Tech. Rep. 723, Dec. '99

The Implementation of Cashmere[1]

Robert J. Stets, DeQing Chen, Sandhya Dwarkadas, Nikolaos Hardavellas,
Galen C. Hunt, Leonidas Kontothanassis, Grigorios Magklis
Srinivasan Parthasarathy, Umit Rencuzogullari, Michael L. Scott

www.cs.rochester.edu/research/cashmere
cashmere@cs.rochester.edu

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

[1]This work was supported in part by NSF grants CDA–9401142, CCR–9702466, and CCR–9705594; and an external research grant from Compaq.

Abstract

Cashmere is a software distributed shared memory (SDSM) system designed for today's high performance cluster architectures. These clusters typically consist of symmetric multiprocessors (SMPs) connected by a low-latency system area network. Cashmere introduces several novel techniques for delegating intra-node sharing to the hardware coherence mechanism available within the SMPs, and also for leveraging advanced network features such as remote memory access. The efficacy of the Cashmere design has been borne out through head-to-head comparisons with other well-known, mature SDSMs and with Cashmere variants that do not take advantage of the various hardware features.

In this paper, we describe the implementation of the Cashmere SDSM. Our discussion is organized around the core components that comprise Cashmere. We discuss both component interactions and low-level implementation details. We hope this paper provides researchers with the background needed to modify and extend the Cashmere system.

Contents

1 Introduction

Cashmere is a software distributed shared memory (SDSM) system designed for a cluster of symmetric multiprocessors (SMPs) connected by a low-latency system area network. In this paper, we describe the architecture and implementation of the Cashmere prototype. Figure 1 shows the conceptual organization of the system. The root *Cashmere* component is responsible for program startup and exit. The *Protocol*[1] component is a policy module that implements the Cashmere shared memory coherence protocol. The *Synchronization*, *Messaging*, and *Services* components provide the mechanisms to support the Protocol component. The *Miscellaneous* component contains general utility routines that support all components.

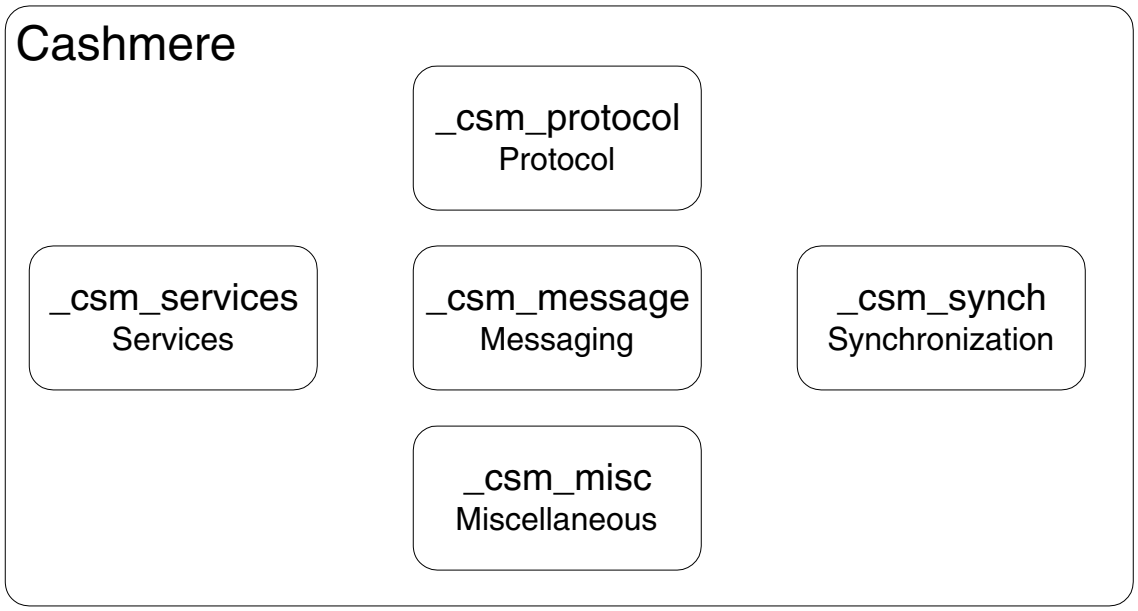

Figure 1: Cashmere organization.

The Cashmere prototype has been implemented on a cluster of Compaq AlphaServer SMPs connected by a Compaq Memory Channel network [3, 4]. The prototype is designed to leverage the hardware coherence available within SMPs and also the special network features that may lower communication overhead. In this paper, we will discuss the implementation of two Cashmere versions that are designed to isolate the impact of the special network features. The "Memory Channel" version of Cashmere leverages all of the special network features, while the "Explicit Messages" version relies only on low-latency messaging. As shall become clear, the difference between the two Cashmere versions is largely confined to well-defined operations that access the network.

In the next section we will provide a general overview of the Cashmere protocol operation. The following sections will then describe the Protocol, Services, Messaging, Synchronization, and Miscellaneous components in more detail. Additionally, Section 8 discusses a few of the software engineering

[1] In this paper, we will refer interchangeably to components by their prose name and their implementation class name.

measures followed during development, and Section 9 discusses several Cashmere debugging techniques.

This doument also contains two appendices. Appendix A discusses known bugs in the system, and Appendix B details the Cache Coherence Protocol (CCP) application build environment. This build environment provides a high-level macro language that can be mapped to a number of shared memory systems.

2 Background

Before studying the details of this paper, the reader is strongly encouraged to read two earlier Cashmere papers [8, 7]. These papers provide good overviews of the Cashmere protocol and provide details of its implementation. A good understanding of the Cashmere protocol and also of the Memory Channel network [3, 4] are essential in understanding the implementation details. In the rest of this Section, we will briefly introduce terms and concepts that the reader must understand before proceeding.

Memory Channel Cashmere attempts to leverage the special network features of the Memory Channel network. The Memory Channel employs a memory-mapped, programmed I/O interface, and provides remote write capability, inexpensive broadcast, and totally ordered message delivery. The remote write capability allows a processor to modify memory on a remote node, without requiring assistance from a processor on that node. The inexpensive broadcast mechanism combines with the remote write capability to provide very efficient message propagation. Total ordering guarantees that all nodes will observe broadcast messages in the same order: the order that the messages reach the network.

Cashmere The Cashmere SDSM supports a "moderately lazy" version of release consistency. This consistency model requires processors to synchronize in order to see each other's data modifications. From an implementation point of view, it allows Cashmere to postpone most data propagation until synchronization points. Cashmere provides several synchronization operations, all of which are built from *Acquire* and *Release* primitives. The former signals the intention to access a set of shared memory locations, while the latter signals that the accesses are complete. Cashmere requires that an application program contain "enough" synchronization to eliminate all data races. This synchronization must be visible to Cashmere.

To manage data sharing, Cashmere splits shared memory into page-sized coherence blocks, and uses the virtual memory subsystem to detect accesses to these blocks. Each page of shared memory has a single distinguished *home node* and an entry in a global (replicated) *page directory*. The home node maintains a master copy of the page, while the directory entry maintains information about the page's sharing state and home node location. Each page may exist in either Invalid, Read, or Read-Write state on a particular node.

An access to an Invalid page will result in a page fault that is vectored to the Cashmere library. Cashmere will obtain an up-to-date copy of the page from the home node via a *page update*[2] request.

[2] In this paper and the Cashmere code itself, a page update is also referred to as a "page fetch" operation.

If the fault was due to a read access, Cashmere will upgrade the page's sharing state for the node and its virtual memory permissions to Read, and then return control to the application.

In the event of fault due to a write access, Cashmere may move the page into *Exclusive*[3] mode if the node is the only sharer of the page. Exclusive pages are ignored by the protocol until another sharer for the page emerges. If there is more that one node sharing the page, Cashmere will make a pristine copy, called a *twin*, of the page and place the page ID into the processor's *dirty list*. These two steps will allow modifications to the page to be recovered later. After these steps, the Cashmere handler will upgrade sharing state for the node and VM permissions to Read-Write and return control to the application.

During a Release operation, Cashmere will traverse the processor's dirty list and compare the working copy of each modified page to its twin. This operation will uncover the page's modified data, which is collectively called a *diff*. (In the rest of this paper, we refer to the entire operation as a "diff".) The diff is then sent to the home node for incorporation in the master copy. After the diff, Cashmere will send *write notices* to all sharers of the page. At each Acquire operation, processors will invalidate all pages named by the accumulated write notices.

The twin/diff operations are not necessary on the home node, because processors on the home work directly on the master copy of the page. To reduce twin/diff overhead, some Cashmere variants (and in particular both versions described in this report) *migrate* the page's home node to active writers.

In the next Section, we will provide a more detailed explanation of the Cashmere Protocol component. The following sections will then discuss the components that support the Protocol.

3 Protocol Component

The organization of the Protocol component is pictured in Figure 2. The component consists of a set of support routines and then four aggregated classes. The `_prot_acquire` and `_prot_release` classes perform the appropriate protocol operations for the Acquire and Release synchronization primitives. The `_prot_fault` class implements Cashmere page fault handling. The `_prot_handlers` class provides a set of handlers for protocol-related messages.

To manage its actions, the Protocol component maintains an area of metadata on each node. Some of this metadata is private to each processor, but most of the metadata is shared by processors within the node. The `_csm_node_meta_t` type in `prot_internal.h` describes a structure containing:

- the node's twins
- node-level write notices
- node-level page directory
- node-level dirty list (also referred to as the *No-Longer-Exclusive list*)
- per-page timestamps indicating the last Update and Flush (Diff) operations and the last time a Write Notice was received

[3] The Cashmere code also refers to this as Sole-Write-Sharer mode.

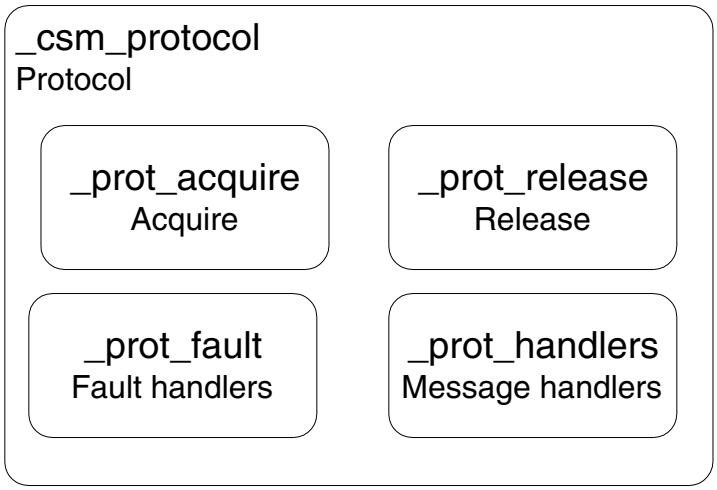

Figure 2: Protocol component structure.

- timestamp of the last Release operation on the node
- a *Stale* vector kept for each page homed on this node, indicating which remote nodes have stale copies of the page

The node-level dirty list contains pages that have left Exclusive mode. In addition to the node-level dirty list, each processor also maintains a private dirty list containing pages that have been modified. The two lists are maintained separately in order to allow the common case, modifications to the per-processor dirty list, to proceed without synchronization.

The timestamps are based on a logical clock local to the node. These timestamps are used to determine when two protocol operations can safely be coalesced. The Stale vector is used by the home migration mechanism to ensure that the new home node is up-to-date after migration.

In the following sections, we will discuss the main protocol entry points and the corresponding implementation of the Cashmere Protocol component.

3.1 Page Faults

All page faults are vectored into the Cashmere `_csm_segv_handler` handler. This handler first verifies that the data address belongs to the Cashmere shared data range. Control is then passed to either the `_prot_fault::ReadPageFault` or `_prot_fault::WritePageFault` handler, depending on the type of the faulting access.

Both of these handlers begin by acquiring the local node lock for the affected page. This node lock is managed by the Page Directory service and is described in Section 4.1. The lock acquire blocks other local processors from performing protocol operations on the page and is necessary to serialize page update and diff operations on the page. Both handlers then update the sharing set entry and transfer control to the `_prot_fault::FetchPage` routine.

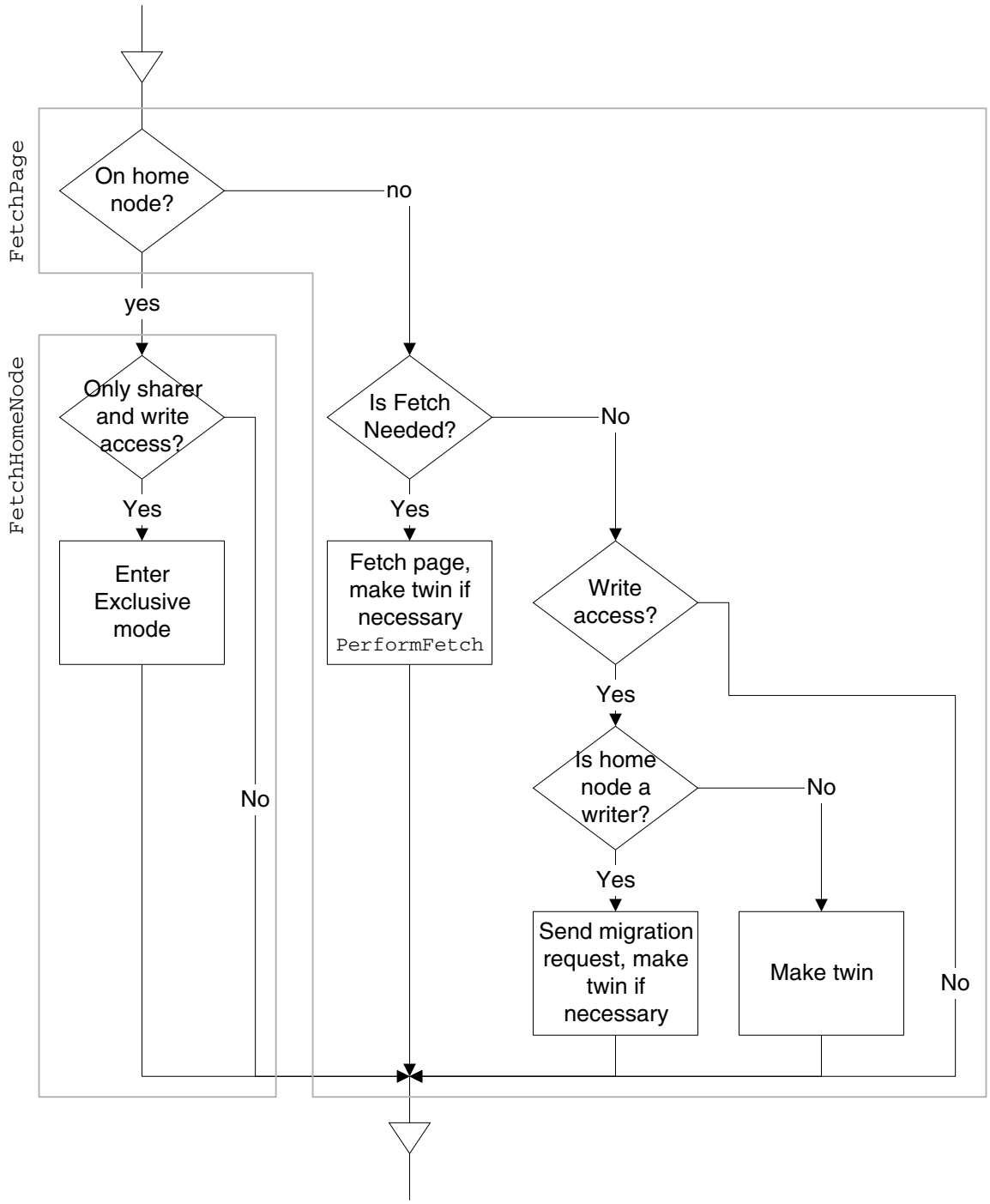

Figure 3: Flow chart of `_prot_fault::FetchPage()` and `_prot_fault::FetchHomeNode()`.

Figure 3 shows the control flow through `FetchPage`. If this node is the home for the page then the fetch process is relatively simple. In `_prot_fault::FetchHomeNode`, the protocol simply checks the current sharing status and enters Exclusive mode if this node is both writing the page and the only sharer. The page will leave Exclusive mode when another node enters the sharing set and sends a page update request to the home node. (As part of the transition out of Exclusive mode, the page will be added to the node-level dirty list of all writers at the home node. This addition will ensure that the writers eventually expose their modifications to the cluster.)

If this node is not the home, then the fetch process is more complicated. The protocol must ensure that the node has an up-to-date copy of the page before returning control to the application. It first checks to see if a fetch is needed. An up-to-date copy of the page may already have been obtained by another processor within the node. The protocol simply compares the last Update and last Write Notice timestamps. If the last Write Notice timestamp dominates then a fetch is required and control transfers to the `_prot_fault::PerformFetch` routine. This routine will send a page update request to the home node and copy the page update embedded in the reply into the working copy of the page.

The `PerformFetch` routine must apply care when updating the working copy of the page. If there are concurrent local writers to the page, the routine cannot blindly copy the page update to the working copy. This blind copy operation could overwrite concurrent modifications. Instead, the protocol compares the new page data to the twin, which isolates the modifications made by other nodes. These modifications can then by copied into the working copy of the page without risk of overwriting local modifications. This routine will also create a twin if the fault is due to a write access.

Returning to Figure 3, if a page fetch is not necessary, then the protocol will check to see if the fault is due to a write access. If so, the protocol will either send a migration request or simply make a twin. The migration request is sent if the home node is not currently writing the page. The home node's sharing status is visible in the Page Directory. (In the Explicit Messages version of Cashmere, the home node sharing status is visible on remote nodes, however the the status may be stale and can only be considered as hint.)

A migration request is sent directly to the home node. A successful request results in a small window of time during which the home node is logically migrating between the nodes. During this window, page update requests cannot be serviced; the update requests must instead be buffered. (As described in Section 5, the Message component provides a utility for buffering messages.) Before sending a migration request, Cashmere will set a flag indicating that a migration request is pending by calling `_csm_protocol::SetPendingMigr`. The `_prot_handlers::PageReqHandler` handler will buffer all page update requests that arrive while a migration is pending for the requested page.

The home node grants a migration request if it is not actively writing the page. The `_prot_handlers::MigrReqHandler` handler is triggered by the migration request. If the request is to be granted, the handler will first set the new home PID indication on its node. (This step begins the logical migration, opening the window of vulnerability described above.) Then, the handler will formulate an acknowledgement message with a success indication and also possibly the latest copy of the page. The latest copy of the page is necessary if the `Stale` vector indicates the new home node does not have an up-to-date page copy.

Upon receipt of the acknowledgement message in the `MigrReplyHandler`, the requestor will update its node home PID indication, if the migration was successful, and also copy any page update to the

working copy. If the migration request was not successful, the requestor will create a twin.

After completion of `FetchPage`, control will return to the appropriate page fault routine (`ReadPageFault` or `WritePageFault`). The `ReadPageFault` routine will release the node lock, change the VM permissions to Read-only and then return control to the application.

The completion of `WritePageFault` requires a few more steps. First, the processor will release the associated node lock. Then, the processor will flush the message store in order to process any messages that may have been buffered during the fault handler's migration attempt. Then the processor will add the page ID to its local dirty list. Finally, the processor can upgrade VM permissions to Read/Write and return control to the application.

3.2 Protocol Acquires

Protocol acquires are relatively simple operations. During an acquire, Cashmere will invalidate all pages listed by the accumulated write notices. Before the invalidation can occur however, the write notice structure must be specially processed.

As will be described in Section 4.3, write notices are stored in a two-level structure consisting of a global, per-node level and a local, per-processor level. To begin an Acquire, Cashmere will first distribute write notices from the global write notice list to the local write notice lists. (This step is not necessary in the explicit messages version of Cashmere where distribution occurs when the message is received.) Then, the processor will invalidate all pages listed in its local write notice list.

The invalidation process is straightforward except in the case where the target page is dirty (*i.e.* modified) locally. In this case, the protocol will flush the modifications to the home node before invalidating the page.

3.3 Protocol Releases

In response to a Release operation, Cashmere will expose all local modifications to the entire cluster. Beginning in `_prot_release::Release`, Cashmere will first distribute any pages in the node-level dirty list to the local dirty list. Then, the protocol can traverse the local dirty list and expose all modifications.

At the home node, all processors work directly on the master copy. Therefore, all modifications are already in the home node copy, so a processor must only send out write notices to expose the changes. The write notices are sent to all nodes sharing the modified page.

At a remote node, the modifications must be sent to the home for incorporation into the master copy. The modifications are sent to the home through a diff operation.[4] If there are local concurrent writers, the protocol will also update the twin to reflect the current state of the working page. This update will eliminate the possibility of sending the same modifications twice. If there are no local concurrent writers, the protocol will release the twin, freeing it for use in some subsequent write fault on another page (see Section 4.2).

[4]The modifications can be pushed to the network either by using the Memory Channel's programmed I/O interface or by copying them into a contiguous local buffer, which is then streamed to the network. The former is faster, while the latter (poorly) emulates a DMA-based interface. This choice is controlled by the `_CSM_DIFF_PIO` macro in `csm_internal.h`.

Cashmere leverages the hardware shared memory inside the SMPs to reduce the number of diff operations. First, if a page has multiple writers within the node and the Release operation is part of a barrier, the diff will be performed only by the last writer to enter the barrier. Second, before performing a diff, Cashmere always compares the last Release and the page's last Flush timestamps. If the Flush timestamp dominates, another node is either currently flushing or has already flushed the necessary modifications. In this case, Cashmere must wait for the other processor to complete the flush and then it can send the necessary write notices. Cashmere can determine when the flush operations are complete by checking the *diff* bit in the page's Directory entry (see Section 4.1).

After the diff operation is complete, the processor can send out the necessary write notices and downgrade the page sharing and VM permissions to Read. This downgrade will allow future write accesses to be trapped.

Ordering between Diffs and Write Notices Cashmere requires that the diff operation is complete before any write notices can be sent and processed. The Memory Channel version of Cashmere relies on the Memory Channel's total ordering guarantee. Cashmere can establish ordering between the protocol events simply by sending the diff before sending the write notices.

The Explicit Message version of Cashmere does not leverage the network total ordering, and so diffs must be acknowledged before the write notices can be sent. To reduce the acknowledgement latency, Cashmere pipelines diff operations to different processors. The code in `Release` basically cycles through the dirty list and sends out the diffs. If the diff cannot be sent out because the appropriate message channel is busy, the page is placed back in the dirty list to be re-tried later.

The code keeps track of the number of outstanding diff messages and stalls the Release operation until all diff operations are complete.

4 Services Component

The Services component provides many of the basic mechanisms to support the Protocol component. Figure 4 shows the structure of the Services component. In this Section, we will discuss the core classes that compose the Services component. The Page Directory component implements the global page directory. The Twins component provides a mechanism that manages and performs Twin operations. The Write Notices component enables processors to send write notices throughout the system, and the Memory Allocation component provides an implementation of `malloc`. In the following sections, we discuss the Page Directory, Twins, Write Notices and Memory Allocation components.

4.1 Page Directory

The Page Directory maintains sharing information for each page of shared memory. The Directory has one entry per page, and is logically split into two levels. The *global* level maintains sharing information on a per-node basis, while the *node* level is private to each node and maintains sharing information for the local processors. The global level is implemented to allow concurrent write accesses without requiring locking, while the node level serializes accesses via fast hardware shared memory locks.

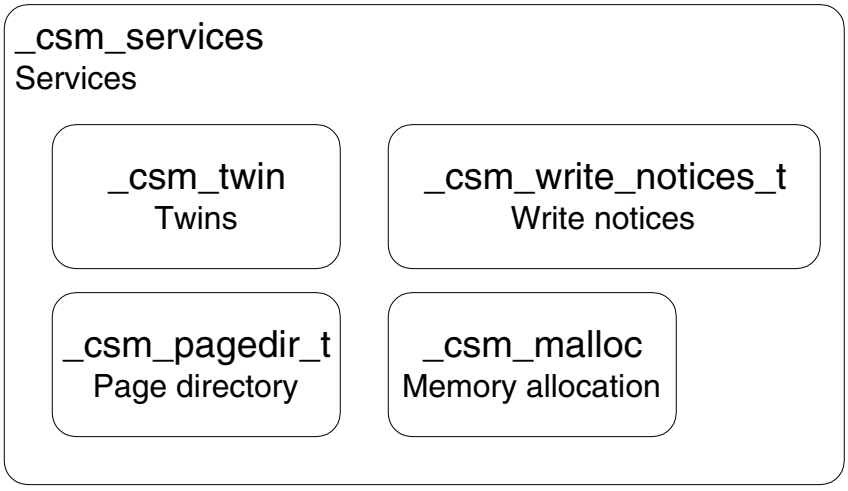

Figure 4: Services component structure.

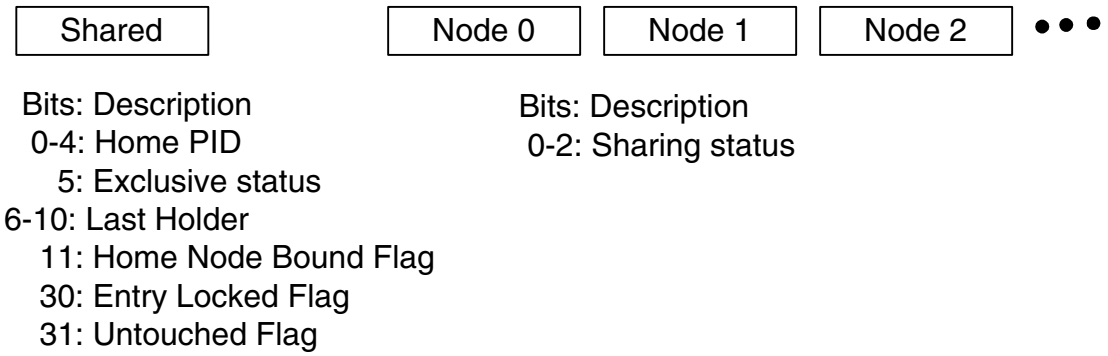

Figure 5: Logical structure of the global directory entry.

Global Page Directory Entry The logical structure of a global page directory entry is pictured in Figure 5. The entry consists of a Shared word and then a set of per-node words. As described below, the separation between the Shared word and the per-node words is necessary to support home node migration. The Shared word contains the following information:

Home PID Processor ID of the home.

Exclusive Flag Boolean flag indicating if page is in Exclusive mode.

Last Holder The Processor ID that last acquired the page. This item is provided for Carnival [6] (performance modeling) support.

Home Node Bound Flag Boolean flag indicating if home node is bound and cannot migrate.

Entry Locked Flag Boolean flag indicating if entry is locked.

Untouched Flag Boolean flag indicating if page is still untouched.

The per-node words only contain the node's sharing status (Invalid, Read, or Read/Write) and a bit indicating if a Diff operation is in progress.

The global directory entry is structured to allow concurrent write accesses. First, the Shared word is only updated by the home node, and the other words are only updated by their owners. The entry can be updated concurrently by this strategy, however, a processor must read each of the per-node words in order to determine the global sharing set. We have evaluated this strategy against an alternative that uses cluster-wide Memory Channel-based locks to serialize directory modifications, and found that our "lock-free" approach provides a performance improvement of up to 8% [8].

The Shared word provides a single location for storing the home PID. A single location is necessary to provide a consistent directory view in the face of home node migration. It also provides a natural location for additional information, such as the Exclusive flag.

Node Page Directory Entry The node level of the page directory entry maintains sharing information for processors within the node. The logical structure is very simple: the entry contains the sharing status (Invalid, Read, Read/Write) for each of local processors. Instead of picturing the logical structure of this entry, it is more informative to examine the implementation. Figure 6 shows the type definition. The `state` field is split into bit fields that indicate the sharing state for each local processor. The other fields are used during protocol operation:

`pValidTwin` A pointer to the twin, if twin is attached.

`bmPendingMigr` A bitmask indicating which local processors have a migration operation pending.

`NodeLock` A local node lock used to provide coarse-grain serialization of protocol activities.

`DirLock` A local node lock used only in low-level routines that modify the directory, thus allowing additional concurrency over the `NodeLock`.

`Padding` A measure[5] to eliminate false sharing.

[5] The padding fields are unused by the protocol, however they can be used to store information during debugging. Several access functions are included in the Cashmere code.

```
// Note: Paddings are calculated with other members of
// _csm_pagedir_entry_t considered.
typedef struct {

  // First cache line (see _csm_pagedir_entry_t)
    csm_64bit_t  state;         // Page state

    csm_64bit_t  *pValidTwin;
        // If twin is currently valid, points to twin else NULL

    csm_32bit_t  bmPendingMigr;
        // per-cid bitmask stating whether the processor
        // is waiting on a pending migration. This could
        // be integrated into the state field.
    csm_32bit_t  padding2[5];

  // Second cache line
    csm_lock_t   NodeLock;      // Locks page inside a node
    csm_lock_t   DirLock;

    csm_64bit_t  padding3[5];
} _csm_node_pagedir_entry_t;
```

Figure 6: Type implementation of the Node Page Directory entry. The csm_32bit_t and csm_64bit_t types represent 32-bit and 64-bit integers, respectively. The csm_lock_t is a 64-bit word.

Directory Modifications The Page Directory is global and replicated on each node. The method used to propagate modifications depends on the specific Cashmere version. The Memory Channel version of Cashmere uses the Memory Channel's inexpensive broadcast mechanism to broadcast changes as they occur. As described above, the structure of the directory entry allows concurrent write accesses.

The Explicit Messages version of Cashmere, by necessity, avoids any broadcasts. Instead, a master copy of each entry is maintained at the entry's home node. Remote nodes simply cache copies of the directory entry. This Cashmere version must ensure that the master directory entry on the home node is always kept up-to-date. Also, Cashmere must ensure that up-to-date information is either passed to remote nodes when needed or that the actions of a remote node can tolerate a stale directory entry.

The three key pieces of information in the directory are the sharing set, the home PID indication, and the Exclusive flag. In the following paragraphs, we examine how the Explicit Messages version of Cashmere propagates changes to these pieces of information.

The global sharing set needs to be updated when a new node enters the sharing set via a Read or Write fault or exits the sharing set via an Invalidation operation. In the former case, the node will also require a page update from the home node. Cashmere simply piggybacks the sharing set update onto the page update request, allowing the home node to maintain the master entry. In the latter case however, there is no existing communication with the home node to leverage. Instead, as part of every Invalidation operation, Cashmere sends an explicit directory update message to the home node. These steps allow both the master copy at the home node and the copy at the affected node to be properly maintained.

Unfortunately, with this strategy, other nodes in the system may not be aware of the changes to the sharing set. In the Cashmere protocol, a remote node only needs the sharing set during one operation: the issuing of write notices. Fortuitously, this operation follows a diff operation that communicates with the home node. The home node can simply piggyback the current sharing state on the diff acknowledgement, allowing the processor to use the up-to-date sharing set information to issue write notices.

The home PID indication also needs to be updated as the home node is initially assigned (or subsequently migrated). We use a lazy technqiue to update this global setting. A processor with a stale home PID indication will eventually send a request to the incorrect node. This node will forward the request to the home node indicated in its cached directory entry. This request will be repeatedly forwarded until it reaches the home node. The home node always piggybacks the latest sharing set and home PID indication onto its message acknowledgements. The message initiator can then use this information to update its cached entry.

The `_csm_packed_entry_t` is provided to pack a full directory entry (home PID, sharing set, bound flag) into a 64-bit value, and this 64-bit value can be attached to the acknowledgements. It is beneficial to pack the entry because Memory Channel packets on our platform are limited to 32 bytes.[6] An unpacked directory entry will result in an increased number of packets and higher operation latency.

Management of the Exclusive flag is simplified by the Cashmere design, which only allows the home node to enter Exclusive mode. This design decision was predicated on the base Cashmere version, which allows home node migration. A page in Exclusive mode has only one sharer and that sharer has Read/Write access to the page. By definition then, the home node will have migrated to the single

[6] The packet size is limited by our Alpha 21164 microprocessors, which cannot send larger writes to the PCI bus. [3]

writer, and so Exclusive mode can only exist on the home node. When the home node is entering Exclusive mode, it can simply update the master entry. The remote nodes do not need to be aware of the transition; if they enter the sharing set, they will send a page update request to the home node, at which point the home can leave Exclusive mode.

The remaining pieces of information in the directory entry are the Last Holder value, the Home Node Bound flag, the Entry Locked flag, and the Untouched flag. The Last Holder value is currently not maintained in the explicit messages version of Cashmere. (Carnival, in fact, has not been updated to run with any Cashmere versions.) The Home Node Bound flag is propagated lazily on the message acknowledgements from the home node. The remaining two flags are only accessed during initialization (which is not timed in our executions) and still use the Memory Channel. Future work should change these to use explicit messages.

4.2 Twins

The Twins component manages the space reserved for page twins, and also performs basic operations to manage twins and diffs. The Twin component interface exports two sets of functions to handle the management and the basic operations. The exported interface and underlying implementation are found in `csm_twins.h` and `svc_twins.cpp`.

The twin space is held in shared memory to allow processors within the node to share the same twins. Upon allocation, the twins are placed into a list (actually a stack) of free twins. The stack is stored inside the twins themselves. The `_csm_twin::m_pHead` value points to the first twin in the stack, and then the first word of each twin in the stack points to the next twin in the stack. When a processor needs to *attach* a twin to a page, it simply pops a twin off the top of the free stack. If the stack is empty, the Twin component automatically allocates additional space. When finished using the twin, the processor *releases* the twin, placing it at the top of the free stack. The two relevant functions in the external interface are `AttachTwin` and `ReleaseTwin`.

The interface also exports several functions to perform twinning and diffing of pages. The diffing functions include variants that handle incoming diffs (see Section 3). Diffing is performed at a 32-bit granularity, and is limited by the granularity of an atomic `LOAD/STORE` operation.[7]

The most complex part of the Twin component implementation is the management of the twin space. Due to the on-demand allocation of twin space and the sharing of twins between processors, each processor must ensure that it has mapped a particular twin before it can be accessed. Inside the Twin component, all accesses to a twin are preceeded by a `_AttachTwinSpace` call that ensures proper mappings are maintained. Note that this call to `_AttachTwinSpace` is even required when releasing a twin, because the twin space is used to implement the free list.

4.3 Write Notices

The Write Notices component provides a mechanism to send and store write notices. The component uses a two-level scheme where write notices are sent from one node to another node, and then later

[7] Our diffing implementation is based on an XOR operation, but is still limited to a 32-bit granularity due to conflicting accesses between multiple concurrent writers within the node.

distributed to processors within the destination.

The Memory Channel version of Cashmere uses a two-level data structure to implement the write notice support. The top level is a global structure that allows nodes to exchange write notices. The structure is defined by _csm_write_notices_t in csm_wn.h. Each node exports a set of write notice *bins* into Memory Channel space. Each bin is owned uniquely by a remote node, and only written by that node. Each bin also has two associated indices, a RecvIdx and a SendIdx. These indices mark the next pending entry on the receiver and the next open entry for the sender. The two indices enable the bins to be treated as a wrap-around queue.

To send a write notice in the Memory Channel version of Cashmere, a processor first acquires a node lock corresponding to the destination bin, and then uses remote write to deposit the desired write notice in the destination bin. The write notice is written to the entry denoted by the current SendIdx value, and then SendIdx is incremented. Finally, the sender can release the associated node lock.

Periodically, the receiver will process each bin in the global write notice list. The receiver begins with the entry denoted by the RecvIdx, reads the write notice, sets the entry to null, and then increments the RecvIdx. The write notice is then distributed to node-level write notice lists (defined in csm_wn.h) corresponding to the processors on the node. This node-level list also contains an associated bitmap with one entry per page. Each bitmap entry is asserted when a write notice is in the list and pending for that page. This bitmap allows Cashmere to avoid placing duplicate entries in the list.

Both SendIdx and RecvIdx are placed in Memory Channel space, and modifications to these indices are reflected to the network. The sender and receiver always have current notions of these variables and can detect when the bin is full. In this case, the sender will explicitly request the receiver to empty its write notices bin. The explicit request is sent as a message via the RemoteDistributeWN function.

The Explicit Messages version of Cashmere uses the messaging subsystem to exchange write notices between nodes, and therefore does not use the global level write notices. The write notices are simply accumulated into a local buffer, and then the caller is required to perform a _csm_write_notices_t::Flush. This call passes the local buffer filled with write notices to the Message component, which then passes the write notices to a processor on the destination node.

The message handler on the destination simply distributes the write notices directly to the affected node-level write notice lists.

4.4 Memory Allocation

The Memory Allocation component provides memory allocation and deallocation routines. This component is only available in the latest version of Cashmere. In our current environment, we perform frequent comparisons between the latest Cashmere and old legacy versions. For this reason, the new Memory Allocation component is not enabled; instead, we use an older malloc routine (found in csm_message/msg_assist.cpp) that is consistent with the malloc in our legacy versions.

The new Malloc component has been tested, however, and is available for use in programs that require frequent allocation and deallocation. The component can be enabled by asserting the _CSM_USE_NEW_MALLOC macro in csm_internal.h.

5 Message Component

The Message component implements a point-to-point messaging subsystem. Each processor has a set of bi-directional links that connect it to all other processors in the system. The link endpoints are actually buffers in Memory Channel space that are accessed through remote write.

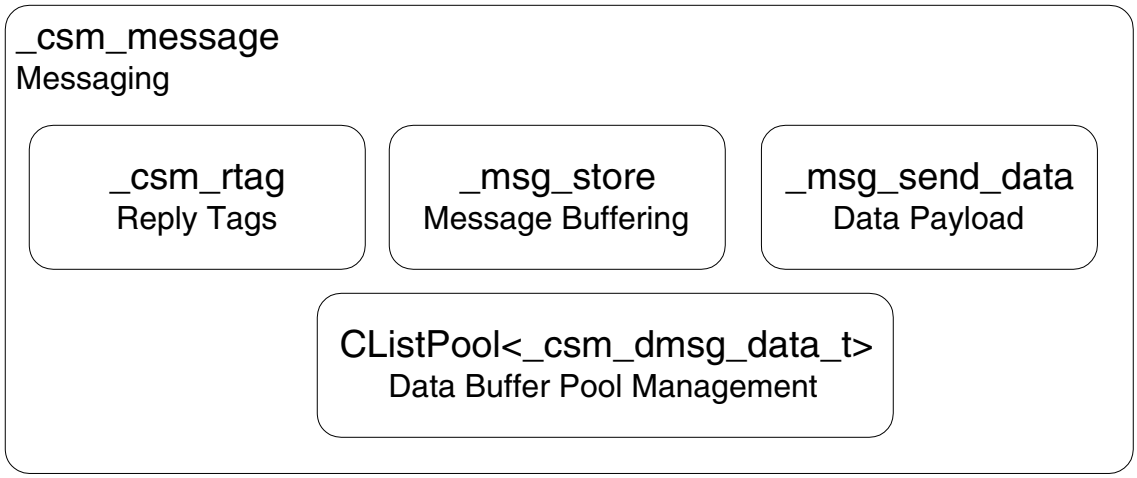

Figure 7: Message component structure.

The component is an aggregate of a set of core routines and a number of support components (see Figure 7). The core routines perform the message notification, detection, and transfer. The support is comprised of the *Reply Tags*, the *Message Buffering*, the *Data Payload*, and the *Data Buffer Pool Management* components.

The Reply Tags component provides a mechanism to track pending message replies. The Message Buffering component maintains a queue of buffered messages that could not be handled in the current Cashmere state. The Data Payload component contains a set of routines to construct the data payloads of each message, and the Data Buffer Pool Management component maintains a pool of data buffers to be used in assembling message replies.

The core routines can be found in `msg_ops.cpp`, and the support classes can be found in `msg_utl.cpp`. The `msg_assist.cpp` file contains routines that serve as helpers for clients accessing the Message component. These helper routines translate incoming parameters into Message component structures and invoke the appropriate messaging interface.

The message subsystem is based on a set of pre-defined messages and their associated *message handlers*. In the following section, we discuss the message types and the structure of the associated handlers. Section 5.2 then discusses the implementation of a round-trip message and Section 5.3 describes the steps necessary to add a new message.

5.1 Messages and Handlers

The Message component defines a number of message types, each distinguished by a unique Message ID (MID) (see `msg_internal.h`). Message types exist for requesting page updates, performing diff operations, and requesting migration, among other tasks. Messages are categorized into three delivery classes:

Sync Synchronous message. The Messaging subsystem will send the message and wait for a reply before returning control to the sender.

ASync Asynchronous message. The subsystem will return control to the sender immediately after sending the message. The receiver is expected to send a reply, but the sender must periodically poll for it with `_csm_message::ReceiveMessages()`.

Very-ASync Very Asynchronous message. This class is the same as ASync, except that the receiver will not reply to the message.

All messages have an associated set of handlers, as defined by the `_csm_msg_handler_t` structure in `msg_internal.h` (see Figure 8).

```
typedef csm_64bit_t (*_csm_handler_t)(const _csm_message_t *msg,
                                      _csm_message_t       *reply,
                                      _csm_msg_post_hook_t *pPostHook);

typedef int (*_csm_post_handler_t)(csm_64bit_t          handlerResult,
                                   const _csm_message_t *reply,
                                   _csm_msg_post_hook_t *pPostHook);

typedef int (*_csm_reply_handler_t)(const _csm_message_t  *reply,
                                    _csm_msg_reply_hook_t *phkReply);

typedef struct {
    csm_64bit_t          mid;

    _csm_handler_t       pre;
    _csm_post_handler_t  post;
    _csm_reply_handler_t reply;

} _csm_msg_handler_t;
```

Figure 8: Type definitions for the structure used to associate handlers with a message.

The "Pre" Handler Incoming messages are initially handled by the Pre handler. This handler is responsible for reading the message body, performing the appropriate action, and then formulating a reply if necessary. The handler's return code instructs the core messaging utility on how to proceed. A Pre handler has the choice of four actions:

Reply return the formulated reply to the requestor

Store this message cannot be processed currently, so store it in a message buffer for later processing

Forward this message cannot be handled at this processor, so forward it to the specified processor

No-Action No further action is necessary

The "Post" Handler Like the Pre handler, the Post handler is executed at the message destination. The core messaging utility will act on the Pre handler's return code (for example, send the reply formulated by the handler) and then invoke the Post handler. This handler is responsible for performing any necessary cleanup. The handler returns an integer value, however currently the value is ignored by the core messaging utility.

The "Reply" Handler The Reply handler is called at the message source, whenever the reply is ultimately received. This handler is necessary to support the pipelining of messages. Pipelining across messages to multiple processors can be accomplished by using Asynchronous messages. As described above, the Message component returns control to the caller immediately after sending an Asynchronous message. The Reply handler provides a call-back mechanism that the Message component can use to alert the caller that a reply has been returned. After the Message component executes the Reply handler, it can then clear the reply buffer. The Reply handler thus serves the dual purpose of notifying the caller when a reply has arrived and notifying the Message component when a reply buffer can be cleared.

The handlers are called directly from the core messaging routines, and so they must adhere to a well known interface. At times however, the handlers must be passed some type of context. For example, a Pre handler may need to pass information to the Post handler, or a Reply handler may need to be invoked with certain contextual information. Cashmere solves this problem by including a *hook* parameter in each handler call. There are two types of handler hooks. The `_csm_msg_reply_hook_t` structure is passed to the Reply handler. A messaging client is responsible for allocating and destroying it. The `_csm_msg_post_hook_t` structure is allocated as a temporary variable by the core messaging routines and passed from the Pre to the Post handler. The reply hook structure uses a more concise format since its members are dependent on the type of the hook. The message post hook structure should eventually be updated to use the same type of structure. The two types are shown in Figure 9.

Example A page update message obtains the latest copy of a page and copies that data into the local working copy of the page. The Pre handler of this message first ensures that the local processor has read permission for the page, and then formulates a reply with the appropriate page data. The core message utility executes the Pre handler, returns the reply, and then calls the associated Post handler. Based on information stored in the message post hook, the Post handler restores the VM permission for

```
// _csm_msg_post_hook_t
// This structure is used to pass information from the pre-hook to
// the post-hook. The pre-hook only needs to fill in the information
// which its corresponding post-hook will need. (Note that most of
// this information is available in the incoming packet, but there
// is no guarantee that the packet will be available at post time.
typedef struct {

  csm_64bit_t  vaddr;    // Virtual address
  csm_64bit_t  pi;       // Page index
  int          pid;      // Processor ID
  csm_64bit_t  MsgMask;  // Message mask
  csm_64bit_t  hook;     // Function-defined
  csm_64bit_t  hook2;    // Function-defined

} _csm_msg_post_hook_t;

// _csm_msg_reply_hook_t
// Reply handlers are called from various places by the messaging
// subsystem.  Clients of the messaging subsystem can pass arguments
// to the reply handler through this structure. The structure simply
// contains a type field and a void *. The void * should point to the
// argument structure, as defined by the client.
// ``rhk'' stands for reply hook.

const  int _csm_rhk_null       = 0;
const  int _csm_rhk_req_page   = 1;
const  int _csm_rhk_req_qlock  = 2;
const  int _csm_rhk_malloc     = 3;
const  int _csm_rhk_diff       = 4;

typedef struct {

    int   type;     // one of the above _csm_rhk_* consts
    void  *psArgs;  // Points to a structure determined by the type

} _csm_msg_reply_hook_t;

// Each type of reply hook has a special rhk type. A simple example:
typedef struct {

    csm_64bit_t   vaddr;
    csm_64bit_t   tsStart;          // Timestamp at start of fetch
    int           bWrite;           // Is this a write access?
    csm_64bit_t   action;           // Action taken by home node
    unsigned long start;

} _csm_rhk_req_page_t;
```

Figure 9: Message hook types used to pass context to the handlers.

the page, thus allowing the permission operation to overlap the reply handling on the source node. The Reply handler on the source node simply parses the reply and copies the page data to the local working copy.

```
struct _csm_dmsg_header_t
{
    csm_v64bit_t    mid;        // Message ID
    csm_v64bit_t    reqno;      // Request number
    csm_v64bit_t    size;       // Size of message
    csm_v64int_t    iSrcPid;    // Source of current msg
                                // (reqs and replies are separate msgs)
    csm_v64bit_t    bInUse;     // True if this link is in-use
    csm_v64bit_t    options[11];
};

const int _csm_dmsg_data_size =
                (2 * _csm_page_size) - sizeof(_csm_dmsg_header_t);

typedef struct {
  csm_v64bit_t   data[_csm_dmsg_data_size/sizeof(csm_v64bit_t)];
} _csm_dmsg_data_t;

struct _csm_dmsg_t {                            // 2 Pages
   _csm_dmsg_header_t  hdr;
   _csm_dmsg_data_t    data;
};
```

Figure 10: Message buffer structure.

5.2 Round-Trip Message Implementation

Message buffers are implemented by the `_csm_dmsg_t` type illustrated in Figure 10. Each buffer consists of a header and a data section. The header contains the message ID (MID), the message request number, the size of the message, and the processor ID of the message source. The `bInUse` flag is asserted if the link is currently in use. The `options` field is used only during initialization. The following text describes the implementation of a round-trip message.

To begin, the sender will call a helper function in `msg_assist.cpp`. The helper function will package the intended message data into a Message component structure and call the appropriate `SendMessage` wrapper function (`MessageToProcessor`, `AsyncMessageToProcessor`, `VeryAsyncMessageToProcessor`).[8]

[8]The `MessageToProtocol` function is a holdover from earlier, protocol processor versions of Cashmere. In these versions, this helper function mapped the desired processor ID to the appropriate protocol processor ID. Without protocol processors, the mapping is an identity function.

In SendMessage, the core messaging utility will first create an *rtag* to mark the pending message reply. Then the messaging utility will copy the message header and data to the request channel's Transmit region. Different methods are used to copy data (*i.e.* 64-bit copy, 32-bit copy, or a diff operation) depending on the MID. The different copy routines are contained in the _msg_send_data class. In the final step of sending a message, the messaging utility will assert the recipient's polling flag by writing to the associated Transmit region.

The process of receiving a message begins when an application's polling instrumentation detects an asserted polling flag. The instrumentation transfers control to the ReceiveMessages function. This function examines all the incoming request and reply channels and begins processing a message (or reply). The first step in processing is to compare the MID against the current *message mask*. Any MIDs that are currently masked are moved from the message buffer to a message store managed by the _msg_store class. The message must be removed from the buffer in order to avoid possible deadlock (especially in Cashmere versions that may require forwarding). When the mask is reset, the caller is responsible for explicitly flushing the message store via a call to FlushMsgStore.

If the message is not masked then the core messaging utilities will process the message via the appropriate Pre and Post handlers. This step of processing depends on the MID, the message type, and the installed handlers (see _csm_message::HandleMessage()).

Both requests and replies are processed through ReceiveMessages, although requests are handled through HandleMessage and replies are handled through ReceiveReply. A messaging client is responsible for calling RecieveMessages so that the incoming request or reply can be handled or stored. Once the message is removed from the buffer, either because processing is finished or because the message is stored, the channel's bInUse flag will be reset. The sender can then push another message through the channel.

5.3 Adding a New Message Type

There are four steps to adding a new message type:

- Add a new MID into msg_internal.h. The new MID must not share any asserted bits (aside from the 0x0feed prefix) with existing mids.
- Create a helper function that builds the Message component data structures and invokes the appropriate SendMessage wrapper. This function should be placed in msg_assist.cpp.
- Add the appropriate Pre, Post, and/or Reply handlers to msg_assist.cpp. The necessary handlers will be determined by the type of message and the necessary processing.
- Register the handlers in the _csm_daemon_register_handlers function.

The most difficult step is to determine the necessary handlers. Pre handlers are mandatory for all messages. Post handlers are only necessary when some type of "cleanup" is needed after the reply is sent. Reply handlers are needed only when the reply has data to be processed.

6 Synchronization Component

The _csm_synch component provides for cluster-wide locks, barriers, and flags, and also for per-node locks and specialized atomic operations. Figure 11 shows the structure of _csm_synch. The _synch_flags, _synch_locks, and _synch_barrier classes are responsible for the implementation of the cluster-wide synchronization operations.

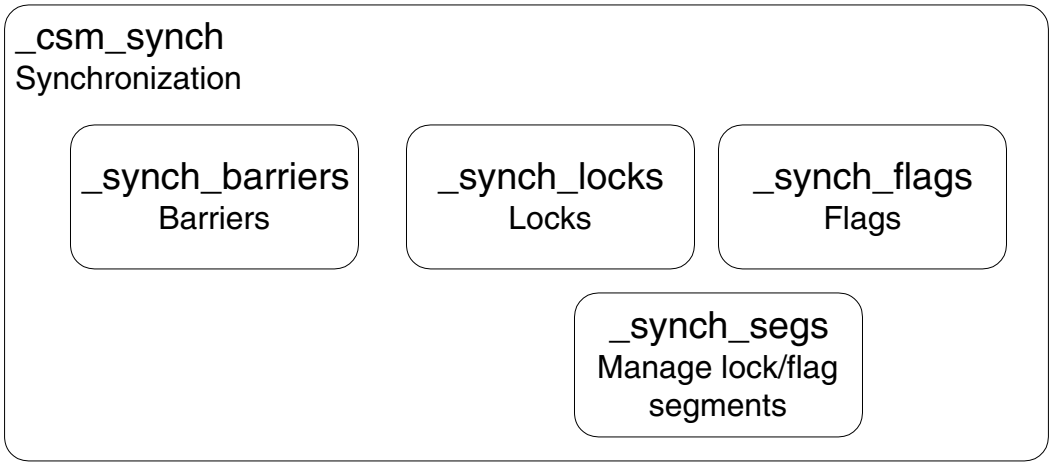

Figure 11: Synchronization component structure.

Cluster-wide synchronization operations are tied directly to the protocol; specifically, the cluster-wide synchronization operations trigger protocol Acquire and Release operations. Lock Acquire and Release operations trigger the associated protocol operation. Barriers are implemented as a protocol Release, followed by global synchronization, and then a protocol Acquire. Waiting on a flag triggers an Acquires; updating a flag triggers a Releases.

In the following discussion, we focus only on the implementation of the synchronization mechanisms, beginning with cluster-wide synchronization and ending with the per-node synchronization mechanisms.

6.1 Cluster-wide Synchronization

The cluster-wide synchronization operations take advantage of the SMP-based cluster by using a two-level synchronization process. A processor works at the node level first, using hardware shared memory mechanisms (see Section 6.2); it then completes its synchronization at the global level.

The global synchronization operations have been developed in two versions: one version leverages the full remote-write, broadcast, and total ordering features of the Memory Channel; the other version is built with explicit messages only. The implementation of barriers and flags is very similar in the two versions. In the case of locks, however, the implementations are very different. We consider the individual synchronization operations below. We then detail the _synch_segs class, which manages the memory backing of the lock and flag structures.

Barriers Barriers are implemented with a single manager that gathers entrance notifications from each node and then toggles a sense variable when all nodes have arrived at the barrier.

```
class _csm_global_barrier_t {

public:
    csm_v64bit_t         arrival_flags[csm_max_nid];
    csm_v64bit_t         episode_number;
    csm_v64bit_t         global_sense;
};

class _csm_barrier_t {
public:
    csm_64bit_t            m_bmArrivalsNode;
    csm_v64bit_t           m_EntranceCtrl;
    csm_v64bit_t           m_ExitCtrl;
    _csm_global_barrier_t  *m_pGlobalBarrier;
};
```

Figure 12: Type implementation of Cashmere barriers.

The type definitions for a barrier ar shown in Figure 12. Again, the structure follows a two-level implementation. The `_csm_barrier_t` is held in shared memory, while the `_csm_global_barrier_t` is mapped as a broadcast Memory Channel region. The `m_bmArrivalsNode` and `m_EntranceCtrl` fields in `_csm_barrier_t` track the processors within the node that have entered the barrier. The former field is a bitmap identifying the particular processors; it is passed into the `_prot_release::Release` function so that the code can determine when the last sharer of a page has arrived at the barrier. (During a barrier, only the last sharer needs to perform a diff.) The latter field counts the local processors that have arrived at the barrier. By analogy, the `m_ExitCtrl` field counts the number of processors that have left the barrier.

The last processor on a node to arrive at the barrier notifies the manager of its node's arrival. In `_csm_global_barrier_t`, the manager maintains an array of per-node arrival flags. In the Memory Channel version of Cashmere, these arrival flags are kept in Memory Channel space, so they can be updated with remote write. In the explicit messages version, the flags are updated via an explicit `CSM_DMSG_BARRIER_ARRIVAL` message.

When all nodes have arrived at the barrier, the master toggles the `global_sense` field in `_csm_global_barrier_t`. Again, this field is updated either via remote write or by an explicit `CSM_DMSG_BARRIER_SENSE` message, depending on the Cashmere version.

In the explicit messages version, the master sends individual `CSM_DMSG_BARRIER_SENSE` messages to the other nodes. Our prototype platform has eight nodes, so this simple broadcast mechanism is reasonable. If the number of nodes wee to increase, a tree-based barrier scheme might provide better performance.

Locks The _synch_locks class provides mutual exclusion lock operations. An application may perform an Acquire, a conditional Acquire, or a Release operation.

The Memory Channel-based locks take full advantage of the special network features. In the code, these locks are referred to as ilocks.[9] The locks are represented by an array with one element per node, and are mapped on each node into both Receive and Transmit regions. The regions are mapped for broadcast and loopback.

A process begins by acquiring a local node lock. It then announces its intention to acquire an ilock by asserting its node's element the array via a write to the appropriate location in the Transmit region. The process then waits for the write to loopback to the associated Receive region. When the write appears, the process scans the array. If no other elements are asserted, then the lock is successfully acquired. Otherwise, a collision has occurred. In the case of a normal Acquire, the process will reset its entry, back off, and then try the acquire again. In the case of a conditional Acquire, the operation will fail. The lock can be released simply by resetting the node's value in the lock array.

This implementation depends on the Memory Channel. First, it uses remote writes with broadcast to efficiently propagate the intention to acquire the lock. It also relies on total ordering to that each node sees the broadcast writes in the same order.

The explicit messages version of locks does not leverage any of these features. This version is borrowed from TreadMarks [1]. It has been modified only to optimize for the underlying SMP-based platform. The locks are managed by a single processor that maintains a distributed queue of lock requestors.

```
// Queue-lock values for the ``held'' field
const int _csm_ql_free         = 0;
const int _csm_ql_held         = 1;
const int _csm_ql_wait         = 2;

// _csm_qlock_t: Queue-based locks. Does not rely on bcast or total ordering.
//
typedef struct {
   int           tail;
   int           next[csm_max_cid];
   volatile int  held[csm_max_cid];
   int           mgr;
   csm_lock_t    lNode;

   csm_64bit_t   padding[2];
} _csm_qlock_t;
```

Figure 13: Definition of queue-based lock type used in Explicit Messages version of Cashmere.

[9] This name was introduced in the first version of Cashmere, where these locks were the *internal* locks. In the current Cashmere version, a more appropriate name may be *mc-locks*, since the implementation depends on the Memory Channel.

The type definition for queue-based locks is pictured in Figure 13. The *tail* field is valid on the manager node and points to the last processor in the request queue. The `next` array has one entry per processor on the node; each entry points to the succeeding PID in the request queue. The `held` field is a flag indicating the processor's request status: free, held, or wait. The `mgr` field indicates the manager PID of the lock, while the `lNode` field is a node-level lock that allows any processor on the manager node to perform management operations. This optimization is allowed by the SMP-based platform.

The steps to acquire a queue-based lock are as follows:

1. Place the requestor's PID at the tail of the request queue.
2. Register the requestor's intention to acquire the lock at the site of the old tail.
3. The requestor should wait for a lock handoff response from the old tail processor.

If the requestor and lock manager are on different nodes, then the first step is accomplished by sending an explicit `CSM_DMSG_REQ_QLOCK` message. Upon receipt of this message, the manager will append the requestor's PID to the request queue by updating the `tail` field of its `_csm_qlock_t` structure. If the requestor and lock manager are on the same node however, the requestor can update the `tail` field using the `lNode` lock to serialize access to shared memory.

The second step can also be accomplished through either an explicit message or shared memory, depending on whether the manager and the tail are on the same node. Through either mechanism, the `_csm_qlock_t` structure on the tail node should be updated so that the tail's `next` field points to the requestor.

In the third step, the requestor will simply spin until its `held` field changes to `_csm_ql_held`. If the tail is on the same node as the requestor, the tail can set the requestor's *held* field directly through shared memory; otherwise the tail will send an explicit `CSM_DMSG_ANS_QLOCK` to the requestor. The associated message handler will set the `held` field.

Flags The implementation of Cashmere flags is very straightforward. In the Memory Channel version, the flags are simply mapped as broadcast regions. In the explicit messages version, the broadcast is accomplished via explicit messages.

In the Memory Channel version of Cashmere, both lock and flag structures are mapped as broadcast regions. And while the size of the structures varies, they both require the same VM support. The `_synch_segs` class manages the associated Memory Channel mappings and provides a mechanism to map lock or flag IDs to a physical location in memory. The class was designed to allow for dynamic allocation of lock and flag segments.

6.2 Node-level Synchronization

Cashmere also provides a number of functions for synchronizing within a node. The functions provide general mutual exclusion locks and also atomic operations. The functions can be split into three categories: "ll", "local", and "safe". A function's category is specified in its name. The functions can be found in `synch_lnode.cpp` and `synch_ll.s`.

The "ll" functions are low-level functions written in assembly language using the Alpha's Load Linked/Store Conditional instructions. The Load Linked instruction loads a memory address and sets a guard flag on that address. A Store operation to that address (by any local processor) automatically resets the guard flag. A Store Conditional operation succeeds only if the guard flag is still set. These two instructions can then be used to implement various atomic operations.

The "ll" functions attempt to perform their atomic operation once, but may fail because of a concurrent Store. The higher level "local" functions repeatedly call the respective "ll" function until the operation succeeds. The "local" functions also call `_csm_message::ReceiveMessages` after a failed "ll" function in order to avoid deadlock.

The "safe" functions are associated with locks. A safe acquire will succeed if the caller already owns a lock. These functions are used to help allow re-entrancy in message handlers.

7 Miscellaneous Component

The Miscellaneous component contains support routines for optimized data copy operations, exception handlers, and statistics. The component also contains routines to allocate shared memory and Memory Channel space. See the code for more details.

8 Software Engineering Issues

The Cashmere code is organized in a directory hierarchy reflecting the internal components. Files and directories in the hierarchy follow a naming convention that identifies their scope in the hierarchy. All files prefixed with a "csm_" are globally visible; other files are prefixed with the name of the component in which they are visible.

To help with code readability, Cashmere uses a modified Hungarian Notation convention, which prefixes each variable name with a type abbreviation. Table 1 lists many of the prefixes used in Cashmere code.

In addition to naming conventions, it is recommended that a Cashmere installation employ RCS or some other source code control system. At the University of Rochester, we have written a local script (`bl_csm`) that *builds a level* of code. This script attaches a symbolic name to the current version of each Cashmere source file, thereby capturing a snapshot of the development process. The script also prompts for a set of comments describing the build level and stores these comments in the `levels.txt` file stored under the root RCS directory. By convention, a programmer builds a level when a new stable code version is reached or when the Cashmere system is benchmarked.

9 Debugging Techniques

Cashmere bugs fall into two main categories based on their impact. Bugs in the first category cause Cashmere to fail immediately, either by crashing or by failing an assert check. Bugs in the second

Prefix	Type
p	pointer
a	array
i, n	integer
b,f	Flag (boolean) value
m_	class member
sc_, s_	static class member
sql	sequential list
_	internal to enclosing scope
bm	bitmask
dbg	debug
ts	timestamp
prx	pointer to a Receive region
ptx	pointer to a Transmit region
phk	message post hook
rhk	message reply hook
idx	index variable
pid	processor ID
nid	node ID
cid	compute processor ID (ID within the node)

Table 1: Cashmere modified Hungarian Notation.

category are caused by races and may not show on all executions or may not manifest themselves until some time after the bug occurs.

The first category of bugs can be tracked by using a debugger. Unfortunately, Cashmere currently is not able to open each application process under its own debugger. There are a few workarounds to help with debugging, however. First, most debuggers can attach to running processes. If Cashmere is invoked with the "-D" flag, then the process will print a message and enter an infinite loop if an assert fails. The user can then attach a debugger to the failed process and check the process state. More simply, the Cashmere application itself can be started inside a debugger. Then process 0 can be controlled by the debugger. These workarounds are admittedly weak, but hopefully they can be helpful in limited cases.

The second category of bugs are tracked only by post-mortem analysis of an event log. The event log is usually created by the `CSM_DEBUGF` macro. (This macro is ignored in the optimized executable.) The macro takes a debugging level and additional, printf-style arguments; it prints those additional arguments if the program's debugging level is greater than or equal to that specified by the initial argument. The debugging level can be set with the "-d:X" command line switch.

Cashmere contains many `CSM_DEBUGF` statements; the number has grown as the code has developed. Currently, the statements are too verbose to trace a lengthy program; they are mainly useful in small test cases. `CSM_DEBUGF` tracing is especially ineffective when debugging difficult races. In most cases, the statements significantly perturb execution and eliminate the race. To help debug races, we have developed two test programs and a special debugging log function designed to minimize logging overhead.

The `test.cpp` program is a relatively simply test of Cashmere. The `wtest.cpp` program is a much more complicated test. Both applications can be found in the CCP hierarchy, which is described in Appendix B. Both operate on a small array, which is partitioned according to the number of processors. The processors operate on all partitions, with the programs verifying data at each computational step. The sharing patterns are relatively easy to follow, but they strongly exercise the Cashmere sharing mechanism. A programmer can view the event logs and track down problems based on the known (and readily visible) sharing patterns.

As an alternative to `CSM_DEBUGF` statements, the programmer can rely in the test applications on a special `_csm_dbg_print_test_data` function, defined in `misc_ops.cpp`. This function is designed is dump a large amount of useful information in as little time as possible, thereby minimizing the perturbation due to logging. The function prints out the first data value in each processor partition, along with the barrier and lock IDs, the local logical clock value, and information specific to the call site. These calls have been placed in key points throughout the protocol, and can be enabled by asserting `_CSM_DBG_TEST` in `csm_internal.h`. The macro should be set to "1" in `wtest` or to "2" in `test`. To enable the full set of these calls, also assert the `DBG` macro in `prot_internal.h`.

Calls to `_csm_dbg_print` are especially effective in debugging the `wtest` application. The main computation phase in this application has each processor visit each array element and assert the bit corresponding to its processor ID. Races are avoided by first obtaining a per-partition lock. This access pattern creates a large amount of very fine-grain sharing.

When the application fails, it prints out the incorrect array element. The programmer can then determine which bit is incorrectly set, and walk backwards through the event log to determine the

cause of the failure. This technique is at times tedious, but is often the only way to discover and correct a subtle race condition in the implementation.

In the rare case when even `_csm_dbg_print_test_data` creates too much perturbation, a programmer can use the `COutputBuffer` class in `misc_uout.[h,cpp]` to keep a log directly in memory. After failure, the programmer can arrange for the log to be dumped to screen or disk.

Another useful tool is `dis`. This tool disassembles an executable, allowing the programmer to associate a program counter value with a line of source code (assuming the executable includes debugging information). Most Cashmere exception handlers print a program counter value so that the exception can be traced to the source code.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[2] A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, LA, January 1995.

[3] M. Fillo and R. B. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1):27–41, 1997.

[4] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2):12–18, February 1996.

[5] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the Twenty-Fourth International Symposium on Computer Architecture*, pages 157–169, Denver, CO, June 1997.

[6] W. Meira. Understanding Parallel Program Performance Using Cause-Effect Analysis. Ph. D. dissertation, TR 663, August 1997.

[7] R. Stets, S. Dwarkadas, L. I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, Toulouse, France, January 2000.

[8] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997.

[9] R. Stets. Leveraging Symmetric Multiprocessors and System Area Networks in Software Distributed Shared Memory. Ph. D. dissertation, Computer Science Department, University of Rochester, August 1999.

A Known Bugs

This Appendix covers known bugs in the current Cashmere system.

- Cashmere with first-touch home nodes (*i.e.* no migration) crashes in barnes, clu, lu, and water-spatial. The version of the code will run, however, when the home node assignments used in an earlier evaluation [7] are specified via the "-l" switch.
- The `wtest` application does not run on 32 processors.
- The CC compiler outputs a large number of warnings when compiling the Cashmere library.

B Cache Coherence Protocol Environment

In our research environment we commonly use several different shared memory protocols, including different versions of Cashmere. The Cache Coherence Protocol (CCP) environment provides a single set of standard code and makefile macros for application development. These macros are mapped at build-time to the target protocol. Currently, the CCP environment supports Cashmere, TreadMarks [1], Broadcast Shared Memory [9], and a plain hardware shared memory system. The environment is designed in a modular manner such that a new protocol can be incorporated simply by creating a makefile macro definition file and by adding appropriate code macros to an existing program.

In the following sections, we will discuss the CCP code and makefile macros, including some usage examples. As one specific example, we will describe the Cashmere-CCP build environment.

B.1 CCP Code Macros

The CCP environment contains a set of macros for both C and Fortran source code. These macros provide access to the basic operations exported by shared memory protocols, such as initialization, memory allocation, and synchronization. To ensure portability, a CCP program must use these macros and avoid any direct invocations of a specific protocol's interface. The program's entry point, *i.e.* `main` (C) or `program` (Fortran), must also be named `CCP_MAIN` or `PROGRAM`, respectively. At build-time, CCP replaces these labels with internal labels that allow the protocol to provide the program's entry point.

The complete set of CCP code macros is contained in the `ccp.h` header file. This file contains the appropriate macro mappings for each supported protocol. The specific mappings are selected by the `CCP_NAME` macro, which is set on the build command line.

Figure 14 contains a small subset of `ccp.h`. At the top of the file, each supported protocol is given a unique macro setting. The remainder of file is split into sections containing the code macro for the associated protocol. The Figure shows a subset of the macros in the Cashmere section, including the include statements, initialization and lock operations. Sections for the TreadMarks protocol are also partially shown in order to provide a flavor of the file's organization. As noted above, CCP currently supports four distinct protocols and also several variants of each protocol.

B.2 CCP Makefile Macros

CCP provides a set of makefile macros that make the build process portable. These macros can be separated into two groups. *Build* macros allow the programmer to specify the desired system, along with other build parameters. *Description* macros allow the targets, dependencies, and commands to be specified in a manner that is portable across the various underlying protocols.

Build Macros There are five build macros that control the selection of underlying protocols and other build options. Normally, these macros are set in the top of the application's makefile or on the `make` command line. Based on the build macro settings, the description macros are automatically set. The five build macros are listed below:

```
/*-----------------------------------------------------------
 *
 * ccp.h
[lines omitted]
#ifndef CCP_H
#define CCP_H

#define CCP_CASHMERE     00
#define CCP_TREADMARKS   01
[lines omitted]

#if CCP_NAME == CCP_CASHMERE
#  include "csm.h"
#  include <stdlib.h>
#elif CCP_NAME == CCP_TREADMARKS
[lines omitted]
#endif

#if CCP_NAME == CCP_CASHMERE

#  define CCP_MAIN                      app_main_
#  define ccp_init_memory_size(s)       csm_init_memory_size((s))
#  define ccp_init_start(argc,argv)     csm_init_start(&(argc), &(argv))
#  define ccp_init(size,argc,argv)      { csm_init_memory_size((size));   \
                                          csm_init_start(&(argc), &(argv)); }
#  define ccp_init_fstart(b,e)          csm_init_fstart((b), (e))

#  define ccp_init_complete()           csm_init_complete()
#  define ccp_distribute(pGlobal,size)  csm_distribute((pGlobal),(size))
#  define ccp_time                      csm_time
#  define ccp_lock_acquire              csm_lock_acquire
#  define ccp_lock_release              csm_lock_release
[lines omitted]
#endif /* CCP_NAME == CCP_CASHMERE */

#if CCP_NAME == CCP_TREADMARKS

#  define ccp_init_memory_size(size)
#  define ccp_init_start(argc,argv)     Tmk_startup((argc), (argv))

[lines omitted]
```

Figure 14: The basic structure of `ccp.h`. Many macro definitions have been omitted due to space constraints. The online version has the complete macro definitions.

PROT controls the choice of shared memory protocol.

LIB_VERS specifies the protocol library version.

C_VERS specifies protocol-dependent processing that must be performed on the application source.

CLVL controls the compilation level (optimized or debug).

LIB_XTRA specifies extra libraries that should be linked.

The file `ccp_build.txt` in the online documentation directory describes the supported settings for these macros. Figure 16 shows a simple example that uses the CCP makefile macros, both the build and description macros, to build a Cashmere application. We will discuss this example further below.

Description Macros The description macros are used to set up the targets, dependencies, and commands. The macros ensure that the proper support files are included or linked and that the proper tools are invoked with the proper flags. These macros are defined in `shared.mk`, which is shown in Figure 15. The description macros specify the compiler, linker, tool flags, file locations, and protocol-specific targets. The application makefile should use the macros prefixed with `CCP`. These macros are set to the protocol-specific settings by a low-level, protocol-specific makefile `$(PROT).mk` that is included at the top of `shared.mk`. `PROT`, of course, is a build macro set directly by the programmer.

The protocol-specific makefiles are responsible for exporting all of the macros required by `shared.mk`. This requirement allows the same makefile to be used for any of the supported CCP protocols.

Figure 16 shows an example CCP makefile for the `sor` (successive over-relaxation) application. The makefile is divided into two main sections. In the first section, the CCP makefile should set the five build macros and then include the `shared.mk` definition file. (In most versions of `make`, command-line assignments will override any makefile settings.) As described in the previous paragraph, the build macros will drive the description macro settings defined in `shared.mk`.

In its second section, the CCP makefile defines the targets, dependencies, and commands necessary to build the application.

B.3 Cashmere-CCP Build Process

The primary tool in the Cashmere-CCP build process is the `csm_cc` script. This script is actually a wrapper of the compiler and the linker, and is meant to be called in place of those two tools. The wrapper script performs a number of pre- and post-processing steps required to create the final Cashmere executable.

Figure 17 shows the script's basic control flow. As mentioned, the script performs either compilation or linking. The initial step in the script is to scan the command line for source files. If source files are present, the script will compile the files according to the steps shown in the left of the flow chart.

```
########################################### shared.mk
[lines omitted due to space concerns]
SHARED = /s23/cashmere/ccp
SHARED_INC = $(SHARED)/include
include $(SHARED_INC)/$(PROT).mk

########################################### Exported macros
#Include paths
INC = -I$(SHARED_INC)

# C/C++ compiler and linker
CC = $($(PROT)_CC)
CCP_CC       = $($(PROT)_CC)
LD = $($(PROT)_LD)
CCP_LD       = $($(PROT)_LD)
CCFLAGS = $(INC) $($(PROT)_INC) $($(PROT)_CCFLAGS)
LDFLAGS = $($(PROT)_LDFLAGS)

# Fortran compiler and flags
CCP_FC      = $($(PROT)_FC)
CCP_FCFLAGS = $($(PROT)_FCFLAGS)
# Fortran linker, flags, and extra libraries
CCP_FL      = $($(PROT)_FL)
CCP_FLFLAGS = $($(PROT)_FLFLAGS)
CCP_FLLIBS  = $($(PROT)_FLLIBS)

# CCP include paths, C/C++ flags, linker flags
CCP_INC      = $($(PROT)_INC)
CCP_CCFLAGS  = $($(PROT)_CCFLAGS)
CCP_LDFLAGS  = $($(PROT)_LDFLAGS)
# CCP C/C++ optimized, debug, profile settings
CCP_CCOPT    = $($(PROT)_CCOPT)
CCP_CCDBG    = $($(PROT)_CCDBG)
CCP_CCPRF    = $($(PROT)_CCPRF)

# Paths for application, CCP, fortran support objects, binaries
CCP_APPS_OBJ = $($(PROT)_APPS_OBJ)
CCP_APPS_BIN = $($(PROT)_APPS_BIN)
CCP_OBJS     = $($(PROT)_OBJS)
CCP_FC_OBJS  = $($(PROT)_FC_OBJS)

# Path for library and library specification
CCP_LIB_PATH = $($(PROT)_LIB_PATH)
CCP_LIB = $($(PROT)_LIB)

# Additional protocol-specific targets
CCP_TARGETS = $($(PROT)_TARGETS)
```

Figure 15: The `shared.mk` file sets the macros used in CCP makefiles.

```
# SOR Makefile
#

# Choose the desired protocol.
PROT = CSM_2L

# Choose code level: optimized
CLVL = OPT

# Code version: polling
C_VERS = _POLL

# Protocol library selection: normal
LIB_VERS =

# Extra Library Selection: none
LIB_XTRA  =


base: appl

# Load in protocol-independent macros
include ../../../include/shared.mk

TARGET = $(CCP_APPS_BIN)/sor
OBJS = $(CCP_APPS_OBJ)/sor.o $(CCP_OBJS)

appl: banner  $(TARGET) $(CCP_TARGETS)

$(TARGET): $(OBJS)   \
           $(CCP_OBJS)             \
           $(CCP_LIB_PATH)/lib$(CCP_LIB).a
        if [ ! -e $(CCP_APPS_BIN) ] ; then mkdir $(CCP_APPS_BIN); fi
        $(LD) $(OBJS) -o $(TARGET) $(LDFLAGS)

$(CCP_APPS_OBJ)/sor.o: sor.c $(SHARED_INC)/ccp.h
        if [ ! -e $(CCP_APPS_OBJ) ] ; then mkdir $(CCP_APPS_OBJ); fi
        $(CC) -c $(CCFLAGS) -o $@ sor.c

clean:
        rm -f $(TARGET) $(TARGET)_s *.S $(CCP_APPS_OBJ)/*.o *.s *~

tags:
        etags *.c *.h
```

Figure 16: Example makefile for the SOR application. The build macros combine with the `shared.mk` include file to define the `CCP_` description macros.

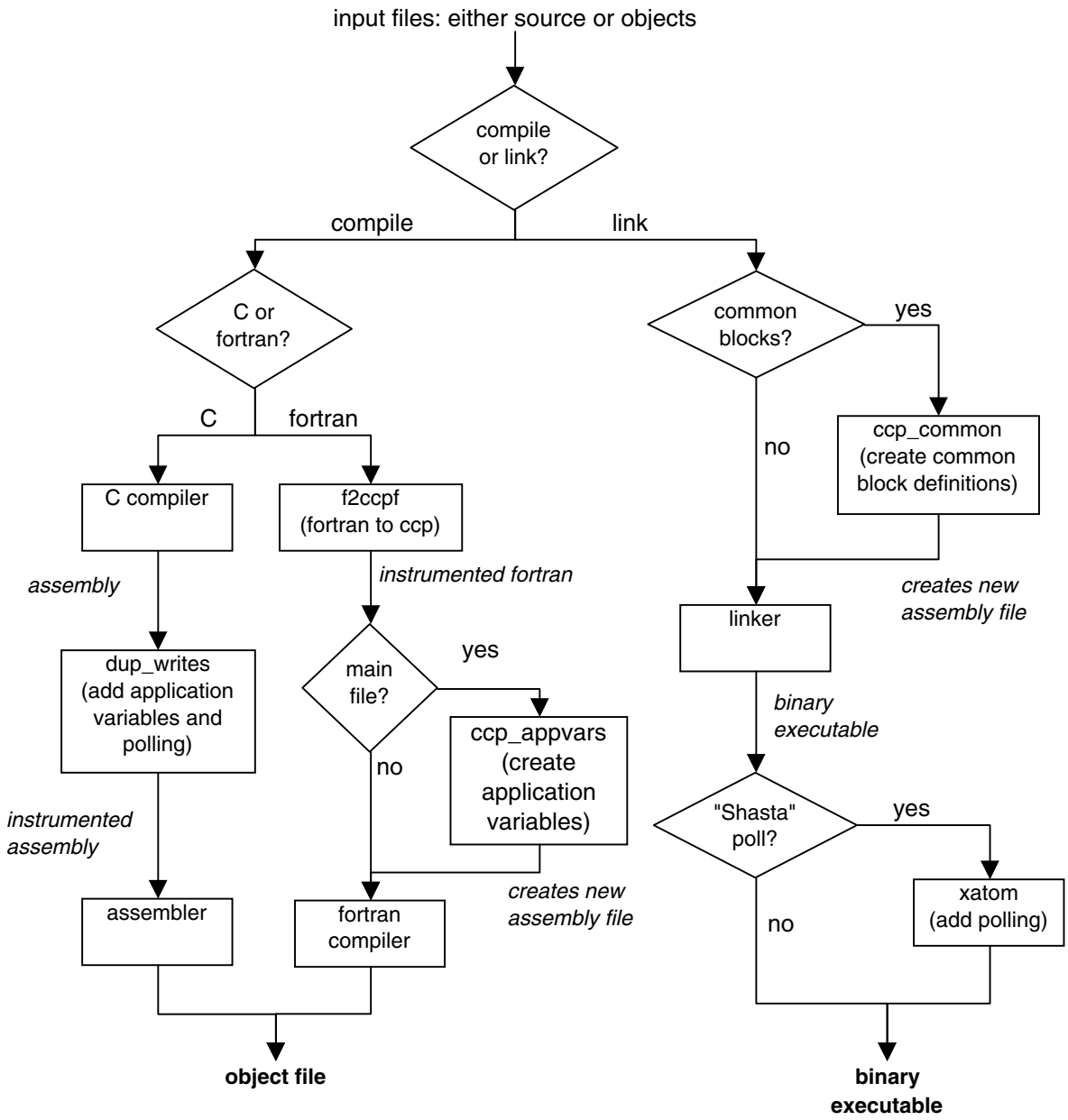

Figure 17: Control flow for `csm_cc`. Intermediate files are denoted by italicized labels on the transitions.

Compilation The `csm_cc` script can compile both C and Fortran files. For a C file, the script will first use the C compiler to create an assembly file that can be instrumented with Cashmere-related code. The `dup_writes` executable adds polling and/or write doubling[10] instrumentation to the assembly file. The executable also sets a number of variables, which are called *application variables* by our Cashmere system, in the application's data segment. Cashmere checks the application variables at runtime to determine how the application was built, *i.e.* the type of polling or write doubling instrumentation.

The final step in a C source compilation is to run the instrumented assembly file through the compiler to produce the final object file.

The compilation process for Fortran source is slightly different, due to differences in the capabilities of the C and Fortran compilers. Our base C compiler (gcc) allows registers to be reserved, while none of our Fortran compilers have this capability. The polling instrumentation performed by `dup_writes` requires two reserved registers, and therefore this instrumentation cannot be used in our Fortran programs. Instead, our Fortran programs use Shasta-style polling, which walks through the binary executable and performs live register analysis to determine what registers are available at polling sites. This difference in compiler capabilities creates a significant difference in the build processes.

A Fortran source file is first run through the `f2ccpf` script. This script locates the main entry point and changes the entry point into an `app_main` subroutine, which is called directly by the Cashmere library. (The Cashmere library hijacks the program's main entry point in order to perform special initialization.) Once the application's main entry point is identified, the `csm_cc` script will also create a simple assembly file containing the application variables. This assembly file is compiled into an object file and ultimately linked into the final binary executable. The final step in the compilation process is then to compile the instrumented Fortran source file into an object file.

In addition, the Fortran code requires a special interface file to the Cashmere C code. The `fort_if.c` file in the `ccp/src` directory provides simple wrapper routines that make the Cashmere library calls available to the Fortran source. In fact, `fort_if.c` is written using CCP calls, so this interface file makes all CCP-supported protocol libraries available to Fortran applications. As described in the previous section, the protocol is chosen through the CCP `PROT` build macro.

Once all the C or Fortran application object files are built, the files can be linked together using the `csm_cc` script. After linking the executable, Shasta-style polling can also be applied. The following text describes the linkage operation.

Linkage To begin the linkage process, the `csm_cc` script uses the `ccp_common` script to parse the object files for common blocks. (Common blocks are used extensively in Fortran, but the assembly language declaration for common blocks can be used by any high level language.) The `ccp_common` script creates a small assembly file with definitions for each common block. These definitions ensure that the common block addresses fall within any address range required by the underlying protocol library.

In the next step, the `csm_cc` script simply links together the specified object files. This step produces a binary executable file. For certain polling choices, *i.e.* Shasta polling, a final post-processing step is required to add the polling instrumentation. This step uses the `xatom`[11] binary modification tool to

[10] Write doubling is used by an early Cashmere version [5] to propragate shared data modifications as they occur.

[11] The xatom tool is a version of `atom` [2] that has been modified to allow the insertion of single instructions.

place inline polling instrumentation in the binary. As mentioned above, the instrumentation tool uses only dead registers in the polling instrumentation.