Building Caches using Multi-Threaded State Machines

Wanda Chiu and John H. Hartman

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Designing a client-side cache for a distributed file system is complicated by concurrent accesses by applications, network communication with the server, and complex relationships between the data in the cache. Despite these difficulties, caches are usually built using threads or finite state machines as the programming model, even though both are inadequate for the task. We have created an infrastructure for cache development based on multi-threaded state machines, which attempts to capitalize on the benefits of both programming models. The state machine allows the global state of the cache to be carefully controlled, allowing interactions between concurrent cache operations to be reasoned about and verified. Threads allow individual operations to maintain their own local state, and propagate that state across transitions of the state machine. We created a prototype of this infrastructure and used it to implement a write-through file block cache that provides multiple-reader/single-writer access to its blocks; although this is a relatively simple cache protocol, it is much easier to implement using multi-threaded state machines than using either threads or finite state machines alone.

1 Introduction

It isn't easy to design a distributed file system cache. Although a client-side cache can improve access times, network traffic, and server load, the usefulness of the cache depends on its hit ratio. The difficulty lies in the trade-off between the cache hit ratio and cache coherency -- on the one hand the cache should hold as much data as possible for as long as possible, on the other hand data in the cache should be consistent (coherent) with the that on the server. This inherent trade-off drives the continuing development of new cache coherency protocols, each attempting to maximize performance by providing the minimal useful level of coherency for the applications it supports using a minimal amount of overhead.

There are several factors that make cache design and implementation difficult. First, the cache must handle simultaneous data accesses by application programs and communication with the server. These two activities are tightly-coupled, and the interactions and dependencies between them are non-trivial. Second, concurrency complicates the cache design. Data accesses can be concurrent, caused by either a single application issuing multiple asynchronous accesses, or several applications issuing synchronous accesses. The interaction with the file server may also introduce concurrency. Messages from the server may arrive asynchronously with respect to data accesses, and the cache may have to handle messages from one or more servers simultaneously. Finally, there may be complex relationships between the objects in the cache. For example, the cache may hold objects of type “block”, “file”, and “directory”, and have to keep track of the relationships between them. An operation on a block may affect the file to which it belongs, and a coherency operation on a file affects its blocks.

Despite the inherent difficulties in designing and implementing a cache coherency protocol, this process has typically been ad-hoc. Each cache is built from scratch, tailored to the particular system and cache coherency protocol that it supports. It is difficult to reason about or modify such a cache; changes to one part of its functionality can have unexpected consequences for the cache as a whole. Correctness is determined through testing and debugging.

In this paper we present a system for designing, developing, and maintaining caches, based on a multi-threaded state machine. The use of an underlying state machine makes the states of the cache and the transitions between those states explicit, simplifying the design and verification of the cache. The state machine is multi-threaded, simplifying support for concurrency and allowing computations to maintain their own private data that aren't encoded in the state machine. We have built a prototype of this system and used it to design and implement a multiple-reader/single-writer write-through cache.

2 Challenges in Cache Design

There are three main challenges in designing a cache: implementing a (potentially complex) coherency protocol; supporting concurrency; and dealing with complex data relationships and aggregate objects.

2.1 Coherency Protocols

A cache coherency protocol ensures that cached data remain consistent with the data stored on the server, so that they are not out-of-date with respect to the server. Different coherency protocols provide different levels of coherency; the goal is to provide a useful level of coherency with a minimum of overhead. In a distributed file system the coherency protocol is necessarily a distributed algorithm, at least requiring communication and coordination between the client and the server. The protocol must deal with non-atomic operations, out-of-order and lost messages, and machine or network failures. Several researchers [Chandra96, Chandra97, Wang98] provide examples of coherency protocols that are affected by these issues; we summarize the problems here.

Ideally, coordination between caches would be an atomic operation; that way there would be no doubt about the resulting states of the two caches. In reality it is difficult to make inter-computer communication atomic. If one cache makes a request of another cache and then waits for a reply, it is possible for other events to demand attention in the meantime. For example, a write or invalidate request during a write-back will need to wait until the write-back is complete.

The cache coherency protocol must also deal with out-of-order or lost messages. Coherency is effected through the message exchange between caches; lost or out-of-order messages can cause inconsistencies in the cache states, causing incoherence. If these problems aren't handled by the underlying communication mechanism, they must be handled by the caches themselves. This is complicated by concurrency -- if a cache sends out a request and waits for a reply, it must also be able to handle replies from previous requests for which the acknowledgment was lost, replies returning in the wrong order, and replies that never arrive at all. Similarly, a cache must be able to handle failures of other machines. A reply may never arrive for a request because the other machine has failed. The cache must not only detect the failure, but recover from it.

2.2 Concurrency

Concurrency also complicates the cache implementation. A cache must handle several operations simultaneously, such as data accesses. These operations necessarily change the cache's state, requiring synchronization to prevent state corruption. One solution is to make all operations non-preemptive, so that the cache handles only one operation at a time. This will prevent corruption, but exacts a high price because some operations wait (possibly a long time) for external events, such as messages from the server. This not only severely limits the cache performance, but can also lead to incorrect behavior and deadlock. Consider a fetch of a block into the cache that requires replacing an existing block, perhaps from the same file. The cache cannot defer the replacement until the fetch is complete, because the fetch cannot complete until the replacement completes. Thus the proper functioning of the cache does not allow operations to be handled serially.

On the other hand, allowing concurrent operations complicates the cache design because of the interaction between active operations and those that have blocked waiting for an external event. Care must be taken to ensure that the operations do not interfere with one another, in particular the active operations to not interfere with those that have blocked. If other operations have run while an operation is blocked, the blocked operation may find the cache state completely changed once it resumes processing. These considerations make concurrency control difficult to reason about and implement.

Concurrent operations also complicate the mechanism for dealing with external events, because there may be several operations simultaneously waiting for different external events. When an event occurs, it must be matched with the correct operation(s) waiting for it. Thus additional bookkeeping is needed to keep track of operations that are waiting for events, and notifying the correct operations when an event occurs.

2.3 Complex Data Relationships

Finally, there may be complex relationships between the data in the cache. For example, a cache may store file blocks, which are aggregated into files, which in turn are members of directories. Modification of a file block may affect the file to which it belongs (e.g. by changing the file's last-modified-time), which in turn may affect the file's directory. Coherency is also complicated by data relationships. Consider a file cache that supports file level consistency. A file can be valid in the cache even when all its blocks are not; a block that is not valid may be missing from the cache, or in the process of being brought into the cache, or being removed. This complicates reading data from a file; if a file were a simple object, which is either present in its entirety or not, then reading data would consist of bringing the file into the cache if necessary, then copying data from the cache into the application's buffer. In a block cache there are many more possibilities -- blocks that are present in the cache can simply be copied, while blocks that are not present must first be fetched. A single access can result in both, because an access can span blocks.

Complex data relationships can also cause operations performed on one object to affect other objects in non-obvious ways. For example, NFS [Sandberg85] returns a file's current modification time when a block is fetched from the server. This modification time is used to verify that the file's blocks of the file are up-to-date and invalidate them if necessary. Thus reading data from one block of a file may cause all the other blocks of the file to be invalidated.

3 Programming Models

Despite the inherent difficulties in building a cache, they are usually designed and implemented in an ad-hoc fashion, using either threads or (less commonly) finite state machines as the programming model. Both models have their advantages and disadvantages for designing and implementing caches, but in general are inadequate.

3.1 Threads

A thread is a useful abstraction for representing a single, independent computation. Each thread has its own execution context and private data (stack), which are saved and restored when the thread blocks. This allows a thread to be blocked without affecting the computation it is performing, and makes threads independent from one another. This in turn makes it possible to reason about, and perhaps verify, each thread of a system independently.

Unfortunately, the threads in a complex system such as a cache often aren't independent. The computations they perform rely on shared data, so that the actions of one thread can affect another. This makes reasoning about the correctness of the system as a whole difficult, and the thread abstraction less useful. Instead of reasoning that individual threads function correctly, the system designer must reason that the individual threads collectively manipulate the shared data correctly, which is a much more difficult problem. As the amount of interdependency between threads grows, so does the weakness of the thread model. Threads are good at modeling manipulations of private data and not so good of shared data, and as the bulk of the computation shifts from using the former to using the latter, the thread abstraction becomes more unwieldy.

3.2 Finite State Machines

A state machine is a concise static description of a dynamic system, such as a cache, and consists of a set of all possible states of the system, a set of events that cause transitions between states, a handler (sequence of actions) for each event handled by each state, and the system's current state. When an event occurs the state machine invokes the appropriate handler (based on the event and the current state) and transitions to a new state when the handler completes. The state machine is thus single-threaded; event handlers run to completion and new events are not handled until the previous handler completes and the state machine enters a new state. This makes event handlers atomic, simplifying the reasoning about their effects and interactions, and eliminates the need for synchronization between threads. If an event occurs for which a handler has not been defined, the event is queued until the state machine reaches a state that can handle it.

By specifying which events are handled by each state, the state machine provides a natural way of reasoning about system correctness and controlling concurrency. All of the system's states are enumerated, as are the events handled by each. Thus, accesses to the shared data of the system are easily controlled: if an event handler accesses the shared data in

a way that should not be allowed given the current state of the system, that event is not handled. Concurrency in a particular state is either allowed or disallowed by either handling or not handling events that represent concurrent operations, respectively.

The state machine model lends itself to automatic verification techniques. The model itself makes designing and modifying caches more systematic and less error prone, but this does not mean that design errors cannot occur. The automatic verification tools can be used to explore the entire state space with all possible interleaving of events to verify the correctness of the state machine.

3.2.1 State Machine Limitations

The pure state machine model does have several limitations, which may explain its limited use in the operating system community outside of network protocols and distributed shared memory protocols. Theses limitations include difficulty in integrating a state machine into a thread-based operating system, state explosion, and limited support for concurrency.

3.2.1.1 Operating System Integration

A state machine is most effective as a self-contained system, such as a network protocol stack, because its model doesn't mesh well with threads. Synchronous operating system functions that block the current thread wreak havoc with a state machine because a new event will not be handled until the current handler completes; if the handler blocks, so does the state machine. This not only limits concurrency, but also leads to deadlock if the blocked operation won't complete until the state machine processes another event. Wrapping all synchronous operations with stub functions that create a new thread to invoke the operation, leaving the original thread to complete the handler, will work if the parameters to the operation do not include addresses on the original thread's stack, but it requires writing stubs for all such operations and introduces overhead to create a new thread.

Interaction with threads is also a problem when the external environment needs to invoke a cache operation. To do so, the external thread must generate an event for the state machine and wait for the state machine to reach a state that indicates the completion of the operation. For example, to read a block from the cache the operating system must generate the equivalent of a "block read" event and wait for the state machine to enter a state that indicates that the read is complete. Wrapper functions can be used to generate the event and block the thread until the operation is complete, the overhead of which may affect performance.

3.2.1.2 State Explosion

A state machine transitions into a new state based only on the previous state and the event that occurred; data can be stored in auxiliary data structures, but only if that data does not affect the state transitions. For example, whether or not a block is in the cache must be encoded in the state machine's states, but the actual data contained in the block need not be. This is a severe limitation, as a complex piece of software such as a cache is likely to have too many states to be practical. Part of the problem is concurrency; for example, a cache that provides multiple readers/single writer access to data must keep track of the current number of readers. Encoding this in the set of states is infeasible.

The state machine's model of concurrency also leads to many *intermediate* states. An intermediate state is one in which the state machine waits for an event that represents the completion of an external operation, but also handles other events in the meantime. It isn't always possible for the state machine to transition to the same intermediate state when waiting for the same event; sometimes the processing done by the event handler depends on the previous state of the state machine. Since the execution context and history are entirely encoded in the (current) intermediate state, different intermediate states must be used if different handling must be done for the same event. An example is waiting for the server's reply to a fetch request when handling a cache miss (Figure 1). Different actions are taken when the reply arrives,

depending on whether a read is in progress or a write, requiring two intermediate states instead of one.

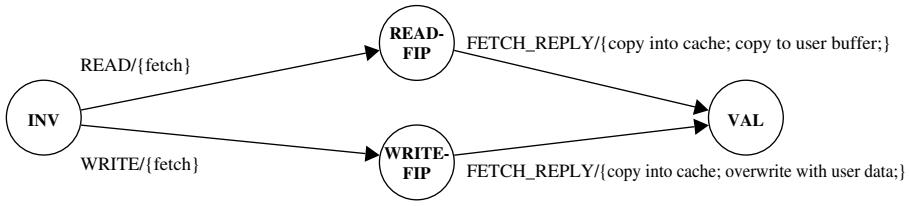

Figure 1 : **State Machine With Intermediate States to Wait for Fetch Replies** A labelled circle represents a state. An arrow represents a transition from one state to the next labelled with the event that triggers the transition and actions taken by the state machine in braces {}. The intermediate states, READ-FIP and WRITE-FIP, are used to wait for replies for fetches issued by the READ and WRITE event processing, respectively. The READ and WRITE processing cannot use the same intermediate state to wait for the fetch reply because different processing is performed when the reply arrives.

Complex data relationships, such as aggregate objects, also lead to a state explosion. In a file block cache, a file object is an aggregation of block objects, some of which are in the cache and some of which are not. The states of the file object are therefore the cross-product of the individual block states, which can easily lead to too many states in the state machine. In addition, it may be necessary to encode the number of blocks in the file in the file's state, exacerbating the problem.

3.2.1.3 Concurrency

It is easy to implement multiple concurrent computations using threads, but difficult to do so using a state machine. With threads a computation consists of the processing done by the thread, perhaps interrupted and resumed if the thread blocks. Private data needed by the computation are stored on the thread's stack. In the state machine model a computation consists of a sequence of event handlers that achieve the same result, separated by states in which the state machine waits for a particular event(s) to occur so that the computation can resume. For example, a read on a file cache may require the state machine to interact extensively with the external environment to fetch the data to be read, replace data from the cache, etc., each time pausing in an intermediate state to wait for an event. This makes it difficult for a state machine to implement concurrent computations.

The biggest problem is that the execution context and history must be encoded in the current state. A handler can use local variables to hold data local to the computation, but once the handler completes, these data are lost; if a subsequent event handler needs these data they must be encoded in the current state. In the example above, a read event on a block in the invalid state allocates a cache block and initiates a fetch to the server. When the fetch reply arrives, the data are copied into the previously allocated buffer; the buffer information must be encoded in the **READ-FIP** state for use by the next handler. This severely limits concurrency -- it may be possible to encode the context of a single fetch in the current state, but it isn't possible to encode the contexts of an arbitrary number of concurrent fetches.

3.3 Teapot

These problems with the pure state machine model have lead to the development of hybrid models, of which Teapot [Chandra97] is an example. Teapot is a domain-specific language for using state machines to implement coherency protocols. A Teapot specification defines the protocol's states, messages (events), and message handlers. A default handler can be specified to handle unexpected messages, such as queueing the message. Teapot generates C code for execution, as well as Murphi code for the Murphi automatic verification system [Dill92] for verifying the correctness of the protocol.

Teapot uses continuations to mitigate the state explosion and concurrency problems. A handler can encapsulate its current context in a continuation via its `suspend` mechanism. When `suspend` is invoked, any of the handler's local variables that will be used after `suspend` returns are stored in a continuation. The state machine then transitions to a specified *subroutine* state, which is a type of state that can only be entered via `suspend`. Subroutine states can be passed parameters which are accessible to all handlers in the subroutine state. When an event occurs in the subroutine

state, its handler is invoked with the continuation as one of the parameters; the handler can call `resume` and pass it the continuation to resume the suspended handler.

Teapot's use of continuations eliminates the need for separate **READ-FIP** and **WRITE-FIP** states. Instead, only a single state is needed, which simply resumes the suspended handler when a reply arrives. The resumed read or write handler then processes the reply event appropriately. This can greatly reduce the number of states needed, as the entire execution context need not be encapsulated in the current state.

Although continuations avoid saving a handler's context in the current state and event, Teapot's mechanism has several limitations. First, Teapot has limited support for multiple suspended handlers. It lacks the selection mechanism that is needed to match a fetch reply event with the correct continuation and other parameters. Teapot always passes continuations and parameters to the handlers in LIFO order, which may not be the order in which fetch reply events occur. In addition, continuations only store the local variables of the current stack frame that will be used when the `suspend` returns. This prevents `suspend` from being called inside a procedure call, limiting its usefulness.

4 Multi-Threaded State Machines

The limitations of both threads and state machines as models for implementing caches suggests a hybrid model, one that combines the benefits of each. Teapot is such a hybrid model, incorporating continuations into the state model, but it has its own limitations. To address these limitations we have developed an infrastructure for designing and implementing caches based on *multi-threaded state machines*. This model uses multiple execution contexts, auxiliary data structures, and hierarchal state machines to address the limitations of the existing models. Multiple execution contexts allow the notion of a computation to be retained: the execution of a series of one or more handlers to process the events that are received as part of the computation. Each computation's local data are stored in a separate context, so that each handler has access both to the state machine's shared data and the data local to the computation. This also allows a state to be defined such that its event processing does not depend on the previous state; handlers that wait for the same event and perform the same processing can use the same intermediate state to wait for the event.

In addition, multiple contexts simplify support for concurrent computations and reduce the state space they require. A separate context can be used to store each concurrent computation's local data, and auxiliary data structures can be used to store shared information such as the number and type of concurrent computations. For example, a file cache that supports multiple readers or a single writer may have a read-in-progress state to synchronize the readers and writers: events related to the existing readers or that signal the arrival of another reader are processed; events that signal the arrival of a writer are deferred until the state machine is out of the read-in-progress state. An auxiliary counter can be used to keep track of the number of readers, and used to transition the state machine out of the read-in-progress state when the number of readers drops to zero.

Finally, encoding all possible states of an aggregate object in the states of the state machine may be infeasible. For example, a file cache that supports concurrent block accesses must keep track of the state of each block. One possibility is to use auxiliary data structures to keep track of all the states of the blocks. An event related to a particular block is only processed if the auxiliary data structure indicates that the block is in an appropriate state to do so. This complicates the file cache state machine because the event handlers themselves need to mimic the functioning of the state machine: an event that cannot be handled must be queued until the corresponding block is in a state that can handle it.

Our approach is to use a hierarchy of state machines in which each block is controlled by its own state machine, which in turn is controlled by the file's state machine. Similarly, the files' state machines are controlled by the cache's state machine. This state machine hierarchy modularizes the control of the cache. The file's state machine performs operations on a block, such as reading it, by delivering an event to the block's state machines and waiting for an event to be delivered in reply when the processing is complete. The file's state machine only has to keep track of the number and type of block accesses in progress, rather than encode the states of its individual blocks.

4.1 Threads

We implemented multiple execution contexts in our model using threads, instead of continuations. There are two main reasons for doing so. First, threads simplify the task of saving and restoring the handler execution context because the

thread context and stack are automatically saved and restored by the underlying thread system when a thread blocks or resumes. The entire state of the thread is saved, not just local variables that will be used when the handler resumes. Thus a handler can be suspended at any level in the call stack, not just the first. This allows common processing that suspends the handler to be implemented as a procedure.

Second, multiple threads make it easier for a handler to invoke a synchronous operation, without blocking the entire state machine. If the thread running the handler blocks, another thread can be used to process subsequent events. This increases concurrency and prevents deadlock.

4.2 Multiple Contexts

Like the pure state machine, a multi-threaded state machine is defined by specifying its states, and a handler and transition for each event in each state. When an event is dispatched in the state machine, the handler specified for the event in the current state is invoked. The handler executes and the state machine transitions into a new state when the handler completes.

Unlike the pure state machine model, in which a handler runs to completion, a handler for the multi-threaded state machine can suspend execution and relinquish control of the state machine to allow other events to be processed. This is done via a `Suspend` mechanism similar to Teapot's, except that instead of creating a continuation the current thread is blocked and the state machine transitions into an intermediate state. The state machine then processes events as specified by the intermediate state's event handlers, perhaps creating more suspended contexts. Contexts are resumed via a resume mechanism, also similar to Teapot's.

Allowing multiple suspended contexts introduces a problem when an event handler resumes a suspended context. The issue is which context to resume if several contexts are waiting for the same event. In some situations it may not matter, but usually the event will be associated with a particular context. In the case of a fetch reply, the context to be resumed is the one that issued the fetch request. But how is the correct context determined? The reply must contain information that allows it to be matched with its request, whether the cache is implemented as threads or a state machine; this information might be a file ID and block ID, sequence number, RPC ID, etc. Tagging each context with this identifying information allows the correct context to be selected when the reply arrives.

In our system, the correct context is selected by comparing the contents of the reply event with *attributes* specified when the context suspended. These attributes consist of name/value pairs whose scope is private to the context. State-machine-specific code in the event dispatcher extracts the appropriate attribute values from the event, and compares these values with the attributes specified in the suspend. If the attribute values match, the context is selected. This selection is done in the event dispatching, before an event handler is invoked, and the context passed as an implicit parameter to the handler. When the handler calls `Resume`, the implicit context is resumed. The handler also has access to the attributes specified when the context suspended.

There are several issues related to the semantics of selection attributes. First, the attributes need to be private to each context because different contexts necessarily have different selection attributes. Second, no ordering can be enforced on the selection process as the events that will resume the contexts can occur in any order. Third, while in the intermediate state the state machine may process an event that causes it to transition into yet another state. This new state may process different events than the original intermediate state, may have different handlers for events, and may use different attributes as selection criteria. For this reason, the name space for selection attributes is scoped by the context, but not the current state. Handlers for the same event in different states must use the same attribute names to identify the criteria for selecting a context.

It is possible for an event to arrive whose attributes do not match any of the suspended contexts'. This can happen, for example, when a duplicate message arrives on the network, multiple replies arrive in response to a multicast request, or the cache protocol simply has an error. In these cases, no context is selected and a new context is created to run the handler.

4.3 Intermediate Handlers

Sometimes it is convenient for an intermediate state's handlers to perform some of the processing when an event occurs, rather than simply resuming the associated suspended context. First, if the state is shared by computations that wait for the same event and perform common processing on that event, then localizing the common processing in the handler of the shared state reduces the amount of code and simplifies the maintenance and debugging - the same benefits as procedures. Second, the state machine may have changed states while the computation is suspended, and the event processing may be different for the different states. If all processing were done in the suspended handler, this handler would have to examine the current state of the machine and perform the appropriate processing. By having the event handler perform the processing this conditional check of the state machine's state is avoided.

An event handler, however, may need information from the suspended handler to process the event, such as the cache buffer information to process a fetch reply. This information cannot simply be passed as parameters to the intermediate state's handlers because there is no guarantee that the state machine won't first transition into another state that has a different set of parameters. This is similar to the problem with selection criteria, and is handled in the same fashion. Information that is needed by the intermediate state handlers is passed by the suspended handler in its attributes. When an event occurs and a suspended context selected, the intermediate state's event handler has access to these attributes, and thus the desired information. Because attributes are global to a context, this solution works even if the state machine changes from the intermediate state to another state.

4.4 Synchronous Operations

A handler can invoke an operation that may block its thread by calling `SyncSuspend`. This routine simultaneously invokes the blocking operation and transitions the state machine into the specified intermediate state. When the blocking operation returns a "return" event is generated and dispatched using the handler thread.

4.5 Verification

One of the benefits of the finite state machine programming model is that it lends itself to automatic verification. Teapot, for example, produces output that can be verified by the Murphi [Dill92] verification system. Murphi performs an exhaustive search of the state machine's state-space, and detects deadlock. Researchers have reported on the usefulness of Murphi in verifying protocols specified in Teapot [Chandra97, Wang98].

Murphi's input is a program that implements the state machine, and consists of data types, global variables, procedures, a set of rules, a set of invariants, and the initial values for the global variables (the machine's initial state). The values of the global variables as the Murphi program runs defines the state space of the corresponding machine. A rule consists of a boolean expression and a sequence of actions; when the boolean becomes true the actions are performed. Murphi executes by repeatedly selecting a rule whose expression is true and performing its actions; after the actions have been performed Murphi verifies that all the invariants are true. Teapot generates Murphi input from its own input that uses the rules and invariants to search the state machine's state space and verify that deadlock is not possible.

Murphi expects the execution of a rule's actions to be deterministic -- if the same rule is applied in the same state, the state machine ends up in the same next state. Any data used by a handler to determine the next state of the machine must be contained in the global variables, so that Murphi knows that they are part of the machine's state, which in turn makes the handler deterministic.

5 Prototype

Our prototype of an infrastructure for developing multi-threaded state machines is implemented in C on the Linux operating system. The facilities it provides consist of several types of objects which implement the run-time support for the state machine and can be customized by the cache designer.

In this section, unless otherwise noted, we use the term *client* to refer to an external thread that dispatches an event to the state machine. This may be the network delivering a message, the file system accessing the cache, or dedicated threads

used by the cache itself.

5.1 Objects

The prototype defines five types of objects that the cache designer uses to implement a cache machine: **Event**, **Attribute**, **State**, **Protocol**, and **StateMachine** objects.

Each event that is delivered to a state machine is represented by an **Event** object. The **Event** object contains information about the event, such as the event type, the state machine to which it is dispatched, and other event-specific data. The **Event** object also contains a semaphore used to block the client when it must wait for processing to complete. A handler uses the **Event** object's `EventUnblock` method to unblock the client when processing is complete.

An **Attributes** object holds a computation's attributes. It consists of a variable-size array of name/value pairs, where the name is an integer and the value is of type **void ***. The **Attributes** object provides methods to create attributes and access their values.

Each state in a state machine is represented by a **State** object. The **State** object consists of two arrays of function pointers, one containing selection functions for the state and the other event handlers. The function pointers are indexed by event type; if a selection function is specified it is invoked to find a suspended context (if one exists), and if a handler is specified it is invoked, otherwise the event is queued.

A **Protocol** object holds all of the **State** objects for a state machine. The same **Protocol** object is shared by all instances of the same type of state machine; they all have the same states and use the same set of attributes. The number of attributes and their values are specific to the state machine, so the **Protocol** object defines methods that must be customized by the designer to create and delete an **Attribute** object for that type of state machine.

Finally, each instance of a state machine is represented by a **StateMachine** object. It contains references to the state machine's **Protocol**, the **State** object for the current state, the **Event** object for the current event being dispatched, and the **Attributes** object for the current computation; a state machine ID; a queue of pending events; a set of suspended computations; and information on the current computation. A state machine may require auxiliary data that are shared by the handlers, e.g., the cached data or a counter for the number of fetches that are in progress. A customized state machine is defined by defining an object that contains a **StateMachine** object as well as any other data fields that are needed.

5.2 Threads

The infrastructure also implements a **Thread** object that is used internally by the run-time system, but isn't directly used by cache designers. **Thread** objects are implemented using Linux threads, which provide a PThreads interface to kernel-level threads. The run-time system maintains a pool of dedicated threads that is shared among all state machines, and avoids the overhead of creating and destroying a thread each time an event is dispatched.

5.3 Dispatching Events

A client uses the **StateMachine** object's `StateMachDispatch` method to dispatch an event to the state machine. The `StateMachDispatch` function invokes the selection function for the event (if one has been specified) to select a suspended computation, and if one is found its thread is used to execute the handler; otherwise, a new computation and a new thread are used. The handler can run to completion, call `Suspend` to suspend itself, or `Resume` (or `Abort`) the suspended handler if one exists. The handler can also `Defer` the event; it is automatically re-dispatched when the state machine enters a new state.

Using a suspended computation's thread to execute the event handler avoids a context switch, but it does require selecting the suspended computation before the handler is invoked. This increases the dispatcher's space requirement (to store the selection function pointers) and complexity. On the other hand, performing the selection in the handler after it has been invoked not only incurs a context switch, but *nested suspends* (when the handler itself suspends before resuming the suspended handler) are a problem. Each `Suspend` blocks another thread, and each `Resume` requires a context

switch.

Another optimization is to use the client that delivers the event to execute the event handler. This can only be done if the client blocks upon dispatching the event and waits for processing to complete, and it introduces a complication that occurs if the thread is unblocked while the data on its stack are still in use by the state machine. For example, in a write-back cache writes return before the data have been written to the server; this usually requires making a copy of the data. This optimization can be supported via a flag for each handler that specifies whether or not the client can execute the handler directly. This is of limited usefulness, however, because references to the data on the thread's stack may be used by several handlers in sequence; it is up to the designer to determine that the thread will not be unblocked until all such references are gone. This limits its usefulness and we are currently investigating a better solution to the problem.

5.4 Suspending, Resuming, and Aborting Handlers

The `Suspend` mechanism suspends the current handler and puts the state machine into a specified intermediate state. The state of the suspended handler is stored in a **StateMachSuspend** object. Nested suspends are handled by storing the reference to the previous **StateMachSuspend** object on the thread's stack before suspending the handler. A handler uses `Resume` to resume a suspended handler.

The `SyncSuspend` is similar to `Suspend` except a procedure and parameters are also specified. `SyncSuspend` uses the handler thread to invoke the procedure, which may block the thread. It is possible for new events to change the state machine's state before the procedure returns; when it does, the handler can only be resumed if the machine is in a state that can handle it. For this reason, a return event is created and dispatched like any other event when the procedure returns.

The `Abort` mechanism aborts a suspended handler. All nested suspends are aborted, not just the current suspend; this allows a computation with a deep call chain to be aborted. A handler is aborted when it is no longer needed and will never be resumed. One complication is how to abort a `SyncSuspend` before the procedure returns. If the procedure cannot be aborted, `Abort` only flags the computation and aborts it when the procedure returns. This may cause some problems with resource management because the thread and other resources are still in use even though the external environment considers the computation aborted. Whether supporting abort will cause a problem is left to the discretion of the designer.

5.5 Saving Attributes

A handler that calls `Suspend` provides the attributes for the selection functions. We currently do not check, via compiler analysis or run-timing checking, that the attributes provided match the attributes needed. If the state machine stays in the intermediate state then static checking can be done. This is done in Teapot, in which parameters are specified for a subroutine state, and its compiler performs static checking. Unfortunately, the arrival of other events may cause the state to change. The attributes used by the handlers in this new state may be supersets or subsets of those used by the original state's handlers. Verifying that the proper attributes exist requires finding all states reachable from the original intermediate state, and ensuring that the union of the attributes they use are specified in the call to `Suspend`. We currently rely on the programmer to perform this analysis.

5.6 An Example

We implemented a simple file system to demonstrate the utility of the multi-threaded state machine model, especially with respect to supporting concurrent computations and complex data, and reducing intermediate states. The file system supports file open, close, read and write operations. It consists of a user-level file service that uses a UNIX file system for storage support, and a client-side file system that caches the file data in a multiple-reader/single-writer write-through cache. This example serves only as a proof of concept, so the clients actually reside on the same machine as the file server and communicate with the file server using named-pipes.

The client file cache is built as a hierarchy of state machines in which each cached file and each cached file block is represented by its own state machine. Stub functions are provided for the state machines that turn cache operations such as

reading data into event dispatches on the appropriate state machines.

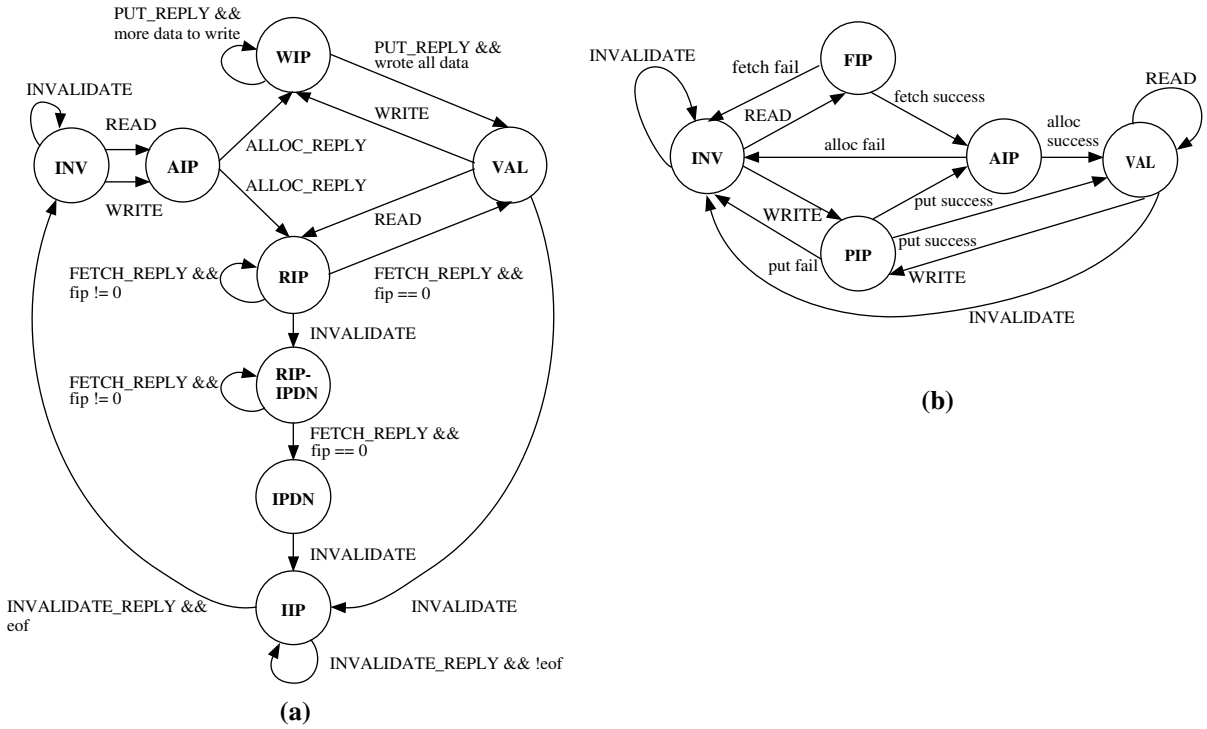

Figure 2 (a) **State Diagram for File Cache Coherency**
(b) **State Diagram for Block Cache Coherency Protocol**

The state diagram for the file cache state machine is shown in Figure 2a. The state machine accepts READ, WRITE, INVALIDATE for reading, writing, and invalidating the cached object. It also accepts ALLOC_REPLY, FETCH_REPLY, PUT _REPLY and INVALIDATE_REPLY events which are reply events for allocation of a cache object, block fetch request, block put request, and block invalidate request, respectively.

5.6.1 Intermediate States

The allocation of a file object demonstrates the sharing of intermediate states. If a state machine does not exist for the new file to be cached, one is created with an initial state of **INV** (**INV**alid). The memory for the cache object itself is not allocated when the state machine is created; it is deferred until a READ or WRITE event is dispatched into the state machine. The handler for the READ or WRITE event in the **INV** state issues an allocation to allocate memory for the object via `SyncSuspend`, suspends the handler, and transitions to the **AIP (A**llocation **I**n **P**rogress**)** state to wait for event that confirms the allocation. Once the memory is allocated, the suspended handler is resumed, which continues the read or write processing. Without support for multiple contexts, a separate allocation state would be needed for the READ and WRITE transitions from the **INV** state because of the different processing they do.

The state machine for a file in the cache is in the **VAL (VAL**id**)** state. This state means the file object is valid, not that all the file's blocks are in the block cache. READ and WRITE events are handled similarly in the **VAL** and **INV** states, except that the cache object is not allocated in the former. In the implementation, a `CacheRead` and `CacheWrite` procedure is used to process the READ and WRITE events in both the **VAL** and **INV** states. Both of these procedures call `SyncSuspend` to wait for FETCH_REPLY and PUT_REPLY events; the use of threads instead of continuations allows `Suspend` and `SyncSuspend` to be called from inside procedures.

5.6.2 Concurrent Computations

The usefulness of concurrent computations is best demonstrated by READ event processing. In general, READ is processed by fetching the data from the block cache one block at a time. The `CacheRead` procedure issues the fetch

request via `SyncSuspend`, transitions to the **RIP (R**ead **I**n **P**rogress**)** state and suspends to wait for the fetch reply. The FETCH_REPLY handler resumes the suspended handler which continues executing `CacheRead`. `CacheRead` issues the fetch request for the next block and repeats the process.

Because the cache supports multiple readers, the **RIP** state must also handle READ events. A READ event starts a new computation to execute the handler, which in turn invokes the `CacheRead` to process the read. The state machine only transitions out of the **RIP** states after all READs have been processed. A counter (**fip**), which is global to all the computations in the state machine, is used to keep track of the number of fetches that are in progress, and only transitions out of the **RIP** state when the fip counter reaches zero.

The **RIP** state also demonstrates how a concurrent computation can cause the state machine to transition to a different state while other computations are currently suspended in the current state. When an invalidation occurs while reads are in progress, the file is only invalidated after all reads have finished. No new reads are accepted in the meantime. As a result, the **RIP** state handles INVALIDATE events, the handler for which defers the event and changes the state machine to the **RIP-IPND** (**R**ead **I**n **P**rogress-**I**nvalidate **P**e**ND**ing) state. **RIP-IPND** defers READ events, ignores new INVALIDATE events, and handles FETCH_REPLY events so that the reads in progress can complete.

Once all reads have completed, the state machine changes to the **IPND** (**I**nvalidate **P**e**ND**ing) state which handles any INVALIDATE event in the queue. Invalidation requests are issued to the block cache to invalidate each block, and the **IIP** (**I**nvalidate **I**n **P**rogress) state is used to wait for INVALIDATE_REPLY events.

Since the cache only supports single-writer semantics, the processing for a WRITE event is much simpler than for a READ, as no concurrent computations are started while a write is in progress. A WRITE is handled by delivering PUT requests to the block caches, one block at a time, for each block to be written. The WRITE handler delivers a PUT request via `SyncSuspend`, transitions to the **WIP (W**rite **I**n **P**rogress**)** state, and suspends to wait for a PUT_REPLY. The state machine remains in the **WIP** state until the last PUT is processed. The write processing is also simplified by the fact that the file cache does not send data to the server directly; this is handled by the block cache.

5.6.3 Block Cache

The state diagram for the block cache state machine is shown in Figure 2b. There are three items of note. First, the block cache supports write-through semantics. On a WRITE event, the block cache first writes the data to the server, and only updates the cache buffer after the data are written to the file server successfully. Second, cache buffer allocation is only performed after data have been read from (for the READ event) or written to the server (for the WRITE event) successfully, and may require replacing another cached object to acquire the memory. This means the memory for the state machine must be acquired from preallocated or dynamically allocated memory, not from replacing another cached object. Finally, as in the file cache state machine, the **AIP** state is shared, which is not possible without multiple contexts.

6 Other Related Work

PCS [Uehara96] is a framework for customizing coherency protocols for distributed file caches in the Lucas file system. It provides a framework and specification language for writing coherency protocols. Protocols use `select` and `recv` operations to specify what messages can be handled, and the framework dispatches only those messages and automatically queues other messages, simplifying the protocol description. Protocols are not specified as state machines in PCS, however. Instead, PCS uses a sequential control flow model in which a protocol consists of a large `select` statement, with handlers for each of the messages handled. These handlers may in turn call `select`, which corresponds to changing the state of the state machine. PCS only supports a single context which it saves in a continuation when a handler blocks on `select` and `recv` calls. This leads to problems similar to Teapot's, and makes it difficult to invoke a synchronous operation inside a message handler. Concurrency occurs when a message handler issues its own `select` and processes another message; this nesting of suspend statements makes the protocol's flow-of-control difficult to reason about and manage.

Statecharts [Harel87, CHSM94] are structured state diagrams used to describe large complex reactive systems. Like our work, it is a hybrid state machine model designed to address the state explosion problem and limited support for con-

currency in the state machine model. Statecharts add depth and orthogonality to state machines by supporting the notion of a group state, which is a state composed from smaller state machines. States that have common transitions to a state can be grouped into an *OR-group* state and replacing the transitions from the individual state with a single transition from the group state, thereby reducing the number of transitions in the state diagram. A system with concurrent components that operate independently can be defined by decomposing the system into state machines for each component and grouped into an *AND-group* state -- all the component state machines in an AND-group state must be active when the group state is active. This eliminates the state explosion problem that results from having the cross-product of the components' states in traditional state machines. The model of concurrency that statecharts provide is for concurrent components that are relatively independent -- they do not share states and only communicate via events. Statecharts do not address the state explosion problem caused by intermediate states, and do not address concurrent operations on the same object as we do. Statecharts are complementary to our work, and it seems possible to add multiple threads support to statecharts to reduce the state space further. Statecharts, however, are much more general than is needed for building caches.

Esterel [Berry92] is an imperative synchronous language for implementing complex reactive systems. Esterel, like statecharts, addresses the limitation of modeling complex reactive systems with concurrent subsystems. Like statecharts, Esterel does not address the issues of concurrent computation as we do.

7 Conclusion

A distributed file cache is a complicated software system. It must deal with concurrent data accesses from application program, asynchronous communication with the file server, and complex relationships between the data in the cache. Cache coherency protocols can be complex, providing acceptable coherency with a minimum of overhead. Standard programming models for designing and developing caches include threads and finite state machines, both of which have deficiencies. To address these deficiencies we have developed a *multi-threaded state machine* programming model. This hybrid model uses state machines to control shared data, while threads enable individual computations that have private data. Threads also reduce the number of intermediate states, and simplify the integration of a cache based on state machines with an operating system based on threads. We have implemented a prototype of our infrastructure and used it to implement a write-through file block cache. This exercise demonstrated the usefulness of multi-threaded state machines in cache design and implementation.

References

[Berry92] G. Berry, and G. Gonithier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2): 87-152, November 1992.

[Chandra96] S. Chandra, B. Richards, and J. R. Larus. Teapot: Language Support for Writing Memory Coherency Protocols. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 237-248, May 1996.

[Chandra97] S. Chandra, M. Dahlin, B. Richards, R.Y. Wang, T.E. Anderson, and J.R. Larus. Experience with a Language for Writing Coherence Protocols. In P*roceedings of USENIX Conference on Domain-Specific Languages*, pages 51-66, October 1997.

[Dill92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522-525, October 1992.

[Harel87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231-274, June, 1987

[Lucas94] P. J. Lucas. An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines. Technical Report UIUCDCS-R-94-1868, Department of Computer Science, Univer-

sity of Illinois, Urbana-Champaign, August 1994.

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119-130, June 1985.

[Wang98] R. Y. Wang, T. E. Anderson, and M. D. Dahlin. Experience with a Distributed File System Implementation. http://www.cs.berkeley.edu/~rywang/papers/spe98.ps, 1998.

[Uehara96] K. Uehara, S. Inohara, H. Miyazawa, K. Yamamoto, and T. Masuda. A Framework for Customizing Coherence Protocols of Distributed File Caches. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 83-90, May 1996.