Experience with a Language for Writing Coherence Protocols

Satish Chandra[1], Michael Dahlin[2], Bradley Richards[3], Randolph Y. Wang[4]
Thomas E. Anderson[4] and James R. Larus[1]

[1]*University of Wisconsin, Madison*
[2]*University of Texas, Austin*
[3]*Vassar College*
[4]*University of California, Berkeley*

Abstract. In this paper we describe our experience with *Teapot* [7], a domain-specific language for addressing the *cache coherence problem*. The cache coherence problem arises when parallel and distributed computing systems make local replicas of shared data for reasons of scalability and performance. In both distributed shared memory systems and distributed file systems, a *coherence protocol* maintains agreement among the replicated copies when the underlying data are modified by programs running on the system. Unfortunately, cache coherence protocols are notoriously difficult to implement, debug, and maintain. Furthermore, the details of the protocols depend on the requirements of the system under consideration and are highly varied. This paper presents case studies detailing the successes and shortcomings of using Teapot for writing coherence protocols in two distinct systems. The first system, *loosely coherent memory* (LCM) [15], implements a particular flavor of distributed shared memory suitable for data-parallel programming. The second system, the *xFS distributed file system* [9], implements a high-performance, serverless file system.

Our overall experience with using Teapot has been positive. In particular, Teapot's language features resulted in considerable simplifications in the protocol code for both systems. Furthermore, Teapot's close coupling between implementation and formal verification allowed us to achieve much higher confidence in our protocol implementations than had previously been possible, reducing the time needed to build the protocols. By using Teapot to solve real problems in complex systems, we also discovered several shortcomings of the Teapot design. Most noticeably, we found Teapot lacking in support for multithreaded environments, for expressing actions that transcend several cache blocks, and for blocking system calls. We conclude that domain-specific languages can be valuable in the specific problem domain of cache coherence. Drawing on our experience, we also provide guidelines for domain-specific languages in the broader context of systems software.

1 Introduction

Cache coherence engines are key components in several parallel and distributed computing systems. Coherence issues arise whenever distributed systems make local replicas of shared information for reasons of performance or availability (or both) because the systems must keep those replicas current while they modify the shared information. Thus, distributed shared memory systems [6,14], distributed file systems [19, 9], and high-performance client-server database systems [12] all implement cache coherence protocols. Coherence in web caching is also a current research topic in the distributed systems community [18].

Tools that facilitate the implementation of cache coherence protocols are important for two reasons. First, coherence protocols, while ubiquitous, show a great deal of variety because the protocol for a particular system is closely linked to its sharing semantics and performance goals. For example, different distributed shared memory systems provide different memory consistency models [13], supporting different assumptions that application programs can make about the currency of cached values. Systems with similar sharing semantics can have vastly different protocols, implementing different algorithms for achieving the same task, albeit with different performance considerations. Thus, each system essentially needs a new coherence protocol. Second, and perhaps more importantly, cache coherence protocols represent complex, distributed algorithms that are difficult to reason about, often resulting in subtle race conditions that are difficult to debug via system testing. Furthermore, to our knowledge, most systems hitherto have not attempted a clear separation between the cache-coherence engine and other implementation details of the sys-

tem, such as fault management, low-level I/O, threads, synchronization, and network communication. It is not difficult to imagine the hazards of this approach. The implementor cannot reason about the coherence protocol in isolation from other details, and any modification she makes in the system can potentially impact the correctness of the coherence protocol, a debugging nightmare. Experimentation with newer protocols is a perilous proposition at best.

Teapot is a protocol writing environment that provides two significant improvements over writing ad-hoc C code. First, it is a domain-specific language specifically designed to write coherence protocols: it forces a protocol programmer to think about the logical structure of a protocol, independent of the other entanglements of a system, and language features of Teapot facilitate the task of expressing the control structure commonly found in coherence protocols. Second, Teapot facilitates automatic verification of protocols because it not only translates Teapot protocols into executable C code, it also generates input code for MurΦ, an automatic verification system from Stanford[10]. MurΦ can then be used to detect violations of invariants in a modest amount of verification time. For example, our system might report a stylized trace of a sequence of events that would cause a deadlock. A protocol can be run through a verification system prior to actual execution to detect possible error cases *without* having to manually rewrite the protocol in MurΦ's input language.

The Teapot work was originally undertaken to aid protocol programmers in the context of the Blizzard distributed shared memory system[24]. Blizzard exports a cache-coherence protocol programming interface to an application writer, so she can supply a coherence protocol that best suits the requirements of her application. Writing such protocols in C, without domain-specific tools, turned out to be a difficult task, fraught with problems of deadlocks, livelocks, core dumps, and most annoyingly, wrong answers. After few initial protocols (all variants of ordinary shared memory protocols) were successfully developed using Teapot, the Blizzard team at Wisconsin wrote several other, more complicated coherence protocols for their system. We report on one such protocol here. Subsequently, the xFS team at UC Berkeley adopted Teapot to write the coherence protocol of their distributed file system. As expected, these teams encountered several rough spots, because the original Teapot design did not anticipate all of the requirements of other protocols in the context of Blizzard, much less those arising in a distributed file system context.

This paper describes our experiences with using Teapot to implement the coherence engines in two distinct systems. In both systems, we found Teapot to be vastly superior to earlier efforts to implement the protocols using C without any domain-specific tools. The paper makes several contributions. First, it highlights the aspects of Teapot that proved successful across several protocols:

- *Domain-specific language constructs*, such as a state-centric control structure and continuations, simplified the protocol writing task.
- *Automatic protocol verification* using the MurΦ system improved system confidence and reduced testing time.

Perhaps more importantly, this paper also discusses shortcomings of the language that became apparent only when we attempted to develop protocols that were much more complicated than the simple protocol examples on which Teapot was originally tested. In particular, our experience indicates that improved support for multi-threaded environments, for protocol actions that affect multiple blocks, for local protocol actions that might block, and for automated verification test strategies would further ease the job of a protocol designer. Finally, the paper generalizes our experience to provide guidelines for future domain-specific languages for systems software.

The rest of the paper is organized as follows. Section 2 provides some basic background on cache coherence protocols and describes the implementation problems generally faced by protocol programmers. Section 3 introduces the language features in Teapot that address the difficulties presented in Section 2. Section 4 presents the case-study of LCM, and Section 5 presents the case-study of xFS. Section 6 describes some related work. Section 7 concludes the paper with implications for domain-specific languages for systems software.

2 Coherence Protocols and Complications

In systems with caching, read operations on shared data typically cache the value after fetching it from remote nodes, in the expectation that future read references will "hit" locally. Write operations on shared data must take steps—coherence actions—so readers with cached values do not continue to see the old value indefinitely. This section describes coherence protocols in more detail in the context of distributed shared-memory systems, though the issues discussed apply equally well to other contexts with appropriate changes in terminology.

Shared-memory systems can be implemented using a pair of mechanisms: access control and communication.

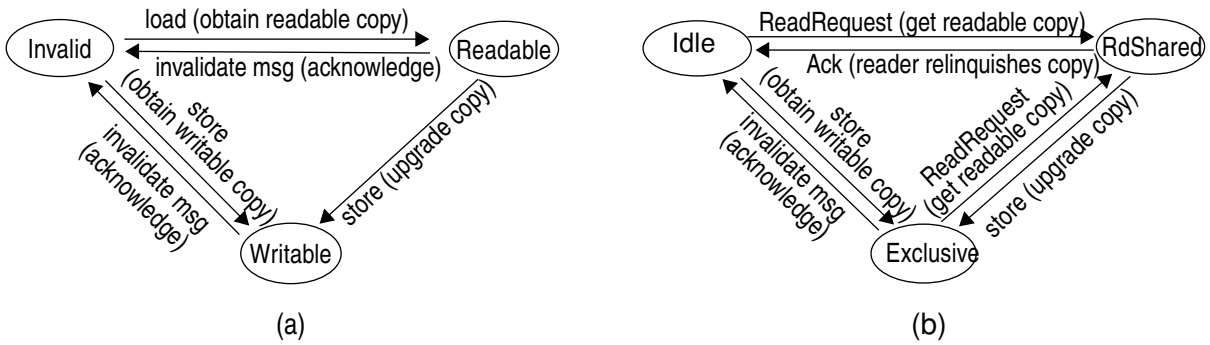

Figure 1: Idealized protocol state machine for (a) the non-home side, and (b) the home side. Transitions are labeled with causes and, in parentheses, actions.

Access control allows the system to declare which types of accesses to particular regions of memory are permitted. These permissions typically include—no access (*invalid*), reads only (*readonly*), and both reads and writes (*readwrite*). Performing an illegal access (for example, writing a *readonly* region) causes an *access fault* and invokes the coherence protocol. Communication allows a system to exchange control information and data between processors. The coherence protocol comes into play at an access fault. It must obtain a copy of the referenced data with appropriate access permissions and satisfy the access. Many protocols designate a *home node* that coordinates accesses to a particular range of memory addresses. The faulting processor sends a request to the home node for a copy of the required data, which responds with the data after updating its bookkeeping information. After receiving the response, the faulting processor typically caches the data so subsequent accesses will succeed without communication.

A common technique for ensuring coherence allows at most a single writer or multiple readers for any block of memory at a time. When the home receives a request for a writable copy of the block, it asks processors currently holding a readable copy to invalidate it, i.e. allow no further accesses. A writable copy can then be sent to the requestor. A cache coherence protocol specifies the actions taken by the home and caching processors in response to access faults and incoming messages. These actions are commonly captured by finite state machines, with transitions between protocol *states* occurring in response to faults and messages. Figure 1 shows sample state machines describing protocol actions for a caching processor and the corresponding home side. Both the home and caching processors associate a state with each memory block. At an access fault or upon a message arrival, the protocol engine consults the appropriate block's state to determine the correct action. Typical protocol actions involve sending messages and updating the state, the access permissions, and contents of a memory block. Home nodes also maintain a *directory*, a per-block data structure that usually keeps track of which processors have a readable copy, or which processor has an exclusive copy.

As an example, consider a (non-home) block that is initially in the *Invalid* state. A processor reading any address within the block causes an access fault, at which time the protocol is invoked. Its action is to send a request to the home node for a readable copy and await a response. Assuming no outstanding writable copy exists (the *Idle* state in Figure 1), the home responds with a readable copy and changes its state to *ReadShared*. The arrival of this message on the non-home side causes the protocol to copy the incoming data to memory and change the block's state to *Readable* (and access permissions are changed from *invalid* to *readonly*).

Unfortunately, specifying protocols is much more difficult than the simple three-state diagrams in Figure 1 would lead one to believe. The main difficulty is that, although the transitions shown *appear* to be atomic, many state changes in response to protocol events cannot be performed atomically. Consider the transition from the *Exclusive* state to the *ReadShared* state in Figure 1. Conceptually, when a request arrives in the *Exclusive* state for a readable copy of a block, the protocol must retrieve the exclusive copy from the previous owner and pass it along to the requestor. The protocol sends an invalidation request to the current block holder, and must await a response before proceeding. But, to avoid deadlock, protocol actions must run to completion and terminate. This requires that an intermediate state, *Excl-To-ReadShared*, be introduced. After sending the invalidation request, the protocol moves to the *Excl-To-ReadShared* state and relinquishes the processor. When the invalidation acknowledgment arrives in this intermediate state, the processor sends a response to the original requestor and completes the transition to *ReadShared*. A revised state diagram incorporating the required intermediate states is shown in Figure 2 (which is still far removed from a realistic protocol).

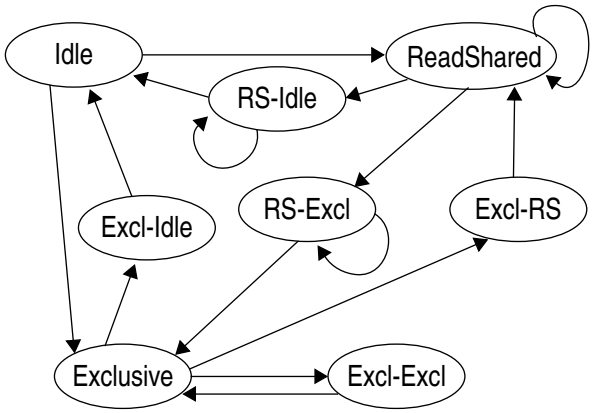

Figure 2: State machine (home side) with intermediate states necessary to avoid synchronous communication.

Introducing intermediate states increases the number of states a programmer has to think about. Furthermore, while in an intermediate state, messages other than the expected reply can arrive. For example, before the invalidation response arrives in the *Excl_To_ReadShared* state, another request for an exclusive copy could arrive from a different processor. A protocol designer must anticipate the arrival of such unsolicited messages and handle them in an appropriate manner. It may be tempting to not take such messages out of the network while they are not welcome: this, however, is not an option on most systems, because messages must constantly be drained out of the network to avoid deadlock in the network fabric [26].

Message reordering in the network adds to the woes of a protocol programmer. For example, processors may appear to request copies of cache blocks which they already have, if a read request message overtakes an invalidation acknowledgment message in the network. The protocol might have to await delayed messages before deciphering the situation and determining the correct action. Without machine assistance, anticipating all possible network reorderings is a very difficult task!

The traditional method of programming coherence state machines usually resorts to ad-hoc techniques: unexpected messages may be queued, they may be negatively acknowledged (nack'ed), or their presence may be marked by a "flag" variable. Additional flag variables are often used to track the out-of-order arrival of messages as well. These techniques invite protocol bugs. Queuing can easily lead to deadlocks; similarly, nack'ing can lead to livelocks or deadlocks. Flag variables are essentially extra protocol state—failing to update or test a flag at all the right places again leads to correctness problems. Moreover, protocols implemented in this style are very difficult to understand and modify.

The case studies presented in sections 4 and 5 show that all these complications were serious issues in the initial state machine versions of those protocols. In the next section, we highlight the features of Teapot that aid a protocol programmer.

3 Teapot

The Teapot language resembles Pascal with extensions for protocol programming support, but fewer built-in types. Space does not permit a complete description of the language; the reader is referred to the original paper [7] for further language details. The Teapot compiler can generate executable C code from a protocol specification, and can also translate it to code that can be fed to the MurΦ verification system[10].

3.1 Verification Support

The generated MurΦ code models the input protocol, but must be supplemented with code describing the system in which the protocol executes. We developed, by hand, routines implementing a typical network model and necessary support routines, as well as a typical *ruleset*. The ruleset drives the verification by generating all possible sequences of loads and stores to one or more simulated shared-memory addresses.

3.2 Teapot Example

A Teapot program consists of a set of states; each state specifies a set of message types and the actions to be taken on receipt of each message, should it arrive for a cache block in that state. We exhibit some of the features of Teapot using an example; The Teapot code in Figure 3 implements coherence actions for a block in the *Exclusive* state at the home node. Suppose the block receives the request message `GET_RO_REQ`, asking for a readable copy. The action code for this message first sends a `PUT_DATA_REQ` message to the current owner (note that the variable `info` is a pointer to the directory data structure). Next, it executes a `Suspend` statement. A `Suspend` statement is much like a "call-with-current-continuation" of functional programming languages. Syntactically, it takes a program label (`L`), and an intermediate state (`Home_Excl_To_Sh`) which it visits "in transition"; the label is passed as an argument to the intermediate state. Operationally, it saves the environment at the point it appears in a handler body and effectively puts the handler to sleep. This mechanism is used to provide a blocking primitive inside a handler,

```
1.  State Stache.Home_Exclusive{}
2.  Begin
3.      Message GET_RO_REQ(id:ID; Var info:INFO; src: NODE)
4.      Var
5.          itor : SHARER_LIST_ITOR;
6.          j : NODE;
7.      Begin
8.          Send(GetOwner(info), PUT_DATA_REQ, id);
9.          IncSharer(info, src);
10.         Suspend(L, SetState(info, Home_Excl_To_Sh{L}));
11.         -- send out a readable copy to all nodes that want a copy
12.         -- (more nodes might want a copy while you were waiting)
13.         Init(itor, info, NumSharers(info));
14.         While (Next(itor, j)) Do
15.             SendData(j, GET_RO_RESP, id, TPPI_Blk_No_Tag_Change);
16.         End;
17.     End;
18.     -- other messages ...
19.     Message DEFAULT(id:ID; Var info: INFO; src: NODE)
20.     Begin
21.         Error("Invalid message %s to Home_Exclusive",Msg_To_Str(MessageTag));
22.     End;
23. End;
```

Figure 3: Teapot example

which physically needs to relinquish the processor every time it is invoked.

What happens in the intermediate state? Figure 4 shows the Teapot code executed when a PUT_RW_RESP message arrives. The handler receives the up-to-date content of the cache block from the network, sets its own state to *ReadShared*, and executes a Resume statement. The Resume is the equivalent of a "throw" for a "call-with-current-continuation" of functional programming. Syntactically, it takes a continuation parameter (C) as an argument. (Note from line 1 in Figure 4 that the continuation variable C is a state parameter and is a part of the environment visible to all the message handlers in that state.) Operationally, it restarts the suspended handler immediately after the Suspend statement that called this intermediate state. Thus, after the Resume statement, GET_RO_RESP messages are sent to the set of requesters (see Figure 3 again, lines 13-16). Continuations in Teapot let us avoid having to manually decompose a handler into atomically executable pieces and sequencing them. Further advantages of the Suspend/Resume primitives are brought out in the case studies Teapot provides a mechanism for handling unexpected messages by queuing. It does not solve the problem of deadlocks directly, but facilitates deadlock detection via verification. In lines 10-13 of Figure 4, all messages not directly handled (DEFAULT) are queued for later execution—these messages are appropriately dispatched once the system moves out of a transient state. Teapot relies on a small amount of system-specific dispatch code to deliver incoming network messages and previously queued messages, based on a state lookup and the message tag. Note that the DEFAULT messages in Figure 3 flag an error because these messages cannot occur in a correctly functioning system.

```
1.  State Stache.Home_Excl_To_Sh{C:CONT}
2.  Begin
3.      Message PUT_DATA_RESP (id: ID; Var info: INFO; src: NODE)
4.      Begin
5.          RecvData(id, TPPI_Blk_Validate_RW, TPPI_Blk_Downgrade_RO);
6.          SetState(info, Home_RS{});
7.          Resume(C);
8.      End;
9.      -- other messages
10.     Message DEFAULT (id: ID; Var info: INFO; src: NODE)
11.     Begin
12.         Enqueue(MessageTag, id, info, src);
13.     End;
14. End;
```

Figure 4: Teapot example (cont'd)

4 LCM

.The Loosely Coherent Memory (*LCM*) [15] coherence protocol implements the semantics of the parallel programming language C** [16] faster than conservative, compiler-implemented approaches. C** is a large-grained data-parallel programming language based on C++ and provides a semantics in which parallel function invocations on aggregate data execute *simultaneously* and *instantaneously*, so conflicting data accesses are impossible. Processes can still collaborate to produce values via a rich set of reduction operations (including user-specified reductions), but the results of these reductions are not available until after all parallel function invocations complete. During a parallel computation, no function invocation can influence the state of another.

LCM helps implement C** by allowing protocol-level copies of shared data to develop at runtime and efficiently *reconciling* copies once all tasks have finished. The compiler uses LCM directives to identify memory accesses in parallel functions that can possibly conflict. At these references, LCM copies the memory block containing the accessed location and makes it private to the invocation. If multiple invocations modify the same location, LCM creates local copies for each invocation. These multiple writable copies preserve the semantics of C**, even though shared memory as a whole is no longer consistent. When the parallel call terminates, LCM reconciles multiple versions of a block to a single consistent value.

LCM provides consistent memory as a default and is similar in many respects to protocols providing sequentially consistent distributed shared memory such as DASH [17], Alewife [1], and Stache[23], but it differs in several key respects. Most importantly, LCM allows global memory to become temporarily inconsistent under program control. During these phases, a given data item may have different values on different processors, making correct management of shared data more difficult. Memory is returned to a globally consistent state by merging these distinct copies into a single value for each data item and ensuring that all processors see these new values. This requires coordination among all processors in the system and mixes computation (merge functions) with traditional protocol actions.

4.1 Initial Implementation

Our first LCM implementation effort was undertaken without the support of any formal methods or tools. The C code source of the Stache (ordinary shared memory) protocol was available to us, so we used it as a starting point and added extra LCM functionality as required. In retrospect, starting with Stache was an unfortunate decision. Stache, while a relatively simple protocol design, is still a large and complex piece of software. Adding LCM functionality required both that the behavior of existing protocol states be altered and that new states be added—a difficult proposition for the unaided programmer. Small changes in existing states (and the addition of a new states) often had far-reaching effects that were difficult to fully anticipate.

It took several months for a single graduate student, working full-time, to complete the basic protocol modifications, after which a debugging phase began. It took roughly as long to debug the modified protocol as it did to write it in the first place since the protocol was riddled with subtle timing-related bugs, the result of the unpredictable effects of our modifications. A suite of applications was used to debug the protocol—each application exercising a new set of path-specific bugs in LCM which had to be isolated, understood, and repaired. It often took days to identify infrequently-occurring bugs, and the resulting "fixes" often introduced new bugs. Even after the LCM protocol had achieved relative stability, user confidence in its correctness was low.

4.2 Teapot and LCM

An early version of the Teapot system was ready for testing as debugging of the hand-written LCM protocol was being completed, and LCM was reimplemented using Teapot to more thoroughly evaluate the system. The Teapot environment was a vast improvement over the hand-coded approach. We found two language features of Teapot particularly useful: the "state-centric" programming model, and the use of continuations to allow blocking operations in handler code.

In Teapot, one declares a protocol state, then lists the actions to be taken for the various messages that could arrive in that state. This contrasts with the "message-centric" approach taken in the handwritten protocol, where a single handler is written for each possible message, and a large conditional statement in its body selects the appropriate action based on the recipient block's state. Organizing the protocol by states instead of message type makes it easier to express and implement for several reasons. First, each handler is now a smaller unit of code. Instead of writing a large message handler that must behave correctly for a block in any state, a self-contained handler is written for each combination of message and block state. Second, programmers typically have a well-defined concept of how each state should behave, and grouping handlers by state instead of message type keeps related information close together. A state's behavior can be understood by scan-

```
1.  State LCM.Home_Excl {}
2.     ... other messages
3.     Message GET_RO_REQ (id: ID; Var info: INFO; src: NODE)
4.     Begin
5.        [...]
6.        If (SameNode(src, GetOwner(info))) Then
7.           Suspend(L, SetState(info, Home_Excl_To_Idle{L}));
8.           If (SameState(GetState(info), Home_Idle{})) Then
9.              SetState(info, Home_RS{});
10.             AccChg(id, TPPI_Blk_Downgrade_RO);
11.          Else
12.             If (InSharers(info, src)) Then
13.                Suspend(L2, SetState(info, Home_Await_PUT_ACCUM{L2}));
14.             Endif;
15.          Endif;
16.          [...]
17.       Else
18.          Send(GetOwner(info), PUT_DATA_REQ, id);
19.          Suspend(L1, SetState(info, Home_Excl_To_Sh{L1}));
20.          IncSharer(info, src);
21.       [...]
22.       Endif;
23.    [...]
24.    End;
```

Figure 5: Teapot handler code containing multiple `Suspend` statements

ning a set of consecutive handlers, instead of having to look through the entire protocol for actions relevant to a given state. This makes modification and debugging easier as well. Of course, in retrospect, we could have adopted a state-centric organization in the handwritten protocol, but the C language did not make the benefits of doing so immediately obvious while the Teapot system enforced a disciplined programming style that utilized the better design choice.

Teapot's continuations also made an enormous improvement in handler legibility. Even for handlers using a single `Suspend` statement, keeping the code on either side of the call in the same handler dramatically increased readability. Some handlers used as many as three `Suspend` statements, and therefore had to be split into multiple code fragments in the handwritten version. Figure 5 shows part of an LCM handler with three `Suspend` statements. Without continuations, this code would have been split into at least four distinct handlers making it much harder to write and debug. Teapot also allows dynamic nesting of continuations, a feature used numerous times during the specification of LCM. For example, the first `Suspend` in Figure 5 moves to the `Home_Excl_To_Idle` state, where other handlers (not shown) may suspend again to await delayed messages

Even with the cleaner design, we uncovered a total of 25 errors with Teapot's automatic verification tool. (Each error was fixed as soon as it was detected and understood, and the verification step was repeated.) Many of these were subtle bugs that were unlikely to occur often in practice, but were all the more dangerous as a result. Figure 6 illustrates an LCM bug that is representative of those found through verification. Both diagrams show messages being exchanged between a pair of processors, with time increasing from top to bottom. In each case, a preceding exchange of messages (not shown) has left the cache (non-home) side with the exclusive copy of a given coherence.block

In Figure 6a, the caching processor performs an LCM modification of the block, creating a version that is inconsistent with respect to other copies in the system. However, since the cache side held the exclusive copy at the time it performed the modification, it first sends a *copy* of the block home. This data can be used by the home to respond to requests for the block from other processors. The block is returned home via a `PUT_MOD` message when the cache side is finished. The second LCM modification then faults and requests the block back from the home.[1] Messages have been reordered in the network such that the first to appear at the home is the request for data. The home detects the reordering, since the requestor already *has* a copy of the block according to directory information. The correct action in this case is to await the `SHARE_DATA` message, then satisfy the request. The home leaves the block in the `Home_LCM` state to denote the fact that at least one processor has created its own version of the block.

1. This scenario arises frequently in applications where a given processor handles several of a set of parallel tasks consecutively.

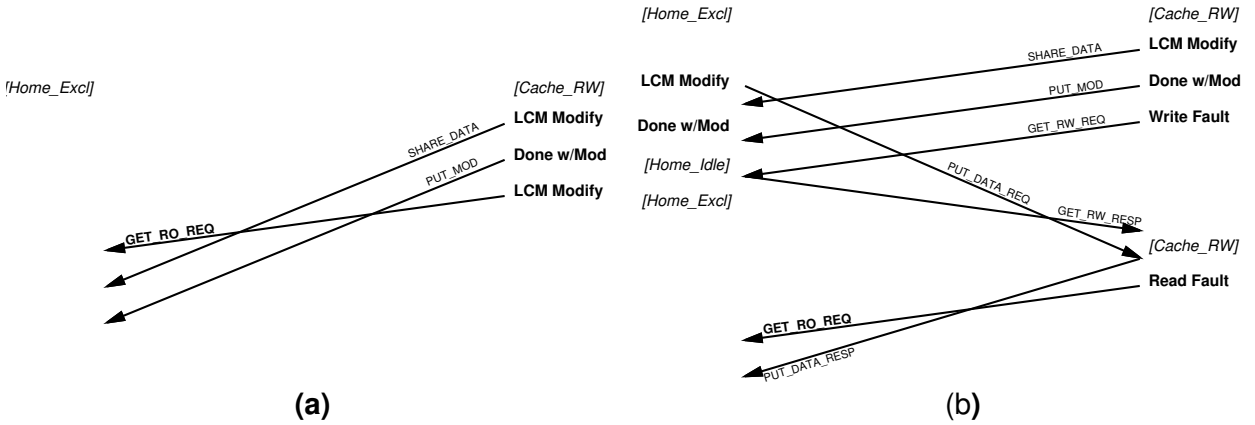

Figure 6: Two different scenarios in which a `GET_RO_REQ` arrives in state `Home_Exclusive`. The appropriate response to the message is different in each case.

Initially, we thought the arrival of the `GET_RO_REQ` in the `Home_Excl` state always implied the message reordering scenario in Figure 6a, and both the hand-written version of LCM and the first Teapot version encoded this assumption. Unfortunately, in the more complicated case shown in Figure 6b, this caused the protocol to respond incorrectly. The home should instead await the `PUT_DATA_RESP` message, transition to the `Home_Idle` state, and satisfy the request. Correcting the protocol is straightforward once the two scenarios have been identified, but it is unreasonable to expect an unaided programmer to have foreseen such a bug, due to the complexity of the cases involved. Enumerating all chains of protocol events and ensuring that they are properly handled is a job much better handled through verification.

Using Teapot, the new version of the LCM protocol was written, verified, and running applications in two weeks' time. Only one bug was uncovered during field testing of the new protocol, and it occurred in a simple support routine that was intentionally *not* simulated.[1] Also, because of Teapot, we were able to implement easily three variants of LCM: one that eagerly sends updates to consumers at the end of an LCM phase, another that manages multiple, distributed copies of some data as a performance optimization, and a version that incorporates both of these features.

4.3 Teapot Shortcomings

While Teapot made it significantly easier to get LCM written and working, it fell short of our needs in several respects. One significant obstacle is Teapot's inability to perform actions across a *set* of blocks. A message handler, for example, can only update the state of the block to which a message is directed. In LCM, action must periodically be taken across a collection of blocks. For example, during the reconciliation phase, a processor returns *all* modified blocks to their homes, where they are merged with copies from other processors. An event handler was written to carry out this flushing operation for a single block, but the handler must somehow be invoked for each block returned. As an application runs, the LCM protocol constructs a list of modified blocks that require flushing at the next reconciliation. This list is traversed when the reconciliation phase begins, and the appropriate event handler invoked on each block. Additional C code was written to traverse the list and invoke handlers in the executable version of the protocol, but this code is outside the scope of the Teapot protocol specification and therefore cannot be verified. The work around in Teapot was to structure the MurΦ ruleset so that, during a reconciliation, it invoked the handlers for each block in the list. This restructuring significantly increased the complexity of the ruleset and therefore the chances that it could contain an error.

Even without operations on sets of blocks the ruleset for LCM was already much more complicated than those for our previous protocols. Unlike Stache, where any arbitrary stream of interleaved loads and stores to shared memory must be handled, LCM only properly handles stylized sequences of loads and stores. There are distinct phases that all processors must agree to initiate, in which only certain access patterns are legal. Encoding this into a ruleset was a lengthy, complicated, and potentially error-prone process, and represented a significant fraction of the work required to implement LCM. It would be preferable to generate such rulesets automatically from a high-level description of a protocol's memory model, but we currently are unaware of any techniques for doing so.

1. The routine was deemed too simple to be hiding any bugs.

The last shortcoming was relatively minor. Teapot currently does not allow the testing of a pair of expressions for equality. There were several places in the protocol where pairs of states or node identifiers needed to be compared, and an external routine had to be written to perform these tests. Future releases of Teapot should extend the language such that comparisons can be done without resorting to external procedures.

5 xFS

xFS, a network file system described in several previous papers[2,9], is designed to eliminate all centralized bottlenecks and efficiently use all resources in a network of workstations. One of the most important features of xFS is its separation of data storage from data management. This separation, while offering superior performance and scalability compared to traditional file systems, also requires a more sophisticated cache coherence protocol. In addition, other aspects of the cluster file system environment—such as multi-level storage and reliability constraints—further complicate the system compared to more traditional DSM coherence protocols. Due to these aspects of the design, we found it difficult to implement a correct protocol with traditional methods. The use of Teapot has resulted in clearer abstraction levels, increased system confidence, and reduced complexity in the implementation of cache coherence in xFS. At the same time, there are significant differences between xFS and the original applications which Teapot was designed to support. These differences have revealed some shortcomings of Teapot.

5.1 Caching in xFS

The three main components of an xFS system are the *clients*, the *managers*, and the *storage servers*. Under the xFS architecture, any machine can be responsible for caching, managing, or storing of any piece of data or metadata by instantiating one or more of these subsystems. Figure 7 shows a sample xFS installation.

Each of the three subsystems implements a specific interface. A client accepts file system requests from users, sends data to storage servers on writes, forwards reads to managers on cache misses, and receives replies from storage servers or other clients. It also answers cooperative cache forwarding requests from the manager by sending data to other clients. The job of the metadata manager is tracking locations of file data blocks and forwarding requests from clients to the appropriate destinations. Its functionality is similar to the directory manager in traditional DSM systems.

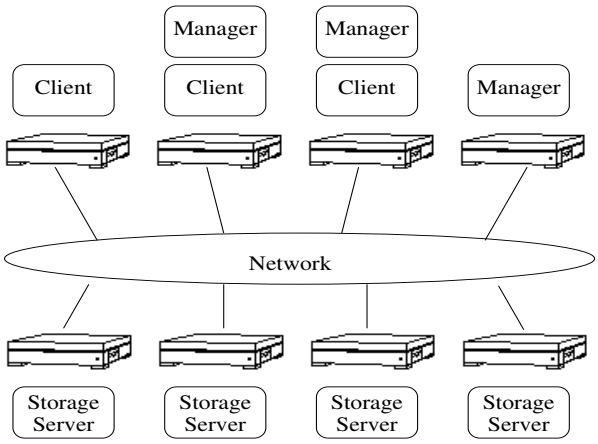

Figure 7: A sample xFS configuration. Clients, managers, and storage servers provide a global memory cache, a distributed metadata manager, and a striped network disk respectively.

Finally, the storage servers collectively provide the illusion of a striped network disk.

xFS employs a directory-based invalidate cache coherence protocol. This protocol, while similar to those seen in traditional DSM systems, exhibits four important differences that prevent xFS from using previously developed protocols and that complicates the design of xFS. (1) xFS separates data management from data storage. Although this separation allows better locality and more flexible configuration, it splits atomic operations into different phases that are more prone to races and deadlocks. (2) xFS manages more storage levels than traditional DSM systems. For example, it must maintain the coherence of the kernel caches, write-ahead logs, and secondary storage. (3) xFS must maintain reliable data storage in the face of node failures, requiring protocol modifications that do not apply to DSM systems. For example, a client must write its dirty data to storage servers before it can forward it to another client. (4) The xFS client is heavily multi-threaded and it includes potentially blocking calls into the operating system, introducing more chances for synchronization errors not seen in DSM systems.

5.2 Implementation Challenges

The xFS design and environment make the implementation and testing of cache coherence in xFS more difficult than in most systems. The usual problems of proliferation of intermediate states and subtle race conditions were even worse for xFS, as described in the following.

5.2.1 Unexpected Messages and Network

Reordering

An xFS node can receive messages that cannot be processed in its current state. This is also a problem in most DSM coherence systems, but it is particularly pervasive in xFS because xFS separates data storage and control and thereby makes it difficult to serialize data transfer messages and control messages with one another: data transfer messages pass between clients and storage servers or between clients and clients while control messages pass between clients and managers or storage servers and managers.

The xFS protocol also suffers from the message reordering problems as mentioned in Section 2. Further compounding the problem, this protocol often allows multiple outstanding messages in the network in order to maximize performance. For example, an xFS manager does not wait until a client completes a forwarding request to continue, so a subsequent invalidate message can potentially reach the same client out of order. Although such ordering can be enforced at the communication layer[5], recent research has argued that this ordering is best expressed with application state[8]. Furthermore, even if the network ensured in-order messages between nodes, the causes mentioned in the previous paragraph would still require xFS to explicitly handle unexpected message arrivals.

5.2.2 Software Development Complexity

Managing the large number of states needed to implement the xFS state machine was a challenge. Although, intuitively, each block can be in one of only four states—*Read Shared*, *Private Clean*, *Private Dirty*, or *Invalid*—the system must, in fact, use various transient states to mark progress during communication with the operating system and the network. Dealing with unexpected or out of order messages, handling the separation between data storage and data management, maintaining multiple levels of storage hierarchy, and ordering events to ensure reliable data storage all increase the number of transient states needed to handle xFS events. Even a simplified view of the xFS coherence engine contains twenty-two states. One needs a systematic approach when dealing with this large state space.

As we were implementing the protocol, it became clear that the C language was too general. Despite our best intentions, aspects of implementations that were not related to protocol specification were mixed in. The result was less modular, less general, harder to debug, and harder to maintain. Although the xFS protocol is similar to many other DSM protocols, we have found it non-trivial to reuse or modify existing codes, due to their ties to the native environments.

5.3 Teapot and xFS

After several unsuccessful attempts at completing the cache coherence protocol using traditional development methods, we decided to rewrite the system using Teapot. Our experience with this domain specific language has been positive. In particular, the close ties between Teapot and the MurΦ verification system have provided us with an effective testing tool for attacking the problem of unexpected event ordering; many of the bugs we found and corrected would have been extremely difficult to isolate through field testing alone. Furthermore, several aspects of the Teapot language have simplified the engineering complexity in our system.

5.3.1 Testing for Unexpected Event Orderings

Figure 8 shows an example of a bug in an early version of the xFS protocol that would have been difficult to isolate via field testing but which MurΦ easily discovered. In this version of the protocol, we saw no need for the manager to maintain sequence numbers for its outgoing messages. If a receiver of a manager request was not ready to act upon it, it simply queued it for later processing. MurΦ found the following deadlock bug:

Initially, client B is the sole cacher of a clean block. (1) Client C sends a read request to the manager. (2) The manager forwards the request to client B. To indicate that Client B should send the data to Client C via cooperative caching; the manager also updates its state to indicate that both client B and C are caching the data. (3) Meanwhile, client A sends a write request to the manager. (4) The manager sends a revoke request to client B, which arrives at client B before the previous forwarding message, invalidating its data. (5) The manager sends a second revoke request to client C, which client C queues, because its requested data has not arrived. (6) Client B sends a write request to the manager, which the manager queues, because its previously sent revoke message has not been acknowledged. (7) The delayed forward message from step 2 finally arrives, which client B queues, because its request to the manager has not been satisfied. Now we have finally reached a deadlock: client A is waiting for the manager to complete the revoke operations; the manager is waiting for client C to acknowledge the revoke request; client C is waiting for client B to supply the desired data; and client B is waiting for the manager to process its write request. One solution is to use sequence numbers to order the outgoing messages for a particular block from the manager, so the sequence of events seen by any client is consistent with the view of the manager.

5.3.2 Reduced Software Development

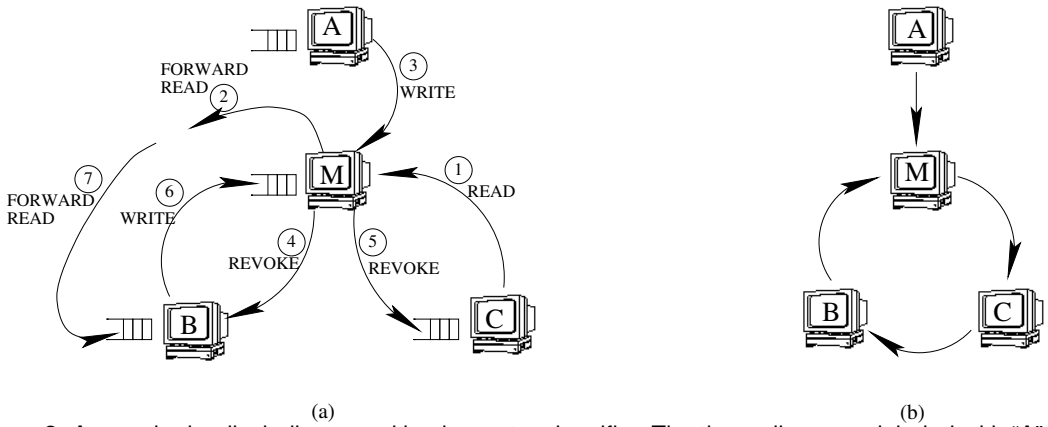

Figure 8: A sample deadlock discovered by the protocol verifier. The three clients are labeled with "A", "B", and "C". The manager is labeled with "M". In Figure (a), arrows denote the directions of the messages. The numbers denote the logical times at which messages are sent and/or received. Shown to the left of each host is a message queue, which holds the requests that are waiting to be processed. Messages that are not queued are processed immediately. In Figure (b), arrows denote the wait-for relationship, and the presence of a cycle indicates a deadlock.

Complexity

Four aspects of the Teapot language simplified the engineering of xFS. First, Teapot's continuations significantly reduced the number of states needed by xFS's protocol by combining each set of similar transient states into a single continuation state. Second, Teapot is a more appropriate notation for specifying coherence protocols because it is more restrictive and the specifications are written in a fairly stylized way; by matching the language to the task at hand, Teapot eliminates a source of bugs. Third, the domain-specific language forces one to concentrate on the problem at hand and separate it from other implementation details. In our case, the use of Teapot has resulted in more modular and more general-purpose code that is well isolated from the rest of the file system. Finally, the domain-specific language encouraged software reuse by isolating features that are common to the class of problems they are designed to solve. In our case, we were able to inherit many support structures such as message queues and state tables from other protocols supplied with the Teapot release, further reducing complexity and chances of errors.

5.4 Teapot Shortcomings

Teapot was designed and is best suited for DSM environments in which the primitives available to protocol handler writers are limited and simple. The xFS coherence engine, on the other hand, must interact with other components of the system such as the kernel and the active message subsystem via more powerful operations such as system calls and thread synchronizations. This difference in terms of power and expressiveness of the handler primitives have revealed some shortcomings of Teapot that were not apparent in its original application domain.

The first shortcoming is the lack of support for multithreading. An xFS client is heavily multithreaded to support concurrent users and react to concurrent requests from the network, but the coherence engine generated by Teapot has a large amount of global state and is difficult to make thread-safe. Transforming the resulting Teapot coherence engine into a monitor was unsuccessful, as subtle thread deadlocks occurred when different xFS threads enter the coherence engine and other xFS modules in different orders.

The second shortcoming concerns blocking operations on local nodes, which occur frequently in xFS coherence handlers. For example, when an xFS client needs to invalidate a file data block it caches, it makes a system call to invalidate the data cached in the kernel. This system call might block, recursively waiting for some other event that requires the attention of the coherence engine. Although Teapot provides good support for blocking operations that wait for remote messages, using the same mechanism to handle local blocking operations is tedious. In the above example, one must split the synchronous system call into asynchronous phases, invent a new node to represent the kernel, invent new states for the kernel node, invent new messages the kernel must accept and generate, and write a number of handlers to tie all these elements together. Better support for local blocking operations would have significantly eased the xFS protocol implementation.

The third shortcoming concerns users' inability to add new arguments to Teapot handlers. We were faced with

the unpleasant dilemma of either modifying Teapot itself or simulating additional arguments via global variables. The former suggests a limitation of the model; the latter work around is bad software engineering and in particular, it makes the multithreading problem worse. A more severe restriction is Teapot's lack of support for operations that affect blocks other than the block on which the current message arrives. The problem arises, for example, when servicing the read fault of one block by an xFS client requires the eviction of a different block. This is similar to the problem encountered by LCM during its reconciliation phase.

6 Related Work

The Teapot work most closely resembles the PCS system by Uehara et al. at the University of Tokyo [25]. They described a framework for writing coherence protocols for distributed file system caching. Unlike Teapot, they use an interpreted language, thus compromising efficiency. Like Teapot, they write protocol handlers with blocking primitives and transform the program into a message-passing style. Our work differs in several aspects. Teapot's continuation semantic model is more general than PCS's, which is a message-driven interpretation of a protocol specification. PCS's application domain is less sensitive to protocol code efficiency, so they do not explore optimizations. Finally, we exploit verification technology by automatically generating an input specification for the MurΦ verification system.

Reactive programming languages, such as ESTEREL [4], are useful for describing reactive systems and real-time applications. Teapot resembles ESTEREL in that it provides a specification of the control part of the protocol, leaving data manipulation to separately written (often in C) support routines. Like ESTEREL, Teapot supports verification and can be translated to executable code. Teapot differs from ESTEREL in that its emphasis is on simplifying the task of programming complicated finite-state machines.

Wing et al. [27] present an eloquent case for using model checking technology with complex software systems, such as a distributed file system coherence protocols. We also use model checking technology, but our primary focus is on a language for writing coherence protocols, and on deriving executable code as well as the verification system input from a single source. They write the input to the model checker separately from their code, which introduces the possibility of errors.

Domain-specific languages have recently found considerable interest in the systems programming community. Recent examples include instruction-set description languages [22, 3], an interface description language for remote procedure call stub generation and optimization [11], and a specification language for automatically generating network packet filters [21].

7 Conclusion: Implications for Domain-Specific Languages for Systems Software

It would be gratuitous to reiterate the successes and shortcomings of Teapot. Instead, we present some generalized insight gained from the Teapot exercise. While our experience has been with only one domain-specific language, we hope that our observations will be useful for designing other domain-specific languages, particularly for systems software. First, we make a few general conclusions, and then a number of detailed points that have emerged out of our experience with building and using Teapot.

- We hope our work provides further and concrete evidence that it is better to build application-specific tools, than to program complex systems with ad-hoc code.

- In our experience, it is more profitable to start with a focused domain-specific language or tool that solves a very specific problem to the satisfaction of a small user-community. Language extension and attempts at generalizing the application-domain should be considered only afterwards. Languages and tools with a large scope to begin with run the risk of being useful to no one, because they could take much longer to design and implement, and ultimately be less useful to users than a more focused tool.

7.1 How big to make the language?

An important consideration when designing a domain-specific language is: how general should the language be? Teapot relies heavily on externally written routines. For example, it has to call a function `SameNode` to compare two values of the type `NODE`, because we could not decide how far, if at all, we wanted to support equality on opaque types in the language. Should procedure calls be a part of the language? If so, are there any restrictions to be observed in the code for the procedures? For example, Teapot does not allow `Suspend` inside called procedures.

Making a language more comprehensive has the advantage that less code needs to be written in external routines. However, a larger language is harder to learn, harder to implement fully, and harder to optimize. While smallness has virtues, a designer should not go overboard and apply senseless restrictions. In Teapot, for example, most users were unhappy about the fixed set of arguments that appeared as handler parameters.

Capturing the commonly occurring programming scenarios is an important role of domain-specific languages. Teapot, for example, incorporates carefully designed abstractions for waiting for asynchronous messages. However, these abstractions were less effective at capturing the scenario of waiting for asynchronous *events* in general. This kind of waiting in xFS had to be cast into the waiting-for-messages idiom using extra messages. In hindsight, the language could have been designed to support asynchronous events, with messages as a special case of events.

For problem domains where it makes sense, it is imperative to think about automatic verification from the very beginning. In Teapot, for example, we maintained a clear distinction between opaque types and their implementation. In fact, the language has no mechanism to describe the implementation of opaque types. This was done so the verification system and C code could provide an implementation suitable for their purpose, rather than providing a common base implementation which may be poor for both purposes. An example of such an abstract type is a list of sharers, which is implemented using low-level bit manipulation in C, but using an array of enumerated type 0..1 in MurΦ. The language provides no pointers or dynamic memory allocation.

7.2 Compiler issues

Ideally, language users should only need to know the language definition, not the details of the language implementation. Even the popular general purpose languages fall short of this ideal by great distances, at least in the context of systems software. We have three observations in this regard. First, storage allocation policy should be made clear—programmers generally like to know where in memory particular variables live and what their lifetime is. In Teapot, the storage for state parameters was not clearly defined. It was not clear to the programmers how the memory management of continuation records happened. In fact, in the current implementation, unless `Suspends` and `Resumes` dynamically match, there would be a memory leak on continuation records, as we do not provide garbage collection. Fortunately, most protocols naturally have such balanced `Suspend` and `Resume` paths. Second, compiler optimizations should be explicitly specified and should be under user control. Even with all the virtues of verification, a systems programmer may need to go to low-level debuggers (perhaps for reasons unrelated to the coherence protocol). A restructuring compiler such as Teapot's makes the generated code harder to trace at runtime. Finally, despite these complications, we believe that aggressive optimizations are essential. In our experience, users are unwilling to compromise efficiency for ease of programming, particularly considering that speed is often the main purpose of distributing a computation.

7.3 Threads

As thread programming enters the mainstream, if domain-specific languages are used to generate components of systems software, their designers must pay close attention to thread support. Even when the language does not currently support threads, if it is successful, sooner or later users with multithreading needs would want to use it. The DSL designer, due to her unique knowledge of the internals, should be prepared to provide recommendations, if not a full implementation, of thread support. The first observation from our experience is that thread support cannot be treated as an afterthought; instead it must be an integral part of the early language design. When we attempted to make Teapot thread-safe as an add-on, we quickly discovered that global state made this an error-prone process. Even though we only introduced a small number of coarse grain locks, they frequently led to subtle synchronization problems because these locks were not exposed at the interface level. They broke abstractions and could easily lead to deadlocks. The second observation concerns the different alternatives that can enable the module written in a domain-specific language to interact with other multithreaded components. We have found that a viable alternative to making Teapot thread-safe is to turn the generated code into a single threaded *event loop* [20]. Instead of allowing multiple threads to execute concurrently in the cache coherence state machine, these threads interact with the single thread of the state machine via events. This approach eliminates unnecessary thread synchronizations inside the state machine.

7.4 Distribution and Cost of Entry

Most users would be reluctant to even install a new programming language, much less learn it. Thus, designers of domain-specific languages should be prepared to do considerable hand-holding: provide a very complete set of examples, documentation, and a distribution that builds "out-of-the-box". The xFS group found that hav-

ing a set of complete examples was a crucial aid to adopting Teapot. However, in the case of Teapot, we faced two stumbling blocks: we had to ask our users to go pick up SML/NJ compiler from Bell Laboratories, and the MurΦ system from Stanford. Many people gave up at this point, even when we offered to lead them through obstacles. Perhaps clever *perl* scripts could be built which would pick up the right software from web. To add to our difficulties, all the pieces of our system—SML compiler, MurΦ compiler, and the Teapot source—were constantly in flux and it was very difficult to maintain coherence [sic]. We see no easy way out of this situation. From the point of view of distribution, it would be best to provide everything in portable C code. However, without drawing upon previously distributed software, we couldn't have built Teapot in a reasonable amount of time.

7.5 A spade is not a general-purpose earth-shattering device

A tool-builder should be up front about what a tool does and does not do. Despite our care, several people thought of Teapot as a verification system, which it is not. In fact, we got an inquiry about Teapot which implied that we have discovered a more practical way of doing model-checking, rather than brute-force state-space exploration! Also, we note that Teapot is not directly suitable for describing hardware cache-coherence controllers because it permits unbounded levels of continuations. We were also asked why Teapot would not be suitable for model-checking systems unrelated to cache-coherence. These observations became apparent when people forced us to think beyond the context of Blizzard style DSMs. One should think carefully about a language's or system's restrictions and why they exist from the beginning, so as not to unnecessarily frustrate potential users.

8 Availability

Teapot is freely distributed. Please see the Teapot page for the latest version: `http://www.cs.wisc.edu/~chandra/teapot/index.html`

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[3] Mark W. Bailey and Jack W. Davidson. A Formal Model of Procedure Calling Conventions. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, California, January 1995.

[4] G'erard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Technical Report nn, Ecole Nationale Sup'erieure des Mines de Paris, nd.

[5] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[7] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[8] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[9] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[11] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas*, Las Vegas, Nevada, June 1997.

[12] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, November 1996.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[14] Kirk L. Johnson, M. Frank Kaashoek, and Deborah A. Wallach. CRL: High Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[15] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[16] James R. Larus, Brad Richards, and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 8, pages 297–342. MITP, 1996.

[17] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[18] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency for the World-Wide Web. Technical report, Department of Computer Science, University of Washington, May 1997.

[19] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[20] J. K. Ousterhout. Why Threads Are a Bad Idea. http://www.sunlabs.com/~ouster/, 1995.

[21] Todd A. Proebsting and Scott A. Watterson. Filter Fusion. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[22] Norman Ramsey and Mary F. Fernandez. The New Jersey Machine-Code Toolkit. In *1995 Usenix Technical COnference*, pages 289–302, New Orleans, LA, January 1995.

[23] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[24] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[25] Keiraro Uehara, Hajime Miyazawa, Kouhei Yamamoto, Shigekazu Inohara, and Takasha Masuda. A Framework for Customizing Coherence Protocols of Distributed File Caches in Lucas File System. Technical Report 94-14, Department of Information Science, University of Tokyo, December 1994.

[26] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[27] Jeannette M. Wing and Mandana Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings ACM SIGSOFT Symposium On The Foundations Of Software Engineering*, October 1995.