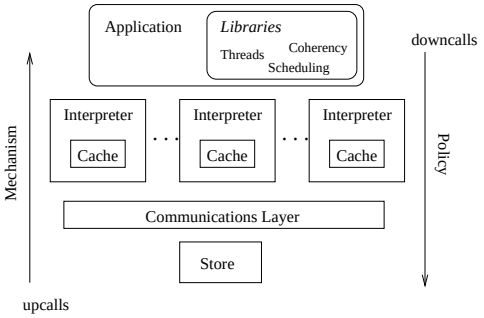

Figure 1. The DPB architecture, allowing computation to be dispersed across a set of nodes.

The definition of mechanism is static; mechanisms are the core activities that define the services available to the application. For example, primitives may include the copying of an object into a local cache or the sending of a communications message. The addition of compliant components requires the identification of any further mechanism and the integration of this mechanism into the appropriate layer. Policy is dynamic, and may use the provided mechanisms to reach its aim. Mechanism can, in turn, invoke policy in order to receive direction.

In order to explore compliance in a practical setting, a pragmatic limit must be placed on the available mechanisms. This instance of the DPB architecture only supports dynamic policy specification at the topmost layer; mechanisms at all lower layers must perform upcalls to this layer in order to receive policy direction.

An application may choose to associate a handler function with an upcall in order to provide policy for that upcall; it is this function that is invoked upon interrupt. The set of managed upcalls can be modified dynamically during computation execution, this information can be passed to lower layers through a downcall. Libraries can be used to provide policy management functions to a range of applications, although an application is free to define their own policy. If a handler function is not associated with an upcall, the core mechanism is used subject to a default policy.

When an object is created it is given a local cache address (CA). When a reference to that object is shared across a node boundary, that object is given a distributed address (DA). The structure of a distributed address is dependent on policy defined at the interpreter level (using mechanisms in the communications layer). Thus, the structure of a distributed address can be chosen to reflect other policies such as collection and coherency. However, a DA is assumed to be globally unique. Mechanism is provided to enable a site ID to be encoded into a DA; policy enables interpretation of this encoding, potentially as either a home-node or a creation site indicator.

When an object becomes persistent it is provided a globally unique persistent ID (PID). An object with a PID or a DA can be accessed as a shared object.

3. Implementation

To implement a compliant coherency subsystem it is necessary to identify what is mechanism and what is policy. Coherency can be defined as those operations that must be performed to maintain consistency upon access to shared objects; hence, mechanism can be defined as the object access, and policy as those decisions that must be made to control these accesses.

Mechanisms are required at the interpreter and store levels to perform low level operations at the request of a higher-level policy. The core mechanisms supported by the DPB fault objects, manipulate objects at a low-level and request remote communications. Policy, implemented at the application level, is used to control these mechanisms. Mechanism can also be invoked laterally by remote nodes through the communications layer.

The communications layer is itself compliant. It defines a core set of messages that it is able to send, including request and response pairs for faults, locks, unlocks, updates and invalidates. A request to send a message is performed through a downcall to the communications layer from either the interpreter or application layers. It is up to the current policy strategy as to which messages are used and which have an effect on the state of the distributed computation.

Invocation of coherency protocol management occurs through the following operations:

- A dereference on a shared object. The object must be located and faulted into the local cache. Policy code can decide whether the object can be replicated, and if not, whether the request can be satisfied or whether it should be blocked. Mechanism primitives are used to copy the object and invoke the appropriate communications operations.

[1] A downcall is realised as the atomic execution of an abstract machine instruction.

- A write on a shared object. Policy code defines what happens on a write request, for example whether a lock is provided or required, whether the lock is immediately released (eager) or delayed (lazy). Mechanism primitives are used to manipulate low-level locking.

- A release of a lock. Policy code defines whether this request is obeyed and whether an update or invalidate policy is used. Mechanism primitives are used to manipulate locking information and to support the implementation of the update/invalidate policy.

3.1 Locking

DPB supports a progressive locking mechanism that may be controlled by compliant policy. The thin and thick lock [4] model allows an object's locking mechanisms to be in one of two phases, namely thin and thick. The first time that an object is locked it enters the thin phase; this change in status can be recorded using a single bit flag. Additional lock requests while the object is locked convert the thin lock to a thick lock. A thick lock provides access to mechanisms that can be used to handle multiple lock requests such as request queues and mutex structures. The thick phase requires that the object be annotated with a pointer to its thick lock. At some later stage (such as when there are no pending lock requests) the lock may progress back to a thin lock.

Thin to thick lock progression facilitates mechanism support for efficient locking in DPB. DSM systems that use this model typically integrate the policy determining when conversion will be done into the mechanism implementation. DPB enables the coherency policy to compliantly make these decisions by providing downcalls to enable lock management to react to application phase changes.

In non-compliant systems a thick lock has a predefined format with support for a static set of locking operations. In the DPB environment, the structure of the thick lock is defined by the compliant policy, potentially on a per-object basis. The thin lock format is statically defined as part of the mechanism structure; a fully compliant system would extend dynamic definition to this layer.

The format for a DPB object is shown in Figure 2. Each object has a bit flag that is set for a local lock. If an additional lock request is received and this flag is set then an upcall is made to request an appropriate thick lock object. The address of the thick lock is inserted into the lock word at the end of the object. A further upcall is made upon lock release. The initial implementation of this model [4] prohibited backwards conversion in order to avoid thrashing, however in a compliant system, policy may be used to determine whether to convert the lock back to a thin lock.

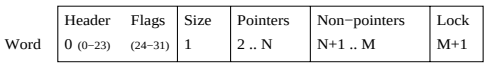

Figure 2. Format of a DPB object.

4. Application Example

QuickSort is a common DSM benchmark [10] that recursively sorts an array of integers by rearranging subarrays around a fixed pivot. A multithreaded implementation of QuickSort exhibits two phases and two data access patterns. The first phase involves initialisation of the data, generally using random values. This can be performed using a multithreaded coordination pattern which performs multiple updates throughout the array in a non-localised manner. The second phase involves sorting of the array, where each thread sorts subarrays of data that are assigned to it. The sorting routine does not require write access to data being sorted by other nodes, defining a localised write pattern.

Two main forms of data are used in the QuickSort algorithm: coordination data and the array data. A centralised task queue is used to hold the indexes for partitions waiting to be sorted. Subbarray tasks are dequeued and queued by the computation threads within the application. Access to the queue is non localised and is used throughout phase two. In addition to the queue, locks and barriers are used to control the flow of execution between the two phases and at algorithm completion.

In the algorithm's first phase, a pool of threads is created to coordinate data initialisation. After this phase, a second pool is created to service sorting requests. During this second phase, the policy for the array is changed. Commonly, once the partition size reaches a defined threshold the subarray is sorted locally using a non-partitioning sorting algorithm, in this case bubblesort. This forms a third phase of computation, but has an identical access pattern to the second.

The compliant policy supporting this algorithm employs both eager [7] (ERC) and lazy [9] (LRC) release consistency strategies. The default strategy assumes that all shared objects will be handled by ERC. The policy module provides interface functions that can be used to set the coherency protocol for an object or, given an array, over a subset of the array's data. These functions can be invoked at any stage to alter the selected coherency protocol.

Figure 3 shows the ProcessBase code for the QuickSort algorithm, illustrating four invocations of the module interface. In lines (1) and (3), the application uses the module interface to define that the objects passed as parameters be handled using ERC, in this case new objects added to the task queue, and the queue itself. Line (2) applies an ERC selection over an array (vector), selecting the entire array by identifying its lower and upper bounds. Line (4) modi-

```
    type qObj is view [ ... ]
    let queue <- loc(nil(qObj))
    let newObj <- fun(lwb: int; upb: int) -> qObj
    begin
        ...
(1)     chooseERC(any(obj))
        obj
    end

    ...

    let qs <- fun (data : *loc[int]);
    begin
(2)     chooseERC_V(any(data),lwb(data),upb(data))
(3)     chooseERC(any(queue))
        let threadId <- loc(0)
        for i <- 1 to threadNumber do
            threadId := start(initThread)
        barrierInit()
(4)     chooseLRC_V(any(data),lwb(data),upb(data))
        enqueue(newQObj(lwb(data),upb(data)))
        for i <- 1 to threadNumber do
            threadId <- start(qsThread)
    end
```

Figure 3. An extract from the QuickSort ProcessBase program.

fies the strategy selection for this data to be LRC, reacting to a computation phase change.

When a shared object is identified by the application through this interface the address of a tag is placed in the object's lock word, indicating the selected protocol. This action is performed using a downcall, which takes two integer values as parameters[2]. The downcall sets the lock word in the object at the location specified by the first integer to the value specified by the second integer. A single tag object is created and used for each type of policy to avoid unnecessary duplication.

An attempted write of a shared object will first attempt to lock the object. This is done by locating the manager of the object's lock and sending a lock acquire message to that node. Receipt of this message invokes mechanism which determines whether the object has been previously locked and if not, moves the object into the thin locking phase. If the object is already in this phase, an upcall is invoked, with the object referenced by the object's lock word passed as a parameter.

The policy module maintains a tree structure of locks, sorted by the address of their respective objects. The lock request upcall determines whether it has been passed a tag identifier or a locking object. The tag object indicates a locking phase change, and also identifies what kind of locking object must be created for the object. For each model, the locking object maintains the current lock holder, and a queue of the site IDs[3] for nodes that are waiting for the lock. The ERC model also maintains the previous lock holder. A downcall is made to set the address of the locking object into the lock word. If the upcall has been passed a locking object, it uses the defined policy for that locking object to perform the lock request.

After a write has completed, mechanism code attempts to release the lock. An upcall is invoked with the address of the object's locking object. If the locking object is managed by ERC, then the release is accepted and a suitable mechanism (update/invalidate) is selected to maintain a consistent view. Alternately, if the object is managed by LRC, the release is delayed until the next acquire. If the locking object shows that there are no sites waiting on the lock then a downcall is invoked that will overwrite the address of the locking object with the address of the tag object for the required policy (converting it back to a thin lock).

The policy code has access to all objects that are locked and the information contained in their locking objects through its tables. It can use this information to perform its own policy changes or to request information from the application using lateral calls. At any point, the policy can be used to gather information about the locking events and models of the system and its data access patterns. The interpreter level has access to this data on a per-object basis (through the lock word), and can thus manipulate the addresses of objects if required.

This example covers just one aspect of coherency in a DSM system, management of lock acquisition and release. Object faulting and the creation of a consistent view can also be structured in such a way. For example, using an invalidate-based model on objects that have moved from a shared to a non-shared phase (where an update-model was previously chosen), or allowing replication for immutable data and prohibiting it for mutable data in a highly restricted coherency system.

5. Related Work

Compliance has been examined relative to a separation of operating system policy and mechanism [13], allowing user-level per-application level selection of behaviour. The model of compliance chosen in this work is at a higher level than that involved in operating system research.

Adaptive DSM systems have a similar aim to this work: to provide more suitable coherency models on a per-application basis. Amza *et al* [3] describe an adaptive DSM system that supports a static set of adaptions, namely single

[2] Allowing pointers to exist as integer values at the application level downgrades the type safety of the ProcessBase system. The alternative, object references within system data structures at the application level, prohibits the automatic collection of such data. The DPB is currently being extended to support a form of *weak pointer*; these pointers can be ignored if they represent the only existing reference to an object.

[3] In a system with a compliant thread migration policy, site IDs might not be sufficient. In this case, a globally unique thread ID must also be recorded.

writer to multiple writer, invalidate to update and aggregation of pages to larger transfer units. This study verifies that adaptation is a valid technique but only provides a static defined set of available adaptations, which may or may not be beneficial to the application. As stated in [3], some applications achieved worse performance due to the adaptive policies. Similar work has been performed by Monnerat and Bianchini [14], Keleher *et al* [8] and Speight and Bennett [17].

Carter *et al* [6] present a system where the application code is annotated to select a coherency model. Once specified, the choice of policy can not change and this choice is again restricted to a static set.

These policies can not be specialised by the application, neither can they be affected directly by the application's phase changes and dynamic access patterns. In these systems, the application is restricted to a defined suite of protocols, a static specification of which protocols to use and the adaptive policies provided by the DSM system itself.

6. Conclusions and Future Directions

This paper has introduced an architecture for a compliant DSM system that supports complete separation of mechanism and policy, and the movement of policy into the domain of the application. Coherency models benefit particularly from this approach as the choice of model is ideally based on the application's data access patterns and phase changes.

The ProcessBase compliant environment allows high-level policy specification for lower-level mechanisms. Not all policy choices have to be made at the application level, indeed the application is free to utilise pure mechanisms without specifying policy. A compliant system is able to support adaptive polices such as those specified in Section 5, while still enabling dynamic policy decision.

The current instance of the DPB architecture supports application-level compliance, providing a pragmatic environment for limited compliance experimentation. Future research will be targeted at multi-level compliance; supporting dynamic policy and mechanism specification at all architectural levels.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proc. of the Second Int. Sym. on High-Performance Computer Architecture*, pages 26–37, February 1996.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Distributed Shared Memory. In *Proc. of the IEEE, Special Issue on Distributd Shared Memory*, volume 87, pages 467–475, March 1999.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronisation for Java. In *Proc. of the SIGPLAN'98 Conf. on Programming Language Design and Implementation*, pages 258–268, June 1998.

[5] W. Brodie-Tyrrell, H. Detmold, K. Falkner, M. Lowry, R. Morrison, D. Munro, S. Norcross, T. Olds, Z. Tian, and F. Vaughan. Distributed ProcessBase Design Document. Technical Report TR2001-01, University of Adelaide, February 2001.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int. Sym. on Computer Architecture*, pages 15–26, May 1990.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. An evaluation of software distributed shared memory. In *Proc. of the 19th Int. Sym. on Computer Architecture*, pages 13–21, May 1992.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Int. Sym. on Computer Architecture*, May 1992.

[10] P. J. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter USENIX Conf.*, pages 115–131, January 1994.

[11] G. N. C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems*. PhD thesis, University of St Andrews, 1992.

[12] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[13] K. Mayes. Trends in Operating Systems Towards Dynamic User-Level Policy Provision. Technical Report TR-93-9-1, University of Manchester, September 1993.

[14] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proc. of the Fourth Int. Sym. on High-Performance Computer Architecture*, February 1998.

[15] R. Morrison, D. Balasubramaniam, R. Greenwood, G. Kirby, K. Mayes, D. Munro, and B. Warboys. A compliant persistent architecture. *Software, Practice & Experience*, 30(4):363–386, 2000.

[16] D. Munro, K. Falkner, M. Lowry, and F. Vaughan. Mosaic: A Non-intrusive Complete Garbage Collector for DSM Systems. In *Proceedings of the Third International Workshop on Software Distributed Shared Memory*, pages 539–546, May 2001.

[17] W. E. Speight and J. K. Bennett. Using Multicast and multithreading to reduce communications in software DSM systems. In *Proc. of the Fourth Int. Sym. on High-Performance Computer Architecture*, February 1998.