CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems

Computation Structures Group Memo 414
October 1998 (Revised: March 1999)

Xiaowei Shen, Arvind and Larry Rudolph
xwshen, arvind, rudolph@lcs.mit.edu

To appear in Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing, Rhodes, Greece, June 1999.

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems

Xiaowei Shen, Arvind and Larry Rudolph
Laboratory for Computer Science
Massachusetts Institute of Technology
xwshen, arvind, rudolph@lcs.mit.edu

Abstract

An adaptive cache coherence protocol changes its actions to address changing program behaviors. We present an adaptive protocol called Cachet for distributed shared-memory systems. Cachet is a seamless integration of several micro-protocols, each of which has been optimized for a particular memory access pattern. Cachet embodies both intra-protocol and inter-protocol adaptivity, and exploits adaptivity to achieve high performance under changing memory access patterns. Cachet is presented in the context of a mechanism-oriented memory model, Commit-Reconcile & Fences (CRF), which is a generalization of sequential consistency and other weaker memory models in use today. A protocol to implement CRF is automatically a correct implementation of any memory model whose programs can be expressed as CRF programs.

1 Introduction

Shared-memory programs have various access patterns, and empirical evidence suggests that no fixed cache coherence protocol works well for all access patterns [1, 4, 5, 12]. For example, an invalidation-based MESI-like protocol assumes no correlation between processors that access the same address before and after a write operation. Furthermore, the protocol behaves as if the processor that modifies an address is likely to modify the same address again in near future. Needless to say, such a protocol is not desirable for many other common access patterns.

Previous research has classified memory access patterns into a number of specific sharing patterns, e.g., the producer-consumer pattern and the migratory pattern [1]. Adaptive shared-memory systems allow multiple coherence protocols to run at the same time, or allow the coherence protocol to adapt to some identifiable access patterns [3, 11]. The main difference in these systems is regarding what and how access patterns are detected. Some heuristic mechanisms have been proposed to predict and trigger appropriate protocol behavior [8].

The implementation of an adaptive cache coherence protocol involves two issues: what adaptivity can be embodied in the protocol, and how and when such adaptivity can be invoked. This paper addresses the first issue and attacks the adaptivity problem from a new perspective. It proposes a cache coherence protocol, Cachet, that provides a wide scope for adapting to changing program behaviors. Cachet is especially suitable for large Distributed Shared-Memory (DSM) systems, and applicable to a wide variety of programmer-centric memory models.

Cachet consists of three *micro-protocols*, each of which is optimized for some common access pattern. Even though each micro-protocol is a complete protocol, we refer to them as such because they constitute parts of the full adaptive protocol. When something is known about the access pattern for a particular address region, the system can employ an appropriate micro-protocol for that region. Moreover, there is scope for adaptive behavior within each micro-protocol based on *voluntary rules* which can be triggered by some observation regarding the access pattern. Such actions can affect performance but not the correctness and liveness of the protocol. Another level of adaptivity is introduced when, based on some observed program behavior or resource situation, the system automatically switches from one micro-protocol to another in a seamless and efficient manner.

Cachet is not just a toy example of an adaptive protocol. It is a sophisticated protocol with some interesting properties. For example, Cachet can function in the presence of limited directory resources in a DSM system, simply by switching to an appropriate micro-protocol when it runs out of directory space. Not all interesting properties of Cachet are necessarily a consequence of adaptivity. For example, Cachet allows write operations to be performed without the exclusive ownership. This not only alleviates potential cache thrashing due to false sharing, but also reduces the average latency of write operations. These advantages make Ca-

Processor Rules				
Rule Name	**Instruction**	**Cstate**	**Action**	**Next Cstate**
CRF-Loadl	Loadl(a)	Cell(a,v,Clean)	retire Loadl	Cell(a,v,Clean)
		Cell(a,v,Dirty)	retire Loadl	Cell(a,v,Dirty)
CRF-Storel	Storel(a,v)	Cell(a,-,Clean)	retire Storel	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	retire Storel	Cell(a,v,Dirty)
CRF-Commit	Commit(a)	Cell(a,v,Clean)	retire Commit	Cell(a,v,Clean)
		$a \notin sache$	retire Commit	$a \notin sache$
CRF-Reconcile	Reconcile(a)	Cell(a,v,Dirty)	retire Reconcile	Cell(a,v,Dirty)
		$a \notin sache$	retire Reconcile	$a \notin sache$

Background Rules				
Rule Name	**Cstate**	**Mstate**	**Next Cstate**	**Next Mstate**
CRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,Clean)	Cell(a,v)
CRF-Writeback	Cell(a,v,Dirty)	Cell(a,-)	Cell(a,v,Clean)	Cell(a,v)
CRF-Purge	Cell(a,-,Clean)	Cell(a,v)	$a \notin sache$	Cell(a,v)

Figure 1. CRF Memory Model Rules

chet more suitable than other protocols for scalable DSM systems.

Sophisticated cache coherence protocols are notoriously difficult to get right for DSM systems, but our job is made easier by two things. First, Cachet implements the Commit-Reconcile & Fences (CRF) memory model [10], which exposes a semantic notion of caches, and decomposes load and store instructions into finer-grain operations. CRF exposes mechanisms that are needed to specify protocols precisely. It separates how a cache provides correct values from how it maintains coherence. Second, we use a design methodology that separates the correctness and liveness concerns in protocol design [9]. It allows us to incorporate rules for adaptive behaviors without having to worry about correctness and liveness issues.

The necessary background knowledge on CRF is presented in the next section. Section 3 gives an overview of the micro-protocols that comprise Cachet. Section 4 presents details of the micro-protocols, and Section 5 presents a seamless integration of two of the micro-protocols. Finally we present our conclusions in Section 6.

2 The CRF Memory Model

CRF is a mechanism-oriented memory model that exposes both data replication and instruction reordering at the programming level [10]. It decomposes load and store instructions into finer-grain operations that operate on a local semantic cache (sache). The model assumes that memory instructions can be reordered as long as data dependence constraints are preserved, and provides memory fences to enforce ordering if needed. There are five memory-related instructions, Loadl (load-local), Storel (store-local), Commit, Reconcile and Fence. We will not discuss instruction reordering and memory fences any further because these features are orthogonal to the commit-reconcile features, which play the central role in cache coherence protocol design.

Each sache cell has an associated state, which can be either Clean or Dirty. The Clean state indicates that the data has not been modified since it was cached or last written back. The Dirty state indicates that the data has been modified and has not been written back to the memory since then.

Figure 1 presents the rewriting rules that specify the CRF model. It contains one rule for each instruction and three background rules that operate between saches and memory. In this paper, we use state transition tables to describe memory models and cache coherence protocols informally but rigorously. Each row in the table represents a rewriting rule, and has three components, a pre-condition, an action and a post-condition. The pre-condition contains the instruction or message to be processed and the corresponding cache or memory state. It behaves as the predicate that must be satisfied before the rule can be fired. The action is typically completing an instruction or issuing a protocol message. The post-condition is the cache or memory state after the action is performed. A formal description of CRF using Term Rewriting Systems (TRS's) can be found elsewhere [10]. Given proper context, it is straightforward to deduce the precise TRS rules from a tabular description.

A Loadl instruction reads the data from the sache if the address is cached; otherwise it stalls until the value of the address is somehow fetched into the sache. A Storel instruction writes the data into the sache if the address is cached, and sets the sache state to Dirty. Since CRF inherently allows a store operation to be performed without coordinating

Micro-protocol	**Commit on Dirty**	**Reconcile on Clean**	**Cache Miss**
Cachet-Base	update memory	purge local clean copy	retrieve data from memory
Cachet-WriterPush	purge all clean copies update memory		retrieve data from memory
Cachet-Migratory			flush exclusive copy update memory retrieve data from memory

Figure 2. Different Treatment of Commit, Reconcile and Cache Miss

with other saches, different saches can have a cell with the same address but different values.

A Commit instruction on a dirty cell stalls until the data is written back to the memory, while a Reconcile instruction on a clean cell stalls until the data is purged from the sache. The Commit and Reconcile instructions can be used to ensure that the data produced by one processor can be observed by another processor whenever necessary. The memory behaves as the rendezvous between the writer and the reader: the writer performs a commit operation to guarantee that the modified data has been written back to the memory, while the reader performs a reconcile operation to guarantee that the stale copy, if any, has been purged from the sache so that the subsequent load operation must retrieve the data from the memory. Exploitation of the flexibility offered by the commit and reconcile operations underlies all the protocols presented in this paper.

While both Loadl and Storel instructions are local operations, data can be propagated between the memory and a sache under proper conditions. A sache can obtain a clean copy from the memory, if the address is not cached (thus it cannot contain more than one copy for the same address). A dirty copy can be written back to the memory, after which the sache state becomes clean. A clean copy can be purged from the sache at any time, but cannot be written back to the memory. The cache, writeback and purge operations are often referred as background operations since they can be invoked voluntarily even though no instruction is executed by any processor.

Many existing memory models including sequential consistency [7] and release consistency [6] can be described as restricted versions of CRF. For example, in release consistency, a store operation is considered to have completed once the local cache has been modified, while the invalidation of stale copies in other caches can be postponed until the following release point. A more aggressive implementation would allow the invalidation for each cache to be furthermore delayed until the next acquire point. Every aspect of release consistency can be expressed programmatically using the CRF primitives.

Since Cachet implements CRF, it is by definition a protocol for all high-level models whose programs can be translated into CRF programs. The translation can be performed statically by the compiler, or dynamically by the processor or the protocol engine. Indeed, different high-level memory models can be used simultaneously in different regions of memory. For example, in a release consistency program, the region of memory used for input/output operations can have the sequential consistency semantics by simply employing an appropriate translation scheme for that region.

3 Overview of Cachet

The CRF instructions provide great flexibility in designing coherence protocols for DSM systems. For example, a protocol may use any cache in the memory hierarchy as the rendezvous for the processors that access a shared memory location, provided that it maintains the same observable memory behavior. Cachet is a seamless integration of several micro-protocols, which are distinctive in the actions performed by the protocol engine while committing dirty cells and reconciling clean cells. Figure 2 briefly describes the different treatment of commit, reconcile and cache miss in the three micro-protocols.

Cachet-Base: The most straightforward implementation simply uses the memory as the rendezvous. When a Commit instruction is executed for an address that is cached in the Dirty state, the data must be written back to the memory before the instruction can complete. A Reconcile instruction for an address cached in the Clean state requires the data be purged from the cache before the instruction can complete. An attractive characteristic of Cachet-Base is its simplicity; no state needs to be maintained at the memory side.

Cachet-WriterPush: Since load operations are usually more frequent than store operations, it is desirable to allow a Reconcile instruction to complete even when the address is cached in the Clean state. Thus, the following load access to the address causes no cache miss. Correspondingly, when a Commit instruction is performed on a dirty cell, it cannot complete before clean copies of the address are purged from all other caches. Therefore, it can be a lengthy process to commit an address that is cached in the Dirty state.

Cachet-Migratory: When an address is exclusively accessed by one processor for a reasonable time period, it makes sense to give the cache the exclusive ownership so that all instructions on the address become local operations. This is reminiscent of the exclusive state in conventional MESI-like protocols. The protocol ensures that an address can be cached in at most one cache at any time. Therefore, a Commit instruction can complete even when the address is cached in the Dirty state, and a Reconcile instruction can complete even when the address is cached in the Clean state. The exclusive ownership can migrate among different caches whenever necessary.

Different micro-protocols are optimized for different access patterns. Cachet-Base is ideal when the location is randomly accessed by multiple processors, and only necessary commit and reconcile operations are invoked. A conventional implementation of release consistency usually requires that all addresses be indistinguishably committed before a release, and reconciled after an acquire. Such excessive use of commit and reconcile operations can result in performance degradation under Cachet-Base.

Cachet-WriterPush is appropriate when certain processors are likely to read an address many times before another processor writes the address. A reconcile operation performed on a clean copy causes no purge operation, regardless of whether the reconcile is necessary. Thus, subsequent load operations to the address can continually use the cached data without causing any cache miss. Cachet-Migratory fits well when one processor is likely to read and write an address many times before another processor accesses the address.

Adaptivity: Each micro-protocol contains some voluntary rules that are not triggered by any specific instruction or protocol message. A voluntary action can be initiated at either the cache or memory side. For example, at any time, a cache engine can write a dirty copy back to the memory or purge a clean copy from the cache. It can also send a cache request to the memory for an uncached address. The memory engine can voluntarily supply some data to a cache, if the memory contains the most up-to-date data. It can also send a writeback or purge request to a cache to request the data copy of some specific address to be written back or purged. Exact details of voluntary actions may vary for different micro-protocols. The voluntary rules provide enormous scope for intra-protocol adaptivity which can be exploited to achieve better performance.

Different addresses can employ different micro-protocols. It is also possible to dynamically switch the micro-protocol that is operating on an address. In general, the same micro-protocol is used for an address in multiple caches. With appropriate handling, Cachet-Base can co-exist with Cachet-WriterPush or Cachet-Migratory for the same address. This is because Cachet-Base always writes the dirty data back on a commit, and purges the clean copy on a reconcile (thus the subsequent load operation has to retrieve the data from the memory). This gives the memory an opportunity to take proper actions whenever necessary, regardless of how the address is cached in other caches at the time.

The micro-protocols form an access privilege hierarchy. Cachet-Migratory has the most privilege in the sense that both commit and reconcile operations have no impact on the cache, while Cachet-Base has the least privilege in the sense that both commit and reconcile operations may require proper actions to be taken on the cache. Cachet-WriterPush has more privilege than Cachet-Base but less privilege than Cachet-Migratory. A cache can voluntarily downgrade a cache cell to a less privileged protocol, while the memory can voluntarily upgrade a cache cell to a more privileged protocol under proper circumstances. The upgrade operation may need to coordinate with other caches.

Heuristic messages and soft states can be used as hints to invoke desired adaptive actions. A heuristic message is a suggestion that some voluntary action or protocol switch be invoked at a remote site. Soft states can be used later as hints to invoke local voluntary actions, or choose between different micro-protocols.

4 Details of Micro-protocols

A coherence protocol that implements the CRF model in DSM systems must deal with at least two following issues. First, some operations of CRF (i.e., the background rules) involve simultaneous state changes in both memory and cache. In DSM systems where cache and memory communicate via message passing, only those rules whose effect is local are feasible. Second, the system may often need to move into some specific direction in order to avoid deadlock or livelock. For example, on a cache miss, the cache must convey some request information to the memory so that the memory can supply the requested data in time. These problems are solved by employing proper protocol messages, and various cache and memory states.

We assume FIFO message passing throughout the paper. Although the memory appears as one component, in DSM systems, different addresses can be distributed in different sites (homes). By FIFO message passing, we mean that messages between a cache site and a home in the memory are always received in the same order in which they are issued. FIFO only applies to messages with the same address.

We use state transition tables to describe protocols informally. Each table consists of three sets of rules, processor rules, cache engine rules and memory engine rules. Each processor rule deals with one memory instruction on a specific cache state. The action usually involves completing (retiring) the instruction, or issuing certain protocol message in order to process the instruction. A Loadl instruction is retired after the data is supplied to the processor, and a Storel instruction is retired after the cache is modified. The processor rules are all mandatory, thus, if a memory instruction can be executed, it must be executed and retired in finite time. An instruction is stalled if it cannot be retired in the current cache state. This can happen, for example, when the cache has to issue a message to the memory and the instruction cannot be processed before the corresponding reply or acknowledgment is received. The stalled instruction remains unchanged and is retried later.

Note that a stalled instruction only means the instruction itself cannot be processed at the time. It does not necessarily block other memory instructions on the same processor from being processed. Indeed, since instructions can be reordered, another instruction can be processed before the stalled instruction completes. Cache coherence and instruction reordering are, however, completely orthogonal issues.

The cache engine and memory engine rules are further classified as mandatory and voluntary rules. In general, a mandatory cache engine rule deals with a protocol message from the memory, and a mandatory memory engine rule deals with a protocol message from some cache site. If an incoming message can be processed, it must be processed and consumed sooner or later. Once a mandatory rule is applied, the triggering message must be consumed. In contrast, a voluntary rule involves no incoming protocol message, thus the cache or memory state becomes the only predicate that determines whether the rule can be invoked.

Notation: The notation '$\langle cmd,a,v \rangle$' represents a message with command *cmd*, address *a* and value *v* (optional). The source and destination of a message can be either a cache site identifier (*id*) or the memory (Home). A message received at the memory side is always from site *id*, while a message received at a cache is always from Home. The notation '$msg \rightarrow$ Home' means sending the message to the memory, while the notation '$msg \rightarrow dir$' means sending the message to a set of cache sites indicated by the directory *dir*.

4.1 The Cachet-Base Protocol

Figure 3 gives the set of rules for the Cachet-Base protocol. When an instruction cannot be processed, Cachet-Base requires the cache engine to take proper action so that the stalled instruction can eventually complete. On a cache miss, the cache sends a cache-request message to the memory to request the data; the cache state becomes CachePending until the requested data is received. When a Commit instruction is performed on a dirty copy, the cache writes the data back to the memory, but requires a transient state WbPending because an acknowledgment is required from the memory. The Commit cannot be retired until the writeback acknowledgment is received. When a Reconcile instruction is performed on a clean copy, the cache purges the cell to allow the Reconcile to complete.

The memory maintains no directory state for cached copies. It handles the writeback and cache-request messages from caches as follows. When a writeback message is received, the memory updates the memory with the committed data and sends an acknowledgement to the cache. When a cache-request message is received, the memory sends a cache-reply message with the requested data to the requesting site. Note that an incoming message can be serviced instantly.

We mention in passing that Cachet-Base can be further optimized. For example, a Storel instruction on a cache miss can be retired by creating a new cache cell with the stored value. There is no need to first retrieve the data from the memory and then overwrite the data. Note such an optimization is possible because the memory maintains no directory information.

Voluntary rules: A cache can purge a clean cell at any time. It can write the data of a dirty cell back to the memory via a writeback message. A cache can send a cache request to the memory to request the data if the address is not cached.

Voluntary rules can be used to improve the performance without destroying the correctness and liveness properties of the protocol. For example, a cache can evict a cell by purging or by first writing back and then purging if it decides that the data is unlikely to be accessed later. Similarly, a cache can prefetch data by issuing a cache-request message. One subtle point worth noting is that it is not safe for the memory to send data that has not been requested by a cache.

4.2 The Cachet-WriterPush Protocol

In Cachet-Base, a reconcile operation on a clean cell forces the cached copy to be purged before the reconcile can complete. The motivation behind Cachet-WriterPush is to allow a reconcile operation to complete even when the address is cached in the Clean state, so that the subsequent load operation can use the data in the cache without causing a cache miss. A cache cell never needs to be purged unless the address has been updated by another processor or the cache gets full.

The Cachet-WriterPush protocol ensures that a clean cell always contains the same data as the memory. To maintain this invariant, the memory location cannot be updated by a writeback operation before all clean copies have been purged from other caches. This protocol becomes attractive when Reconcile/Loadl operations are more frequent than Storel/Commit operations.

Figure 4 gives the set of rules for Cachet-WriterPush. For each address, the memory maintains a directory state,

Processor Rules

Instruction	Cstate	Action	Next Cstate
Loadl(a)	Cell(a,v,Clean)	retire Loadl	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Loadl	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Storel(a,v)	Cell(a,-,Clean)	retire Storel	Cell(a,v,Dirty)
	Cell(a,-,Dirty)	retire Storel	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Commit(a)	Cell(a,v,Clean)	retire Commit	Cell(a,v,Clean)
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	retire Commit	$a \notin cache$
Reconcile(a)	Cell(a,-,Clean)	retire Reconcile	$a \notin cache$
	Cell(a,v,Dirty)	retire Reconcile	Cell(a,v,Dirty)
	$a \notin cache$	retire Reconcile	$a \notin cache$

Voluntary C-engine Rules

	Cstate	Action	Next Cstate
	Cell(a,-,Clean)		$a \notin cache$
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)

Mandatory C-engine Rules

Message from Home	Cstate	Action	Next Cstate
$\langle$Cache,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean)
$\langle$WbAck,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)

Mandatory M-engine Rules

Message from *id*	Mstate	Action	Next Mstate
$\langle$CacheReq,$a\rangle$	Cell(a,v)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v)
$\langle$Wb,a,$v\rangle$	Cell(a,-)	$\langle$WbAck,$a\rangle \rightarrow id$	Cell(a,v)

Figure 3. The Cachet-Base Protocol

C[*dir*], which contains the identifiers of the sites in which the address is currently cached. The processor rules are similar to those in the Cachet-Base protocol, except that a Reconcile instruction can complete even when the address is cached in the Clean state.

On a cache miss, the cache sends a cache-request message to the memory and sets the cache state to be CachePending until the requested data is received. On a Commit instruction, if the address is cached in the Dirty state, the cache writes the data back to the memory via a writeback message and sets the cache state to be WbPending until the writeback is acknowledged.

The role of the memory is more complicated for writeback operations because the memory must ensure that the copies of the same address in all other caches are consistent. One way to achieve consistency is to purge all the cached copies of the data. Therefore, when a writeback message is received, a purge request is multicast to all caches listed in the directory, except the cache from which the writeback was received. The writeback acknowledgment is withheld until the memory has received acknowledgements for all the purge-request messages. The transient state T[*dir*,*sm*] is introduced for this bookkeeping purpose in the memory. In the transient state, *dir* indicates the sites which have not yet acknowledged the purge request, and *sm* contains the suspended writeback message that the memory has to acknowledge (only the source and the value need to be recorded). Every time a purge acknowledgment is received, *dir* is updated properly. When *dir* becomes empty, the memory services the suspended message by updating the memory and acknowledging the cache with a writeback acknowledgment.

The real picture is slightly more complicated because several sites may initiate writebacks simultaneously. The bookkeeping accommodates this situation by recording all the suspended writeback messages in the transient state T[*dir*,*sm*]. The memory, however, must have the address purged from all sites except one. It accomplishes this by acknowledging a writeback message with either a WbAck or WbAckFlush. If a cache receives a WbAck acknowledgment, it keeps a clean copy; otherwise it purges its copy. After all the purge requests have been acknowledged, the memory responds to each suspended writeback except one by a WbAckFlush message. In fact, it is safe for the memory to purge the address from all sites, that is, send WbAckFlush message to everyone! The rules in Figure 4 allow both of these possibilities (see the second and third rule from the bottom).

It is worth noting that the transient state T[*dir*,*sm*] can save space by just maintaining a counter instead of the direc-

Processor Rules

Instruction	Cstate	Action	Next Cstate
Loadl(a)	Cell(a,v,Clean)	retire Loadl	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Loadl	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Storel(a,v)	Cell(a,-,Clean)	retire Storel	Cell(a,v,Dirty)
	Cell(a,-,Dirty)	retire Storel	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Commit(a)	Cell(a,v,Clean)	retire Commit	Cell(a,v,Clean)
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	retire Commit	$a \notin cache$
Reconcile(a)	Cell(a,v,Clean)	retire Reconcile	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Reconcile	Cell(a,v,Dirty)
	$a \notin cache$	retire Reconcile	$a \notin cache$

Voluntary C-engine Rules

	Cstate	Action	Next Cstate
	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \rightarrow$ Home	$a \notin cache$
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)

Mandatory C-engine Rules

Message from Home	Cstate	Action	Next Cstate
$\langle$Cache,a,$v\rangle$	$a \notin cache$		Cell(a,v,Clean)
	Cell(a,-,CachePending)		Cell(a,v,Clean)
$\langle$WbAck,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)
$\langle$WbAckFlush,$a\rangle$	Cell(a,-,WbPending)		$a \notin cache$
$\langle$PurgeReq,$a\rangle$	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \rightarrow$ Home	$a \notin cache$
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	$a \notin cache$		$a \notin cache$

Voluntary M-engine Rules

	Mstate	Action	Next Mstate
	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,ϵ])

Mandatory M-engine Rules

Message from *id*	Mstate	Action	Next Mstate
$\langle$CacheReq,$a\rangle$	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])
$\langle$Wb,a,$v_1\rangle$	Cell(a,v,C[$id \mid dir$])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,Wb(id,v_1)])
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,Wb(id,v_1) $\mid sm$])
$\langle$Purged,$a\rangle$	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,sm])
	Cell(a,-,T[ϵ,Wb(id,v) $\mid sm$])	$\langle$WbAckFlush,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])
	Cell(a,-,T[ϵ,Wb(id,v)])	$\langle$WbAck,$a\rangle \rightarrow id$	Cell(a,v,C[id])
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])

Figure 4. The Cachet-WriterPush Protocol

tory to remember the number of acknowledgments expected from caches. It can also save space by immediately updating the memory and not saving the value in the suspended message.

A cache responds to a purge request on a clean cell by purging the clean data and sending a Purged message. If the cached copy is dirty, the dirty data is forced to be written back via a writeback message. In data-race-free programs, a cache cannot receive a purge request on a dirty cell unless the request is issued voluntarily from the memory.

A protocol message received at a cache engine can always be processed and consumed immediately. However, when the memory engine receives a cache request while the address is in the transient state, the request message cannot be processed. It is critical that a stalled message not block other protocol messages from being received and processed. A fair scheduling mechanism is needed to ensure that every stalled message is eventually processed.

It is worth noting that Cachet-WriterPush can be optimized further. An instruction is stalled when the address is in some transient state in the cache. This constraint can be relaxed under certain circumstances. For example, a Reconcile instruction can be retired when the address is cached in the CachePending state. The optimization is useful since a cache may voluntarily send a cache request to the memory to prefetch the data. It is desirable that such a voluntary action not block subsequent instructions from being completed.

Voluntary rules: At any time, a cache can purge a clean cell, and notify the memory of the purge operation via a Purged message. It can also write the data of a dirty cell back to the memory via a writeback message. Furthermore, a cache can send a message to the memory to request a data copy for any uncached address, even though no Loadl or Storel instruction is performed by the processor.

The memory can voluntarily send a data copy of any address to any cache, provided the directory shows that the address is not cached in that cache. Thus a cache may receive a data copy even though it has not requested it. Additional rules are needed to handle a data copy that is received while the address is not in the CachePending state. The memory can also voluntarily initiate a purge request to purge clean copies of any address.

We show how to enhance Cachet-WriterPush with the update capability to demonstrate the utility of voluntary rules. This requires introducing some soft states at the memory. When the memory is notified that a cache copy has been purged, it records that site identifier. Later when the memory is updated and all suspended writeback messages are acknowledged, it multicasts the new data to those sites in which the address was just purged. The correctness follows trivially from the fact that the memory can voluntarily send a data copy to a cache in which the directory shows that the address is not cached.

4.3 The Cachet-Migratory Protocol

When a memory location is accessed predominantly by one processor, all the operations performed by the processor should be inexpensive. Figure 5 gives the set of rules for the Cachet-Migratory protocol, which is suitable for this situation. It allows each address to be cached in at most one cache so that both commit and reconcile operations can complete at that site regardless of the cache state of the address. Memory accesses from another site may incur a large expense since the exclusive copy has to be migrated to that site before the accesses can be performed.

The memory maintains which site currently has cached the location. When the memory receives a cache request while the address is cached in another cache, it sends a flush request to the cache to force it to flush its copy. The transient state T[*id*,*sm*] is used to record the site where the address is currently cached, and the suspended cache request (only the source needs to be recorded). The suspended request will be resumed when a Purged or Flushed message is received.

Similar to Cachet-WriterPush, when the memory receives a cache request while the address is in some transient state, the request message must be stalled. A stalled message should not block other incoming messages from being processed.

Voluntary rules: At any time, a cache can purge a clean copy, and notify the memory of the purge operation via a Purged message. It can also flush a dirty copy and write the data back to the memory via a Flushed message. It is worth noting that no acknowledgment is needed from the memory for the flush operation. In addition, a cache can send a request to the memory to request an exclusive copy for an uncached address.

The memory can voluntarily send an exclusive copy to a cache, if the address is currently not cached by any cache. If the memory state indicates that an address is cached in some cache, the memory can voluntarily send a flush request to the cache to force the data to be flushed from the cache.

5 Integration of Micro-protocols

It is easy to see that different addresses can employ different micro-protocols without any interference. The programmer or the compiler can inform both the cache and the memory about which protocol is to be used on each address. One way to implement this idea is to tag every state and message command with the name of the protocol. We use subscripts b, w and m to represent the Cachet-Base, Cachet-WriterPush and Cachet-Migratory protocols, respectively. Thus, the Clean state will be replaced by the Clean_b, Clean_w

Processor Rules

Instruction	**Cstate**	**Action**	**Next Cstate**
Loadl(a)	Cell(a,v,Clean)	retire Loadl	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Loadl	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \longrightarrow$ Home	Cell(a,-,CachePending)
Storel(a,v)	Cell(a,-,Clean)	retire Storel	Cell(a,v,Dirty)
	Cell(a,-,Dirty)	retire Storel	Cell(a,v,Dirty)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \longrightarrow$ Home	Cell(a,-,CachePending)
Commit(a)	Cell(a,v,Clean)	retire Commit	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Commit	Cell(a,v,Dirty)
	$a \notin cache$	retire Commit	$a \notin cache$
Reconcile(a)	Cell(a,v,Clean)	retire Reconcile	Cell(a,v,Clean)
	Cell(a,v,Dirty)	retire Reconcile	Cell(a,v,Dirty)
	$a \notin cache$	retire Reconcile	$a \notin cache$

Voluntary C-engine Rules

	Cstate	**Action**	**Next Cstate**
	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \longrightarrow$ Home	$a \notin cache$
	Cell(a,v,Dirty)	$\langle$Flushed,a,$v\rangle \longrightarrow$ Home	$a \notin cache$
	$a \notin cache$	$\langle$CacheReq,$a\rangle \longrightarrow$ Home	Cell(a,-,CachePending)

Mandatory C-engine Rules

Message from Home	**Cstate**	**Action**	**Next Cstate**
$\langle$Cache,a,$v\rangle$	$a \notin cache$		Cell(a,v,Clean)
	Cell(a,-,CachePending)		Cell(a,v,Clean)
$\langle$FlushReq,$a\rangle$	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \longrightarrow$ Home	$a \notin cache$
	Cell(a,v,Dirty)	$\langle$Flushed,a,$v\rangle \longrightarrow$ Home	$a \notin cache$
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$

Voluntary M-engine Rules

	Mstate	**Action**	**Next Mstate**
	Cell(a,v,C[ϵ])	$\langle$Cache,a,$v\rangle \longrightarrow id$	Cell(a,v,C[id])
	Cell(a,v,C[id])	$\langle$FlushReq,$a\rangle \longrightarrow id$	Cell(a,v,T[id,ϵ])

Mandatory M-engine Rules

Message from id	**Mstate**	**Action**	**Next Mstate**
$\langle$CacheReq,$a\rangle$	Cell(a,v,C[ϵ])	$\langle$Cache,a,$v\rangle \longrightarrow id$	Cell(a,v,C[id])
	Cell(a,v,C[id])		Cell(a,v,C[id])
	Cell(a,v,C[id_1]) ($id_1 \neq id$)	$\langle$FlushReq,$a\rangle \longrightarrow id_1$	Cell(a,v,T[id_1,CacheReq(id)])
$\langle$Flushed,a,$v\rangle$	Cell(a,-,C[id])		Cell(a,v,C[ϵ])
	Cell(a,-,T[id,sm])		Cell(a,v,T[ϵ,sm])
$\langle$Purged,$a\rangle$	Cell(a,v,C[id])		Cell(a,v,C[ϵ])
	Cell(a,v,T[id,sm])		Cell(a,v,T[ϵ,sm])
	Cell(a,v,T[ϵ,CacheReq(id)])	$\langle$Cache,a,$v\rangle \longrightarrow id$	Cell(a,v,C[id])
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])

Figure 5. The Cachet-Migratory Protocol

or Clean_m state, and the Cache message by the Cache_b, Cache_w or Cache_m message, etc.

With slight modification, we can let the memory choose the micro-protocol for each address. A cache request can be generated without a micro-protocol tag, and the memory system can respond with a Cache_b, Cache_w or Cache_m message. The cache can then have the data cached in the Clean_b, Clean_w or Clean_m state depending upon the type of the response received. The memory, however, cannot dynamically switch from one micro-protocol to another on an address without some other significant modifications.

The Cachet-Base micro-protocol can coexist with either Cachet-WriterPush or Cachet-Migratory on the same address, but Cachet-WriterPush and Cachet-Migratory cannot coexist with each other. Cachet-Base gives the memory an opportunity to take appropriate actions whenever necessary, because a dirty copy is always written back on a commit, and a clean copy is always purged on a reconcile so that the subsequent load operation has to retrieve the data from the memory. The lack of space does not allows us to present the complete Cachet protocol. In the rest of this section, we present a protocol that seamlessly integrates the Cachet-Base and Cachet-WriterPush micro-protocols.

Figure 6 gives the rules for the integrated protocol, most of which are taken directly from the two micro-protocols. One critical observation is that the micro-protocols share the cache state Invalid and the memory state $C[\epsilon]$. The common Invalid state indicates that a cache draws no distinction between different micro-protocols for an uncached address. The common $C[\epsilon]$ state means that the address is not cached in Cachet-WriterPush, but may be cached in Cachet-Base. Since the memory maintains no information about which caches have the address cached in Cachet-Base, it should always assume that some caches contain Cachet-Base copies.

Certain critical invariants are maintained to ensure the correctness of the protocol. As in Cachet-Base, the memory cannot send a Cache_b message to a cache unless it receives a cache request from that site. Furthermore, before the memory sends a Cache_b copy, it must make sure that no Cachet-WriterPush copy has been sent to the cache regarding the same address. As in Cachet-WriterPush, the memory cannot be updated when the directory shows that Clean_w copies exist in some caches. Thus, if an address is cached in the Clean_w state, the cached value must be the same as the memory value.

Some unexpected cases must be dealt with properly in the integrated protocol. For example, the memory can receive a Wb_b message while the directory shows that the address is cached in Cachet-WriterPush in some caches. The Wb_b message must be suspended until all Clean_w copies are purged. Similarly, a cache can receive a Cache_w message while a Cachet-Base copy is cached. If the cache state is Clean_b, it can be upgraded to Clean_w with the cache updated. If the cache state is Dirty_b, the cache sends a Purged message to the memory to inform the memory that the Cache_w copy is not accepted.

If a cache has an address cached in the Clean_w state, it can send a Purged message to the memory, and downgrade the Clean_w state to Clean_b. This can happen either voluntarily when the cache intends to downgrade a cell from Cachet-WriterPush to Cachet-Base, or mandatorily when it receives a purge request from the memory. The actual purge of the data can be delayed until the next reconcile point. This lazy-purge technique can be useful in reducing potential cache thrashing due to false sharing.

Note that when the memory receives a cache request while the address is in some transient state with suspended writeback messages, the memory can service the cache request by supplying a Cachet-Base copy to the requesting site. Thus, unlike Cachet-WriterPush, no message needs to be buffered (or retried) at the memory side. It is also worth noting that the WbAckFlush_w message has been merged with the WbAck_b message for they carry the same information.

As an example of how the adaptivity can be exploited, consider a DSM system with limited directory space. When the memory receives a cache request, it can respond under either Cachet-Base or Cachet-WriterPush. One reasonable strategy is to always supply a Cachet-WriterPush copy except when the directory is full, in which case it supplies a Cachet-Base copy. Moreover, the memory can send a purge request to a cache to downgrade a cache cell from Cachet-WriterPush to Cachet-Base so that the resumed directory space can be used for other Cachet-WriterPush copies. This simple adaptivity will allow an address to be resident in more caches than the number of cache identifier slots in the directory.

6 Conclusion

Cachet is one cache coherence protocol, although for pedagogic reasons it has been presented as an integration of several micro-protocols. It is also possible to treat Cachet as a family of protocols because of the presence of voluntary rules that can be invoked without an instruction or protocol message. Cachet's voluntary rules provide enormous extensibility in the sense that various heuristic messages and soft states can be employed to invoke these rules. One can consider each heuristic and associated soft state as giving rise to a new protocol.

One way to think of Cachet is that its rules define a toolkit of coherence primitives that can be used to build coherence protocols on-the-fly. When an instruction or protocol message is received, the protocol engine can execute any of the legal coherence actions, without the fear of destroying the correctness and liveness. This is completely different from

Processor Rules

Instruction	**Cstate**	**Action**	**Next Cstate**
Loadl(a)	Cell(a,v,Clean$_b$)	retire Loadl	Cell(a,v,Clean$_b$)
	Cell(a,v,Dirty$_b$)	retire Loadl	Cell(a,v,Dirty$_b$)
	Cell(a,v,Clean$_w$)	retire Loadl	Cell(a,v,Clean$_w$)
	Cell(a,v,Dirty$_w$)	retire Loadl	Cell(a,v,Dirty$_w$)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Storel(a,v)	Cell(a,-,Clean$_b$)	retire Storel	Cell(a,v,Dirty$_b$)
	Cell(a,-,Dirty$_b$)	retire Storel	Cell(a,v,Dirty$_b$)
	Cell(a,-,Clean$_w$)	retire Storel	Cell(a,v,Dirty$_w$)
	Cell(a,-,Dirty$_w$)	retire Storel	Cell(a,v,Dirty$_w$)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)
Commit(a)	Cell(a,v,Clean$_b$)	retire Commit	Cell(a,v,Clean$_b$)
	Cell(a,v,Dirty$_b$)	$\langle$Wb$_b$,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	Cell(a,v,Clean$_w$)	retire Commit	Cell(a,v,Clean$_w$)
	Cell(a,v,Dirty$_w$)	$\langle$Wb$_w$,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	retire Commit	$a \notin cache$
Reconcile(a)	Cell(a,-,Clean$_b$)	retire Reconcile	$a \notin cache$
	Cell(a,v,Dirty$_b$)	retire Reconcile	Cell(a,v,Dirty$_b$)
	Cell(a,v,Clean$_w$)	retire Reconcile	Cell(a,v,Clean$_w$)
	Cell(a,v,Dirty$_w$)	retire Reconcile	Cell(a,v,Dirty$_w$)
	$a \notin cache$	retire Reconcile	$a \notin cache$

Voluntary C-engine Rules

	Cstate	**Action**	**Next Cstate**
	Cell(a,-,Clean$_b$)		$a \notin cache$
	Cell(a,v,Dirty$_b$)	$\langle$Wb$_b$,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	Cell(a,v,Clean$_w$)	$\langle$Purged,$a\rangle \rightarrow$ Home	Cell(a,v,Clean$_b$)
	Cell(a,v,Dirty$_w$)	$\langle$Wb$_w$,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ Home	Cell(a,-,CachePending)

Mandatory C-engine Rules

Message from Home	**Cstate**	**Action**	**Next Cstate**
$\langle$Cache$_b$,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean$_b$)
$\langle$Cache$_w$,a,$v_1\rangle$	Cell(a,-,Clean$_b$)		Cell(a,v_1,Clean$_w$)
	Cell(a,v,Dirty$_b$)	$\langle$Purged,$a\rangle \rightarrow$ Home	Cell(a,v,Dirty$_b$)
	Cell(a,-,CachePending)		Cell(a,v_1,Clean$_w$)
	Cell(a,v,WbPending)	$\langle$Purged,$a\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	$a \notin cache$		Cell(a,v_1,Clean$_w$)
$\langle$WbAck$_b$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_b$)
$\langle$WbAck$_w$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_w$)
$\langle$PurgeReq,$a\rangle$	Cell(a,v,Clean$_b$)		Cell(a,v,Clean$_b$)
	Cell(a,v,Dirty$_b$)		Cell(a,v,Dirty$_b$)
	Cell(a,v,Clean$_w$)	$\langle$Purged,$a\rangle \rightarrow$ Home	Cell(a,v,Clean$_b$)
	Cell(a,v,Dirty$_w$)	$\langle$Wb$_w$,a,$v\rangle \rightarrow$ Home	Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	$a \notin cache$		$a \notin cache$

Voluntary M-engine Rules

	Mstate	**Action**	**Next Mstate**
	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache$_w$,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,ϵ])

Mandatory M-engine Rules

Message from *id*	**Mstate**	**Action**	**Next Mstate**
$\langle$CacheReq,$a\rangle$	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache$_b$,a,$v\rangle \rightarrow id$	Cell(a,v,C[dir])
		$\langle$Cache$_w$,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])
	Cell(a,v,T[dir,sm]) ($id \notin dir$)	$\langle$Cache$_b$,a,$v\rangle \rightarrow id$	Cell(a,v,T[dir,sm])
	Cell(a,v,T[dir,sm]) ($id \in dir$)		Cell(a,v,T[dir,sm])
$\langle$Wb$_b$,a,$v_1\rangle$	Cell(a,v,C[dir])	$\langle$PurgeReq,$a\rangle \rightarrow dir$-id	Cell(a,v,T[dir,Wb$_b$(id,v_1)])
	Cell(a,v,T[dir,sm])		Cell(a,v,T[dir,Wb$_b$(id,v_1)$\mid sm$])
$\langle$Wb$_w$,a,$v_1\rangle$	Cell(a,v,C[$id \mid dir$])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,Wb$_w$(id,v_1)])
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,Wb$_w$(id,v_1)$\mid sm$])
$\langle$Purged,$a\rangle$	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,sm])
	Cell(a,-,T[ϵ,Wb$_b$(id,v)$\mid sm$])	$\langle$WbAck$_b$,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])
	Cell(a,-,T[ϵ,Wb$_w$(id,v)$\mid sm$])	$\langle$WbAck$_b$,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])
	Cell(a,-,T[ϵ,Wb$_w$(id,v)])	$\langle$WbAck$_w$,$a\rangle \rightarrow id$	Cell(a,v,C[id])
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])

Figure 6. Integration of Cachet-Base and Cachet-WriterPush

the Teapot method [2] where the burden of ensuring the correctness and liveness of the system falls on the programmer.

In Cachet, a store operation can be performed without the exclusive ownership, which effectively allows multiple writers for the same address simultaneously. This can reduce the average latency for write operations and alleviate potential cache thrashing due to false sharing. Moreover, the purge of an invalidated cache cell can be deferred to the next reconcile point, which can help reduce cache thrashing due to read-write false sharing.

Cache states in processors are usually maintained at the cache-line level which typically contains 8 to 64 cells. Large cache lines are known to have the false-sharing problem, which can severely degrade performance. An ideal implementation of Cachet keeps a dirty bit for each cell even though all the cells of a cache line are swapped in and out together. In the absence of such support, we have developed a data merge mechanism that allows modifications of the same cache line from different processors to be properly combined at the memory. Since a write can be performed without the exclusive ownership, there can be multiple writers for the same cache line at the same time.

Implementation of adaptivity requires mechanisms to discover the access patterns and conveying the heuristic information to the protocol engines. There are several possible solutions for this problem. Access patterns can be given by the programmer as program annotations, or detected through compiler analysis or runtime statistic collection. When access patterns are statically recognized by the programmer or the compiler, the information can be conveyed to the underlying protocol engine in a straightforward manner. Another possibility is to expose a Cachet interface to the compiler so that appropriate coherence actions can be invoked directly by the program. The dynamic detection of access patterns is likely to require hardware support specific to a given implementation. A detailed discussion of this issue is beyond the scope of this paper.

The maximum performance advantage of Cachet would be realized when microprocessors directly support CRF and compilers translate programs written under programmer-centric high-level memory models into CRF programs. Cachet is an ideal protocol for building scalable DSM systems even using commercial microprocessors and SMP's, since it can be incorporated at L3 or higher-level of cache hierarchy in a transparent manner.

Acknowledgment: Xiaowei Shen acknowledges the valuable summer experience in Charles Schulz's group at IBM T. J. Watson. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft Huachuca contract DABT63-95-C-0150.

References

[1] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[2] S. Chandra, B. Richard, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[3] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[4] S. Eggers and R. H. Katz. Evaluating the Performance for Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[5] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, Nov. 1994.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[7] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[8] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *International Symposium on Computer Architecture*, 1998.

[9] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, MIT, June 1997.

[10] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta*, May 1999.

[11] P. Stenstrom, B. brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[12] W. D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.