TECHNICAL REPORT 94–14

A Framework for Customizing Coherence Protocols of Distrbuted File Caches in Lucas File System

Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto,
Shigekazu Inohara, and Takashi Masuda

December 9, 1994

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE
A Framework for Customizing Cohrence Protocols of Distributed File Caches in Lucas File System

AUTHORS
Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto, Shigekazu Inohara, and Takashi Masuda

KEY WORDS AND PHRASES
distributed file system, distributed shared memory, cache coherence protocol, protocol customization

ABSTRACT
In cooperative applications such as group CAD and group software development systems, multiple processes communicate with each other by sharing complex data consisting of nested structures and pointers. Although the sharing of complex data structures in the distributed environment is achieved through the technology of distributed shared memory, a single cache coherence protocol cannot efficiently serve various access patterns generated by cooperative applications. By having a framework for allowing cooperative applications to customize coherence protocols according to their behaviors, hence, significant improvement of run-time efficiency is expected. This paper describes a framework of protocol customization for the sharing of volatile and persistent data in cooperative applications. The major obstacle in user-level customization of protocols is that there are too many states and state transitions in an unabstracted protocol to enable average users to describe them. The protocol customization system (PCS) in this paper solves this problem by introducing high-level model for protocol description that abstracted away non-determinism of messages, synchronization among hosts, and local paging actions. Consequently, users can define with brief descriptions cache coherence protocols that are adapted for particular applications. The ability of PCS to describe different kinds of protocols is examined in this paper, and its run-time performance and memory usages are investigated.

REPORT DATE	**WRITTEN LANGUAGE**
December 9, 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
17	33

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
Submitted to the 15th International Conference on Distributed Computing Systems.

DISTRIBUTION STATEMENT
First issue 35 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

A Framework for Customizing Cohrence Protocols of Distributed File Caches in Lucas File System

Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto,
Shigekazu Inohara, and Takashi Masuda

December 9, 1994

Abstract

In cooperative applications such as group CAD and group software development systems, multiple processes communicate with each other by sharing complex data consisting of nested structures and pointers. Although the sharing of complex data structures in the distributed environment is achieved through the technology of distributed shared memory, a single cache coherence protocol cannot efficiently serve various access patterns generated by cooperative applications. By having a framework for allowing cooperative applications to customize coherence protocols according to their behaviors, hence, significant improvement of run-time efficiency is expected. This paper describes a framework of protocol customization for the sharing of volatile and persistent data in cooperative applications. The major obstacle in user-level customization of protocols is that there are too many states and state transitions in an unabstracted protocol to enable average users to describe them. The protocol customization system (PCS) in this paper solves this problem by introducing high-level model for protocol description that abstracted away non-determinism of messages, synchronization among hosts, and local paging actions. Consequently, users can define with brief descriptions cache coherence protocols that are adapted for particular applications. The ability of PCS to describe different kinds of protocols is examined in this paper, and its run-time performance and memory usages are investigated.

1 Introduction

New applications that improve the efficiency of cooperation of users using the power of distributed computers have been under intensively research and development. In such applications that support cooperation of users, which we refer hereafter "distributed cooperative applications," users' processes in distributed hosts often share large data structures containing nested structure and pointers. Moreover, the core part of the shared data structures is persistent (namely, stored in the secondary storage) in such distributed cooperative applications that support the long-term cooperation of users; users of a group software development system share hierarchically versioned source and object code of new software products, and users of a group CAD system shares design works in the secondary storage.

By equipping files with the memory-mapping interface and coherent distributed file caches, a distributed file system provides distributed cooperative applications with an efficient way of sharing persistent data with complex structure. Through memory mapping, multiple processes can share files containing nested data structures and pointers without relocation, marshaling nor conversion.[1] In file sharing in a single host, we can even avoid copying of data between processes. With coherent distributed file caches, in which the technology of cache coherence protocols in the area of distributed shared memory (DSM) [19, 10, 13, 6] is extended to the persistent data management, users can cooperate via shared persistent data structures transparently both in a single host and in distributed hosts.

Although this memory-mapped distributed file caches offer a convenient vehicle for distributed cooperative applications, a potential problem of this approach is that we cannot expect a single cache coherent

[1] Pointer conversion is one of the major bottlenecks of persistent systems, even though we can implement it more efficiently than it once was because of the evolution of lazy conversion techniques called pointer swizzling [33, 32, 9].

protocol to serve a wide range of cooperative applications efficiently. A number of DSM coherence protocols have been proposed so far, yet no single protocol is known to be the best. Distributed cooperative applications, in particular, have much opportunity for application-specific customization of coherence protocols. For example, if a version management system for cooperative applications uses three-level hierarchy consisting of personal and group workspaces on top of a global database [16], a coherence protocol can exploit the hierarchy to determine whether to deliver each user with a strictly up-to-date copy of shared data or a nearly up-to-date copy, thereby reducing the number of messages for invalidations and updates. A distributed file caches, hence, should have a framework for incorporating application-specific customization and optimization of cache coherence protocols.

This paper presents a protocol customize system (PCS) of memory-mapped distributed file caches, which enables distributed cooperative applications to optimize the behavior of file caches in application specific manners. PCS is integrated in the distributed cache management layer of a distributed file system, and allows applications with brief description to define their own cache coherence protocols. Although there are several attempts to allow users to choose a preferable cache coherence protocol from a fixed set of system-provided protocols (for example, Munin [6]), no existing system provides as much flexibility as the present system does.

The primary technical challenge of PCS is to reduce the number of states and state transitions in a protocol that the users of PCS describe, without introducing much degradation of run-time performance nor excessive memory usage. This support is essential for user-level description of file caches coherence protocols, because they tend to be more complicated than a coherence protocol of simple DSM because the former handle both volatile and persistent states of caches. For example, a file cache protocol should write dirty caches back to disks periodically in order to cope with system crashes, which is not necessary for simple on-memory DSMs. The protocol can also exploit disks in each host, in addition to the disks in the original file location, as a temporal persistent storage of caches. PCS decreases the number of user-observable states and state transitions using three methods: (1) reordering messages so that users of PCS can be freed from non-essential interleaving of message arrival, (2) hiding local state transitions that are not essential for state transitions of distributed file caches, and (3) allowing the sequential description of protocols; by assigning a thread for each page that interprets the state machine of the page, some states in a protocol are implicitly represented as a set of local variables and program counter of the thread.

The rest of this paper is organized as follows. Section 2 presents the backgrounds of PCS and problems in protocol description. Section 3 fully describes the three methods for decreasing user-observable states and state transitions in PCS. Section 4 contains the complete definition of PCS and concrete examples. Section 5 estimates the ability and performance of PCS. Section 6 discusses related work, and Section 7 concludes this paper.

2 Backgrounds

This section describes motivations and problems for protocol customization.

2.1 Motivations for Coherency Protocol Customization

Customization of file cache coherence protocols for cooperative applications has been motivated mainly by three reasons. First, although a number of protocols have been proposed until today, and intensive studies have been done for characterizing these protocols, no single coherence protocol is known to be the best. Existing protocols for DSM include sequential consistency [19], weak consistency [10], release consistency [13], lazy release consistency [17], causal consistency [1], and various protocols included in Munin [6]. An adaptive cache coherence protocol proposed by A. Cox and R. J. Fowler [7] provides an example of the effectiveness of protocol customization. Their protocol watches the run-time accesses of applications to caches, and switches the cache coherence behavior between a write-invalidate policy and a migratory policy. This protocol has a state machine slightly modified from a standard write-invalidate protocol. This kind of improvement can be

done easily in a user-controlled way with a framework of protocol customization. Namely, users can enable and disable the customization of protocols on page basis, on user basis, and based on other user-defined criteria. In the case of memory-mapped files, we can further exploit the mapping-mode information such as 'read-only mode' and 'read-write mode' for the customization of protocols [31].

The second reason is that contention of concurrent data accesses occur more often in cooperative applications than in numerical applications that the existing DSM protocols target at. Since intensive accesses to small set of shared data items is one of the fundamental nature of cooperative applications, some contention cannot be avoided by programming efforts. In addition, accesses to shared data items are performed not only by human users but also by computer programs, because some cooperative applications have mechanisms for notifying processes of particular changes of shared data, and the processes activates user-defined codes for reacting to the changes. With these mechanisms, which are similar to triggers in active database systems [23, 8, 12, 3], successive triggering of actions and reactions might occur without human intervention. This fact implies that the performance of cache coherence protocols can be one of the primary factor that characterizes the overall performance of cooperative applications.

Third, behaviors of concurrent processes in a cooperative applications are usually regulated using some transaction frameworks. We are able to exploit the semantics of transactions to optimize cache coherence protocols in application-specific (or transaction-model-specific) ways. A protocol for version management systems with three-level hierarchy that mentioned in the introduction is an example of this kind. Note that, like DSM protocols, no single framework for cooperative transactions serve all cooperative applications. A number of transaction frameworks for cooperative applications have been proposed (e.g. altruistic locking [29], split transactions [27], sagas [11], transaction groups [30], and participant transactions [15]), and each of them has its own advantages and disadvantages. This fact also supports the approach of cache protocol customization.

2.2 Platform of Protocol Customization

The platform of this research is a memory-mapped distributed file system which we call Lucas File System (LucasFS). In order to make the sharing of persistent data structures containing pointers, LucasFS provides users with a 64-bit single-level storage by exploiting the full power of the forthcoming 64-bit microprocessors.[2] A user-level server in each host (called cache server or CS) maintains the coherence of memory mapped files on the distributed single-level store by communicating with each other through user-customizable cache coherence protocols. In each host, a cache server maintains the mapping and protection of caches by interacting with the Mach micro-kernel through Mach external pager interface [21]. In addition, a storage server (SS), which exists for each exported file system, accesses its disk in response to CSs' requests, and maintains the network-wide state of pages shipped from the disk.

LucasFS maintains the mapping addresses of files in users' address spaces, and if users are in cooperation, LucasFS maintains the mapping address identical for each user. Cooperative applications hence can share persistent data structures without pointer conversion overheads, byte-copying between the server and clients that could be optimized with the copy-on-write sharing technology, and marshaling and unmarshaling in client-to-client communication. Since the address management scheme is not strongly related to the cache coherence issue we discuss in this paper, we omit its details. This can be found in another document [14].

2.3 Problems of Protocol Description

The most difficult problem in writing a coherence protocol of distributed file caches is that a protocol, if it specifies all aspect of cache management actions, often has so many states and state transitions. A complete coherence protocol of file caches has to manage a number of elements including the asynchrony of process interaction, time-outs and retries in the non-reliable networks, page replacement policies, page fault handling and page protection management, periodic write-back of dirty caches, and copy-on-write sharing; each of

[2]The name Lucas is after a nineteenth-century French mathematician, E. Lucas, who introduced the famous puzzle "tower of Hanoi." It is said that the world will end if someone completes the puzzle of 64 plates after $2^{64} - 1$ moves.

them increases the number of states and state transitions. Since it is apparently nonsense to let protocol designers (i.e., users who wish to customize a protocol) write such full details, they need a model of protocol description with relatively high level of abstraction.

From the experience of writing several file cache protocols, we found that the increase of states and state transitions of a protocol is mainly due to three reasons.

1. Non-determinism of message arrival in the distributed system causes transitional states that are non-essential for the specification of a protocol.
2. Interaction between the cache server and micro-kernel causes internal states that are non-essential for the specification of a protocol.
3. The programming of a protocol based on a single-threaded message-driven style discourages protocol designers to represent some states implicitly with the program context (that is, the program counter and local variables in nested function calls).

Non-determinism of message arrival: In contrast with a cache coherence protocol in a shared-bus multiprocessor machines where a single arbiter can control the overall behavior of multiple caches, all hosts in the distributed environment work independently and messages among them could sometimes be lost and duplicated. The non-determinism of message arrival, in particular, causes non-essential states in a protocol because it cannot determine whether messages arrive in the order of their intention. Suppose a cache server waits for message A and message B. The cache server must be programmed to receive these messages correctly in both cases, A arrives before B and B arrives before A. Consequently, the protocol is forced to have two transitional states "waiting for A" and "waiting for B" in addition to the states "before receiving A and B" and "received both A and B". In addition, another message can arrive while the cache server is waiting for the message A or B. The protocol is thus forced to contain the code after each message reception for queuing or discarding messages that arrive when the cache server is not able to handle them. In fact, protocols rarely make use of this kind of non-determinism of message arrival, hence we can safely hide the non-determinism from their programming model.

Interaction between the kernel and cache server: In each host, a cache server and micro-kernel interact with each other for file cache operations (invalidation and update) and physical memory management. The interaction between these two modules always uses asynchronous messages in order to maximize the concurrency of the two modules; in particular, the ability of receiving messages from other hosts during a local file cache operation in the micro-kernel is essential. This design, however, results in increased number of non-essential internal states and state transitions of protocols. For example, suppose a cache server, CS, holds a read-write copy of a page (see figure 1). Although CS stays in a single state from a point of view outside CS, it takes two different states locally since either the micro-kernel or the pager owns the copy. The internal states, which are invisible from the outside, do affect the complexity of the state machine in CS. For example, when a CS receives a cache flush request, the CS can flush the page immediately if CS owns the page (the left side of figure 1), while CS must first send a flush request to the micro-kernel to own the page if the micro-kernel owns the page (the right side of figure 1).

Lack of Program Context: The programming of protocols is usually based on the single-threaded event-driven style, where each message that arrives to a cache server is handled by the single event dispatch loop that invokes a procedure that handles an event specified by the message. Each handler procedure, corresponding to a single event, finds a target page by examining the contents of the message, looks up the current state of the target page, and chooses an appropriate action to the event according to the current state of the page. This programming style is well suited to RPC stub generation, because most RPC stub generators assume a server to have a single message entry point and corresponding event dispatch loop.

This programming style, however, forces protocol designers to represent all message-related intermediate states as explicit states. Since the stream of processing is scattered, protocol designers cannot describe a

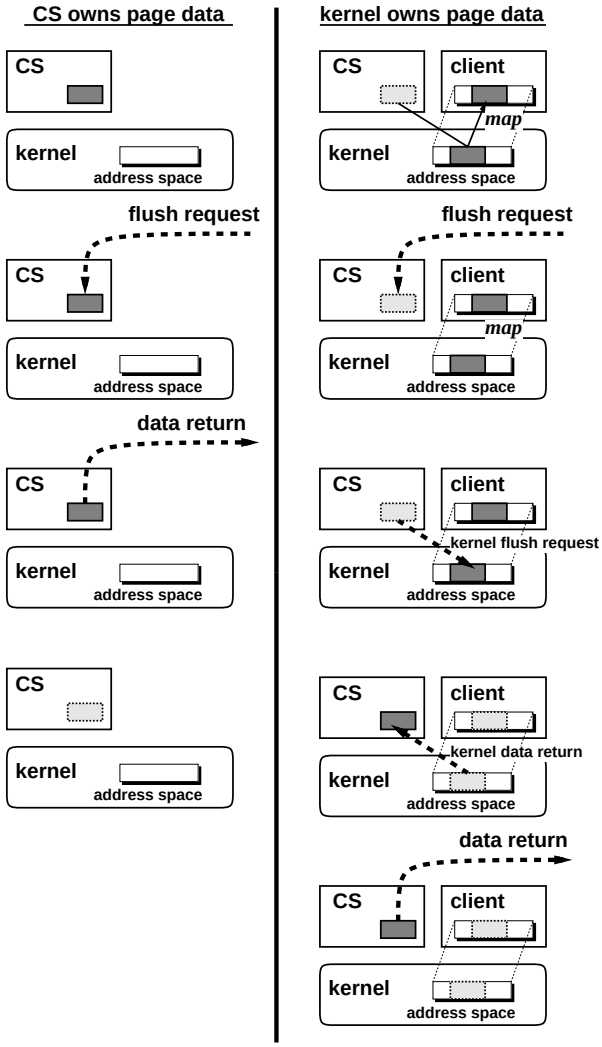
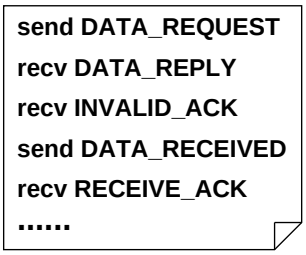
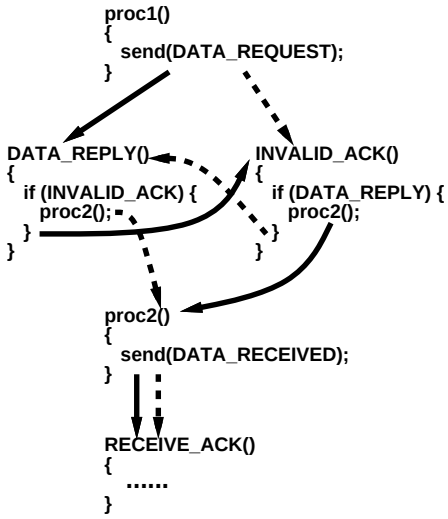


Figure 1: Interaction between the cache server and micro-kernel.



(a) An ideal protocol flow.



(b) An actual protocol description.

Figure 2: Scattered description of a sequential protocol flow.

transition in an intuitive way. To be more concreate, consider a situation where a transition from state X to state Y consists of the reception of two messages (`DATA_REPLY` and `INVALID_ACK`), followed by the transmission of a `DATA_REQUEST` message [Figure 2-(a)]. In this case, the single-threaded event-driven style forces a protocol designer to describe three intermediate states "`DATA_REPLY()`," "`INVALID_ACK()`," and "`RECEIVE_ACK()`" as in Figure 2-(b).

If, on the other hand, a protocol designer can assign a event dispatch loop for each page by introducing multi-threading in the cache server, some states are represented implicitly with the program context (that is, the program counter and local variables in nested function calls). In the above example, the code for transition from state X to state Y contains all the necessary action: "`send DATA_REQUEST, receive DATA_REPLY, receive INVALID_ACK, send DATA_RECEIVED`, and `receive RECEIVE_ACK`" Because each page has a thread assigned for the state machine of the page, suspending the thread during the transition from X to Y causes no degradation of response time of other pages.

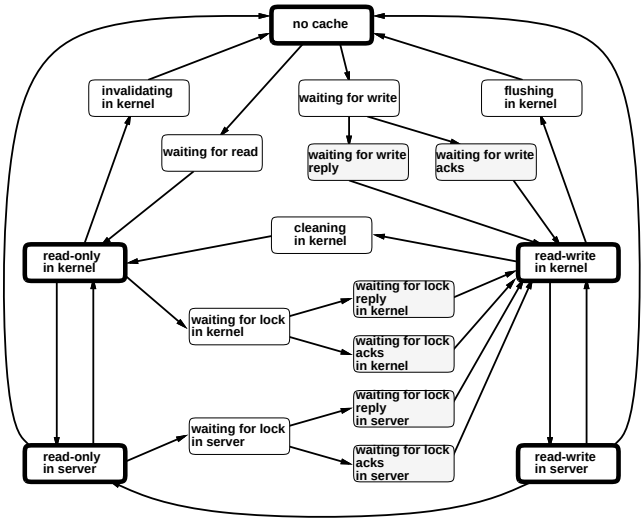
3 Protocol Customize System

PCS provides a high level programming layer for protocol designers where they need to see only a small number of essential states in protocols using the following three methods:

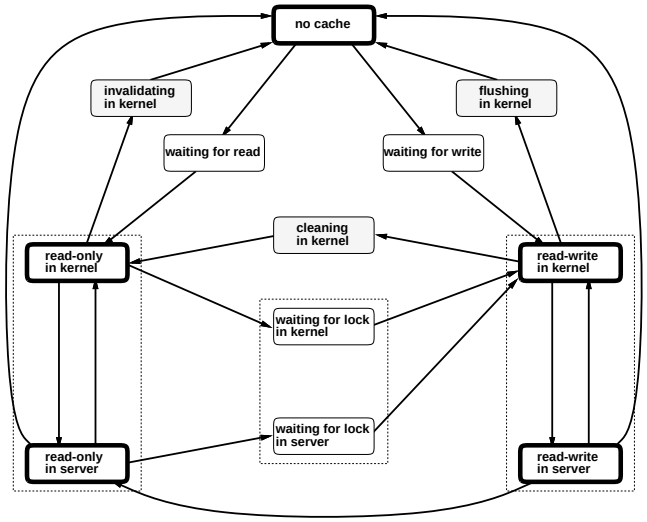
1. reordering messages to eliminate message disorder,
2. hiding unnecessary communications and states between the kernel and the cache server,
3. allowing the sequential description of protocols by assigning a thread for each page.

We describe each point in turn.

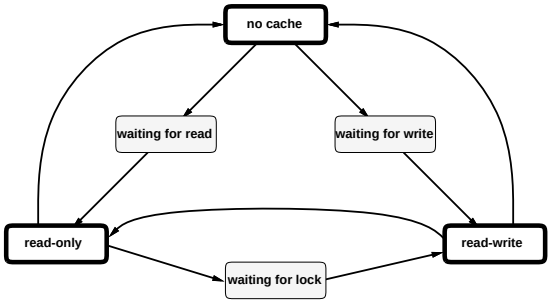
Figure 3-(a) illustrates the state transition graph of the cache server on the write-invalidate protocol where we wrote in the C programming language. This graph consists of eighteen states, including five basic states (in bold boxes) and thirteen transient states, and transitions among them. We use this example throughout this section to show how these transient states are eliminated by applying the above three methods.



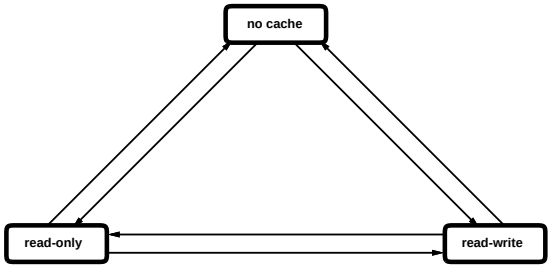
(a) Initial version of states diagram



(b) After reordering messages



(c) After hiding intra-kernel communications



(d) After eliminating message-waiting transient states

Figure 3: States of the write-invalidate protocol.

3.1 Reordering of Message Arrivals

First, PCS make message arrival abstract. Generally, messages in the distributed environment are not received just in the order they are sent. There sometimes exist race conditions message arrival as well. PCS reorders messages to eliminate disorder that users do not intend and transient states to manage disorder. On PCS, users can consider that they receive a message if and only if the message has arrived *and* they can handle the message. Messages that users cannot immediately handle are automatically queued into a message queue. Users specify the kinds of messages to receive with two primitives `select` and `recv`. `select` specifies selective waiting of multiple kinds of messages, while `recv` is regarded as `select` specifying a single kind of messages. When the state machine of a page issues a `select` or `recv` primitive, it is assigned a message selection tag that corresponds to the set of messages it can receive. When a message arrives, PCS enqueues it in a per-page message queue. If the state machine of the corresponding page becomes ready to receive messages, PCS finds a message in a queue that matches the message selection tag of the state machine, and feeds the message to the state machine.

As a result, the PCS code "`recv A ; recv B`" accepts messages A and B regardless of the order A and B arrive. In the example of the write invalidation protocol in figure 3-(a), six states in mesh boxes manage message disorder. After reordering of messages, these six states are eliminated, and consequently the number of states decreases from eighteen to twelve as shown in figure 3-(b).

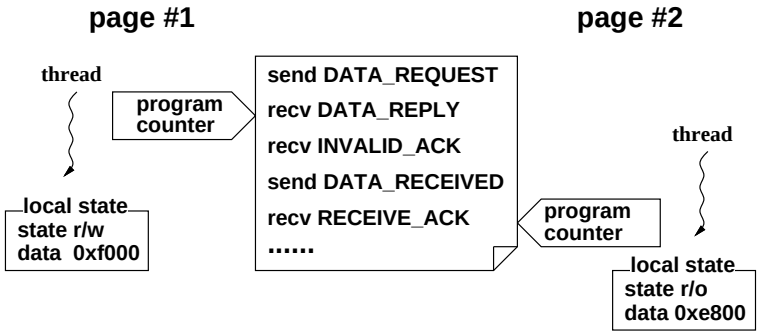


Figure 4: Sequential description of protocols.

3.2 Hiding Internal Kernel States

Second, let us turn our attention to the fact that protocol designers may not know all communications between the kernel and cache server. Hiding these communications also decreases the number of states that protocol designers manage. We call these states *internal states* of cache servers, which are classified into two groups, internal stable states and internal transient states.

Internal stable states result from the fact that either the kernel or the cache server can hold the latest copy of a page as mentioned in section 2.3. Figure 3-(b) depicts such states with broken-lined boxes. PCS are able to provide a view that regards these two states as the same state.

Internal transient states arise from asynchronous message passing between the kernel and cache server. In the unabstracted protocol, the cache server must create a transient state to handle upcalls from the kernel. PCS hides communications between them and manages internal transient states automatically. Internal transient states are eliminated in this way. After the elimination of both internal states of cache servers, six states remain as shown in figure 3-(c).

3.3 Sequential Description of Protocols

PCS allows sequential description of protocols. Because of RPC interface such as Mach MIG and Sun RPC, where callee functions are dispatched according to the kind of messages in the per-server grain, all procedures for a message should be written within the function corresponding to the message. Hence an action to wait for receiving a message in the middle of a transition, for instance receiving an acknowledgement message after a sending message, should be described by adding a new state to wait for the message, which causes increase of states to manage. In PCS, protocol designers can describe protocols in more simple way, because PCS allows them to wait for messages in the middle of a transition. Even a nested selection are allowed in the PCS code, which results in elimination of a non-essential intermediate states. In the example of write-invalidate protocol, three transient states to wait messages, figured in broken boxes on figure 3-(c), are eliminated after this abstraction, and just only three basic states remain finally as in figure 3-(d).

To allow sequential description, however, we need to have extra resources instead of states, namely, threads. Because a thread may suspend somewhere in a PCS code for a page to wait for messages, each page should logically have a thread for managing each page concurrently. Each thread involves local states, a stack and a program counter of each page (Figure 4). A stack accompanied with a thread occupies a page at least. Since this results in reducing the size of cache of a cache server to a half, the efficiency of caching is greatly degraded.

To solve this problem, PCS prevents multiple threads from concurrently suspending to wait for messages by transforming the user-provided source code. PCS first extracts message reception primitives from the original source. After this extraction, a single thread of PCS waits for messages and calls functions, which

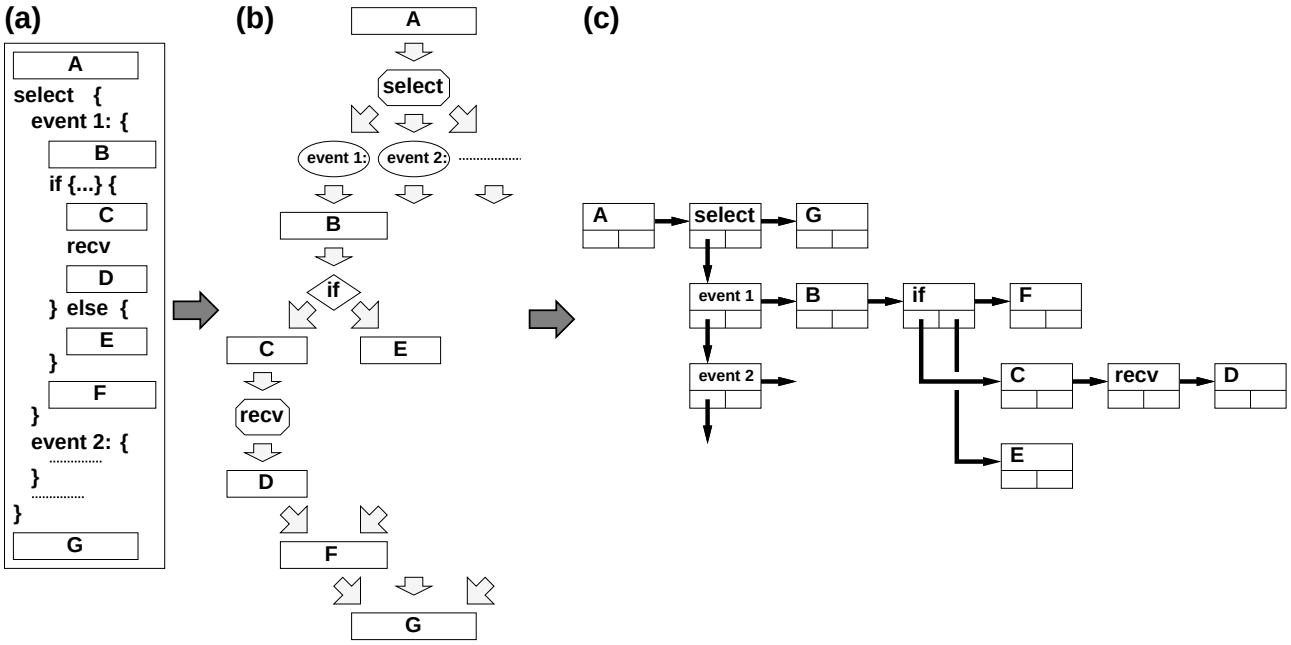


Figure 5: Transformation of PCS sources.
(a) PCS source code, (b) extraction of `select` and `recv`, (c) a parse tree after eliminating blocking primitives.

include none of waiting primitives according to the kind of messages. Figure 5 illustrates the transformation of PCS source codes. Figure 5-(a) shows an original PCS source code that consists of blocking primitives (`recv` or `select`) and sequences of non-blocking primitives, which are represented as boxes labeled A, B, ..., G. PCS parses the source code to extract blocking primitives as shown in figure 5-(b), and reconstructs them into a parse tree where blocking primitives are detached from non-blocking primitives, which can hence be executed without thread suspension, as in figure 5-(c). This transformation can be regarded as a simple form of the transformation to the continuation passing style [2], which is used as an intermediate representation on ML. PCS preserves continuations of waiting primitives, and confines the program context of a thread, such as a program counter and local states, into the continuations.

4 Details of PCS

We present detailed specification and examples of PCS in this section. Since LucasFS consists of two servers called cache server (CS) and storage server (CS), PCS also provides the same structure as LucasFS, so that protocols are described both in CS part and SS part. PCS Language is an interpreter language based on Tcl [26], which has csh-like syntax. PCS adds to Tcl primitives to manage cache states and messages. PCS provides the following abstractions to protocol designers: (1) port, (2) message, (3) event, (4) data, and (5) page.

4.1 Abstractions of PCS

Port A port specifies a destination server to send messages. Ports are categorized into two types, *receive* ports and *send* ports. CS and SS have their own *receive* ports. A file or a page knows the *send* port where it

tag	value	examples
`event`	event name	`create_msg msg "DATA_REQUEST"`
`reply_to`	sender's port	`send $msg(reply_to) reply_msg`
`send_to`	port	`set msg(send_to) $owner_port`
`dataID`	data	`set_data $pageID $msg(dataID)`
`dealloc`	deallocation flag	`set msg(dealloc) 0`
`prot`	protection state	`set msg(prot) RW`
`pageID`	page	`set msg(pageID) $pageID`

Table 1: System-defined fields of a message variable.

is stored (SS port). A *receive* port can be sent to other servers on `reply_to`-field of messages, and it becomes a *send* port. Other's *send* port can also be sent on messages to a third server.

Message A message consists of tuple of field-value pairs, which may include system-defined fields, such as data, ports, protection states, and any other user-defined fields. A message is an associative array in PCS, which is called a *message variable*. Table 1 shows system-defined fields of a message variable, and users can also define and send any values other than system-defined values. A sender first creates a message variable with an event name by the primitive `create_msg`, sets values to fields by `set`, and sends to a receiver by `send`. A received message invokes a message event as described below.

Event Events are received with `recv` or `select` primitives. Granularity of events ranges among a page, a file and a server. For example, mapping of a file is a file-grain event, and a page request is a page-grain event. In addition, events are classified into message events, system events, and user-defined events. These events differ only in their invocation, and they are treated in the same way. Message events are invoked when messages arrive. System events are invoked when system-defined actions take place, such that mapping or unmapping (file-grain events), or a first page fault (page-grain events). User-defined events issued by `lfs_event_async()` and `lfs_event_sync()` provide interface between PCS and user programs. Application programs can use these primitives to offer run-time information to PCS.

Data A data is a unit of memory chunk of sharing among servers and the kernel. It is identified by data-ID. It is set into a page by primitive `set_data` and gotten from a page by `get_data`. `read_data` and `write_data` manage accesses of data to the secondary storage. `alloc_data` allocates data, and `dealloc_data` deallocates it. Data is copied or moved with messages specified by the `dealloc`-field of a message variable.

Page A page is a unit to manage accesses and protection states. A page is identified by page-ID. A page includes a data (an accessible memory chunk) and a protection state. A protection state may indicate one of three states: not accessible, read-only accessible, or read-write accessible. A `set_prot` call associate a protection state with a page. `get_prot` returns the current protection state of the page.

4.2 Example 1: Write-Invalidate Protocol

This subsection presents an example of actual protocol description. Figure 6 is a part of PCS code for cache servers that expresses the write-invalidate protocol. For convenience of the following explanation, the brief summary of the syntax of PCS language. Variables are created and modified by the `set` primitive, and referred by the prefix `$`. Variables are either simple variables or associative arrays. "{" "}" specifies a list-structure. "[" "]" substitutes the result of evaluated contents just like the back-quote in csh.

```
 1: CS_PAGE: select msg {
 2:     CLIENT_DATA_REQUEST: {
 3:         create_msg rmsg DATA_REQUEST
 4:         set rmsg(prot) $msg(prot)
 5:         set rmsg(need_data) $msg(need_data)
 6:         send [SS_port $pageID] rmsg
 7:         recv DATA_REPLY dmsg
 8:         mrecv $dmsg(inv_cnt) INV_COMPLETED
 9:         if {$dmsg(should_return) == 1} {
10:             create_msg ack DATA_RECEIVED
11:             send [SS_port $pageID] ack
12:         }
13:         if {$rmsg(need_data) == 1} {
14:             set_data $pageID $dmsg(dataID)
15:         }
16:         set_prot $pageID $msg(prot)
17:     }
18:     ....
19: }
```

Figure 6: A part of PCS code for write-invalidate protocol.

A protocol usually consists of a large `select`-block. When an message arrives, the contents of the message are assigned to a message variable that is an argument of the `select` statement, and subsequent statements correspondent with the event are executed. If a page request from the kernel arrives, the block labeled `CLIENT_DATA_REQUEST` will be executed. `create_msg` in the third line creates a new message variable named `rmsg` whose event name is `DATA_REQUEST`, which means a request of cache to the storage server (SS). Line 4 and 5 copy contents of `msg` to `rmsg`. `send` primitive in line 6 sends `rmsg` to the storage server of the page. `SS_port` returns the port-ID of SS of given `pageID`. The next `recv` waits for the event named `DATA_REPLY`. If a message containing the `DATA_REPLY` event arrives, PCS sets the contents of the message to a message variable `dmsg`. `mrecv`, which is a syntax sugar of `recv`s with specified times of iteration, waits until the specified count of messages, `$dmsg(inv_cnt)` in this case, are received. Notice that the CS did not know how many invalidation acknowledgements (`INV_COMPLETED`) will come until the reply message (`DATA_REPLY`) has come. This protocol advances correctly even if `DATA_REPLY` comes after some or all `INV_COMPLETED`s have come, because all acknowledgements are queued and are brought out after `DATA_REPLY` has been received. In line 9 to 12, this CS creates the message of `DATA_RECEIVED` and sends it, if SS waits for the acknowledgement (`$dmsg(should_return) == 1`). CS sets received data to the page if need, and change the protection state as requested.

4.3 Example 2: Memory-Mapped Stream

The second example is the memory-mapped stream protocol, where a sender and a receiver communicate with each other through mapped pages. This protocol requires extra calls from sender and receiver, in users' program other than standard memory accesses. The event `SEND` signifies that a written page is ready to

```
CS_PAGE: select msg {
    ....
    SEND: {
        set_prot $pageID N
        create_msg rmsg "DATA_SEND"
        set rmsg(dataID) [get_data $pageID]
        set rmsg(dealloc) 1
        send $receiver(pageID) rmsg
    }
    RECV: {
       if {$data_exists == 1} {
           event_complete "RECV" $pageID
       } else {
           set data_exists -1
       }
    }
    DATA_SEND: {
       if {$data_exists == -1} {
           event_complete "RECV" $pageID
       }
       set data_exists 1
       if {$msg(dataID) == ""} {
           set_data $pageID [alloc_data]
       } else {
           set_data $pageID $msg(dataID)
       }
    }
}
```

(a) PCS code

```
...
lfs_map(..., &p);
for (i = 0; i < length; i += SIZE) {
   /* write operations */
   lfs_event_async("SEND", p+i);
}
...
```

(b) sender's code

```
...
lfs_map(..., &p);
for (i = 0; i < length; i += SIZE) {
   lfs_event_sync("RECV", p+i);
   /* read operations */
}
...
```

(c) receiver's code

Figure 7: A part of code for memory-mapped stream.

lang	protocol	lines
C	write-invalidate	2286
PCS	write-invalidate	195
	memory mapped stream	79
	release consistency	111
	causal consistency	236

Table 2: Comparison of code size for protocols.

send, and the event `RECV` signifies that a page is tried to read. Figure 7 provides the part of the protocol and user-programs. Figure 7-(b) presents a sender's code, which writes mapped pages and sends them, and figure 7-(c) provides a receiver's code, which receives pages and reads them. `lfs_event_async("SEND")` creates an *asynchronous* event named `SEND`, while `lfs_event_sync("RECV")` creates an *synchronous* event named `RECV`. The different of two events is due to the fact that `SEND` need not wait for the completion of its sending, while `RECV` should wait for the completion of data transfer. In the protocol of figure 7-(a), `SEND` changes the protection state to no-access, and sends data to a receiver with the name of `DATA_SEND`. If the data has already been sent when `RECV` is called, in which case `$data_exists == 1`, users' `RECV` call is terminated by `event_complete "RECV"`. If the data has not yet been sent, PCS sets a flag to indicate waiting (`data_exists == -1`), and suspends users' `RECV` call. The event `DATA_SEND` triggers suspended receiver's event `RECV`, and sets sent data.

5 Experiments

The PCS has been implemented on top of the Mach 3.0 micro-kernel on DECstation 5000 (MIPS R3000 33MHz). We have already described four different protocols in PCS to evaluate its effectiveness of description. These four protocols are: (1) write-invalidate protocol, (2) memory mapped stream protocol, (3) release consistency protocol, and (4) causal consistency protocol.

5.1 Comparison of Code Size

Four kinds of protocols above are described in PCS Language, and the lines of protocols are compared (table 2). PCS can describe the write-invalidate protocol about one tenth more briefly than the C programming language. Though a simple comparison does not make sense, it indicates how PCS allows brief descriptions. Though we did not write protocols other than write-invalidate protocol, the memory-mapped stream and the release-consistency protocol can be described in lines shorter than or equal to the write-invalidate protocol of PCS. The causal consistency protocol needs a little more lines than the write-invalidate protocol, but the approximately same codes are used in both protocols, and consequently the protocol can be written based on the write-invalidate protocol with little effort. We can describe each protocol in just one day, which is much shorter term than in the C language.

Furthermore, the protocols (2) and (3) require annotated synchronization primitives, such as `release` and `acquire`, in addition to ordinary memory accesses in user programs. PCS handles these synchronization primitives as user-defined events, which require no more extra description than other kinds of events in PCS.

5.2 Run-Time Performance of PCS Interpreter

Although the main purpose of the current implementation of PCS is to describe several actual protocols and to verify their effectiveness, it is important to know overheads of PCS interpreter. Before we make experiments to estimate overheads of PCS, we have predicted that the cost of overheads to interpret PCS is

action	cost (msec)
`mach_task_self()`	0.031
thread context switch	0.018
page fault (on default pager)	3.9
page fault (on external pager)	1.6
send a null message	7.42
null RPC	19.0
send a message with n pages	$16.6 + 9.43 \times n$

Table 3: Costs of primitive actions.

	local	remote
(a) LucasFS without PCS (msec)	5.93	49.2
(b) LucasFS with PCS (msec)	18.2	62.9
overheads (b) − (a) (msec)	12.3	13.7
ratio (b) / (a)	3.07	1.28

Table 4: The cost of a page fault.

smaller than the cost of exchanging messages among remote hosts. That is the first major reason why we have first selected an interpreter form for implementing PCS. The second major reason is that preprocessing PCS sources is required to prevent wasting memories from the reason that mentioned in section 3.3.

Table 3 shows the cost of primitive actions of PCS on DECstation 5000. The cost of `mach_task_self()` system-call, which gets task's own port, represents the cost where the processor gets into the kernel mode and out of it. A thread context switch is performed with `condition_wait()` and `condition_signal()` explicitly. The cost of a page fault is measured from the time when a memory access begins to the time when it resumes, while the cache server supplies a requested page immediately.

Table 4 provides the page fault costs of (a) LucasFS server without PCS and (b) LucasFS server with PCS, where the coherence protocol of both case is write-invalidate protocol. This table states that (b) costs about three times larger than (a) in the local case, and 28% larger in the remote case. Since the cost of interpreter itself is constant, overheads relatively decrease in the remote case. The assumption that the cost of interpreter is rather non-significant compared with the communication cost is almost true in the remote case. The time for interpreting a statement of PCS costs about 50μsec to 500μsec, depending on its complexity. Table 5 shows the cost of PCS interpreter for performing some PCS primitives. Although overheads of interpreting are less than those of remote communications in current environment, they still occupy a large amount in the local case, and if throughput of network communications increases, overheads of PCS cannot be ignored. We intend to reduce interpreter overheads by making PCS compile a source into a intermediate representation.

6 Related Work

Most existing distributed file systems provide single coherence protocol adapted for a single purpose [20]. For example, Andrew File System [24] provides a single coherence protocol, the session semantics. Since it offers a consistent view of files in a session during open, it cannot provide a shared view of a file for distributed clients if two clients in distributed hosts open and write a file concurrently. Sprite File System [25] provides the sequential consistency with invalidation usually, however it stops caching and switches to a remote service

action	cost (msec)
`Tcl_Eval` "`set a 1`"	0.052
`Tcl_Eval` "`create_msg`"	0.19
`Tcl_Eval` "`send`"	0.46

Table 5: The cost of interpreting Tcl.

when two or more writer share a file simultaneously. Echo File System [4] provides sequential consistency, directories as well as files, with cache and token invalidation. This systems, however, provides single protocol for all types of files.

Some existing DSM systems provide multiple coherence protocols. Munin [6], which is a software DSM, provides multiple protocols, and allows users to select ones according to their behaviors. An adaptive cache coherency protocol of A. Cox and R. Fowler [7] detects accesses of shared data and switches a sharing policy between migratory and replication. In contrast to PCS, these systems offer a fixed set of protocols, and no methods for customizing is provided. FLASH at Stanford University [18], which is a hardware supported DSM system, provides a programmable processor that manages cache coherence, and allows system designers to define coherence protocols. It has flexibility compared with fixed coherence systems, however a for users,

Typhoon and Tempest [28] also provides hardware programable processors, and allows users to customize it. This system has a maximal flexibility, but users are forced to program an unabstracted state machine.

Study of languages to describe protocols provides so called protocol formal description languages, such as LOTOS [5] and PSF [22]. The main purpose of these formal languages is to verify protocols, which include structures of distributed systems, all possible states, and all kinds of messages. These protocol formal description languages require full descriptions to verify their completeness, and hence they are unsuitable for addition or modification to existing protocol descriptions.

7 Conclusion

This paper has described the protocol customize system (PCS) in the Lucas distributed file system, which supports cooperative data sharing by memory-mapping and distributed shared memory methods. PCS decreases states of protocols to manage by three methods of abstraction, abstraction of message arrivals, abstraction of the cache server and the kernel, and abstraction of transient states to wait for messages. By using PCS, protocols can be written in a short term with brief descriptions, with little overheads in both of memory usage and run-time performance. The cost of overheads in interpreting PCS language may further decrease by compilation of PCS source codes.

We are planning to attach dynamic adaptation of multiple protocols to PCS because a single application may not perform well with a single protocol even after some optimizations. Suppose an application consists of distributed processes, which cache shared read-only copies and refer them, and a burst of writes to a single page occurs while a short term. The write-invalidate protocol is appropriate for this application at ordinary time when they share read-only copies, while the write-update or other protocols is more appropriate while write contention occurs. A switchover among multiple protocols can support such situation. Problems accompanied with a switchover follow: (1) Misalliance of states and messages between two different protocols. (2) How to decide a moment for switching protocols. Although user's annotation may solve (2), the PCS server can use reference information and change protocols voluntarily.

References

[1] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing*

Systems. IEEE Computer Society, 1991.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Catriel Beeri and Tova Milo. A model for active object oriented database. In *Proceedings of the 17th International Conference on Very large Data Bases*, pages 337–349, September 1991.

[4] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Degital SRC, 1993.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In *Computer Networks and ISDN Systems 14*, pages 25–59. Elsevier Science Publishers, 1987.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct 1991.

[7] 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*, pages 98–108, May 1993.

[8] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. of the 1990 ACM SIGMOD Conference*, 1990.

[9] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[10] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[11] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of 1987 ACM-SIGMOD International Conference on Management of Data*, pages 249–259. ACM, May 1987.

[12] N. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very large Data Bases*, pages 327–336, September 1991.

[13] 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 Annual International Symposium on Computer Architecture*, pages 15–26. IEEE, May 1990.

[14] Shigekazu Inohara, Keitaro Uehara, Hajime Miyazawa, and Takashi Masuda. Sharing of persistent data structures for distributed cooperative work on versioned 64-bit address spaces. Technical Report 93-07, Department of Information Science, University of Tokyo, December 1993.

[15] Gail E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the 6th International Conference on Data Engineering*, pages 560–567. IEEE Computer Society, February 1990.

[16] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[17] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[18] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr 1994.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[20] E. Levy and A. Silverschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, Dec 1990.

[21] Keith Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, January 1992.

[22] S. Mauw and G. J. Vetlink, editors. *Algebraic Specification of Communication Protocols*. Cambridge university press, 1993.

[23] D. R. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proc. of the 1989 ACM SIGMOD Conference*, 1989.

[24] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. *Communication of the ACM*, 29(3):184–201, March 1986.

[25] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

[26] J. K. Ousterhout. Tcl: An embeddable command language. In *1990 Winter USENIX Conference Proceedings*, pages 133–146, Jan 1990.

[27] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very large Data Bases*, pages 26–37, 1988.

[28] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Apr 1994.

[29] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A strategy for coping with long-lived transactions. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 19.1–19.24, September 1987.

[30] A. H. Skarra and S. B. Zdonic. *Object-Oriented Concepts, Databases, and Applications*, chapter Concurrency Control and Object-Oriented Databases, pages 395–421. ACM Press, 1989.

[31] K. Uehara, H. Miyazawa, S. Inohara, and T. Masuda. Coherence protocols for distributed cooperative applications in Lucas distributed file system. In *IPSJ SIG Reports 94-OS-63*, pages 1–8, Mar 1994.

[32] F. Vaughan and A. Dearle. Supporting large persistent stores using conventional hardware. In Antonio Albano and Ron Morrison, editors, *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 34–53, San Miniato, Italy, September 1992. Springer-Verlag.

[33] P. R. Wilson. Operating system support for small objects. In *Workshop on Object-Oriented Operating Systems*, pages 80–86, 1991.