USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

The Case for a Session State Storage Layer

Benjamin C. Ling and Armando Fox
Stanford University
{bling, fox} @ CS.Stanford.edu

Abstract

This paper motivates the need for a session state storage layer. Session state is used in a large class of applications. Existing session state storage solutions often rely on ad-hoc choices such as databases or file systems, and exhibit various drawbacks such as poor failure/recovery behavior or poor performance, often because the solutions are *too general*. We present a design for a simple, fast, scalable, and fault-tolerant session state store that we believe accurately addresses the needs of session state retrieval, while avoiding the drawbacks of existing solutions.

Introduction

The concept of a user session is present in nearly all client-facing applications, including web-based applications. A user actively works for a period of time, called a *session*, until he signs out, or his session expires after a fixed interval. During the session, the application may produce temporary data relevant to the user's session, e.g. which step the user has completed in the application workflow. Upon session completion, the temporary state is no longer needed.

Session state is state whose lifetime is the duration of a user session and is relevant to a particular user; the duration of a user session is application specific. Examples of session state are user workflow state in enterprise software and user navigation in eCommerce.

Many architectures have modules that produce and use session state, including emerging industry standards such as J2EE[1]. Regardless of architecture, session state and its storage is a building block that is useful. Session state is a large class of state, as there are many different types of data that can be generated and used depending on the application at hand. In this paper, we focus on an interesting subset of session state, with distinct requirements and properties, which we describe in the upcoming sections.

In Section 1, we present an example of how session state is used. Next, we discuss the requirements and properties for the subset of session state that we address, and discuss the functionality that is necessary to provide a session state store, also highlighting what is *not* necessary. We then discuss current solutions and why they are inadequate. We propose a "middle-tier" storage layer to address session state. We discuss future work and related work, and conclude.

1. What is session state?

In Section 1, we describe a large subcategory of session state by describing how it is used, its properties, the requirements it imposes on its storage, as well as the functionality required to support it. Various different categories of session state exist. However, in the remainder of this paper, we will use the term "session state" to refer to the *subcategory* of session state which we describe below.

We use the example of a user working on a web-based marketing application to illustrate how session state is often used. The user is building a marketing campaign, specifying target customers and the offers they should receive.

A large class of applications, including J2EE-based and web apps in general, use the interaction model below:

- User submits a request (to add a targeted customer), request is routed to a stateless application server. This server is often referred to as the middle-tier.
- Application server retrieves the full session state for user (which includes the campaign data).
- Application server runs application logic (adds a customer to the targeted set of customers)
- Application server writes out entire session state
- Results are returned to the user's browser

Session state must be present on each interaction, since user context or workflow is stored in session state. If it is not, the user's workflow and context is lost, which is seen as an application failure to the end user, and is usually unacceptable from a product requirement standpoint. Session state, in this context, includes any temporary application state that is associated with a single user; the loss of this state is akin to losing a few hours work.

Typically, session state is on the order of 3K-200K [2]. Session state retrieval is also in the critical path of the control path – processing of the request cannot continue unless session state has been retrieved.

These requirements imply that session state solutions should have the following properties, besides traditional properties such as availability, scalability, performance:

- Session state retrieval should be fast, or negligible when compared to application processing
- Failures of the state store or any of its subcomponents should not result in data loss, otherwise users perceive an application failure

- Recovery of the state store and its failed subcomponents should be fast, for the same reason

Some important properties/qualities of the session state we focus on are listed below. Session state:

1. **Is not shared.** Each user reads his own state. Unlike state in its full generality, session state is accessed in a fixed pattern of alternating reads and writes: *Read 1* of session state for user U is followed by *Write 1,* which is followed by *Read 2,* followed by *Write 2.*
2. **Is semi-persistent.** Session state must be present for a fixed interval *T*, but can be deleted after *T* has elapsed.
3. **Is keyed to a particular user.** An advanced query mechanism to do arbitrary searches is not needed.
4. **Is updated on every interaction.** Session state such as user context in a web-based application is updated in its entirety on every interaction, as described earlier. A new copy of the state is written on every interaction.
5. **Does not need ACID [7] semantics**. Session state is transient, and state that requires transactions is not included in the class of session state we address.

Given these properties, the functionality necessary for a session state store can be greatly simplified (each point corresponds to an entry in the previous numbered list):

1. **No synchronization is needed.** Since the access pattern corresponds to an access of a single user making serial requests, no conflicting accesses exist, and hence race conditions on state access are avoided, which implies that locking is not needed.
2. **State stored by the repository need only be semi-persistent –** a temporal, lease-like [3] guarantee is sufficient, rather than the "durable" guarantee that is made in ACID [7].
3. **Single-key lookup API is sufficient.** Since state is keyed to a particular user and is only accessed by that user, a general query mechanism is not needed.
4. **Previous values of state keyed to a particular user may be discarded.**
5. **No need to support full ACID –** only atomic update is necessary; since each write writes out all of the user's session state, consistency is trivial and isolation is guaranteed. Durability is not needed.

Existing solutions and why they are inadequate

Currently, session state storage is done with one of the following mechanisms: Relational Database (DB), file system (FS), single-copy in-memory, replicated in-memory.

Frequently, enterprises use either the DB or FS to store session state, often reasoning, "I already have a DB and FS, why don't I just use one of them to store session state?" This simplifies management, since only one type of administrator is needed. However, there are several drawbacks to using either a DB or FS to handle session state, besides the costs of additional licenses and complexity of administration:

D1 Contention. Unless a separate DB/FS is created for session state, requests for session state and requests for persistent objects contend for the same resources. Session state read/write requests are frequent, which can interfere with requests for persistent objects that are housed by the same physical resource.

D2 Failure and recovery is expensive. If a crash occurs, recovery of the DB or FS may be slow, often on the order of minutes or even hours. Recovery time for a DB can be reduced if checkpointing is done frequently, but reduces performance under normal operation. There exist DB/FS solutions that have fast recovery, but these tend to be quite costly [9]. Even if recovery is on the order of seconds, in a large scale application, hundreds or thousands of users may see a failure if they attempt to contact the server at time of recovery.

D3 Session cleanup is painful. After state is put into a DB or FS, some process has to come back and look at the data and expire it, or else the data continues growing without bound. Reclaiming expired sessions degrades performance of other requests to the DB or FS.

D4 Potential performance problems. Reading/writing state objects to a DB/FS may sometimes incur a disk access in addition to a network roundtrip.

On the other hand, in-memory solutions (IMS) avoid several of the drawbacks of FS and DB, and are generally faster than FS/DB oriented solutions. In-memory solutions rely on affinity, and require a user to "stick" to a particular server that stores his copy of session state. A hardware load-balancer can guarantee affinity. However, the app-processing tier is no longer stateless; session state is being stored by the application server; it must serve the dual roles of application processing as well as providing state storage. Affinity is a key property for in-memory solutions to operate well – the main advantage of storing state in memory is to avoid a network roundtrip to the DB. If no affinity is present, then a server must incur a roundtrip to pass the request to the appropriate server. Affinity limits load balancing options, since load balancing must be done on the granularity of a user, rather than that of a request.

When only a single copy of a user's session state is stored on a corresponding application server, if a server crashes, state for some users is lost. The crash will be

manifested to users as an app failure, which is usually unacceptable.

A primary-secondary scheme is often used for a replicated solution. In BEA WebLogic™ [5], a J2EE application server, servers are given unique IDs and form a logical ring. A server *S* elects the server *T* that trails *S* in the logical ring to be its secondary. All users who are pinned to *S* as a primary share *T* as a secondary. A cookie is written out to the user's browser designating the primary and secondary.

During normal operation, all updates to session state are synchronously written, first at the primary, and then synchronously replicated at the secondary.

On failure of *S*, WebLogic™ does one of the following, depending on configuration.

1. A subsequent request destined for *S* is assigned to a random server *U*, because the load balancer recognizes the failure of *S*. *U* will copy the state information from the secondary *T*, and then rewrite the user's cookie, designating *U* as the new primary and *T* as the secondary.
2. A subsequent user request destined for *S* is assigned to the secondary *T*, because the software load balancer recognizes the failure of *S*. *T* will designate itself as the primary, replicate the session state information to its secondary *V*, and then rewrite the user's cookie, designating *T* as the new primary and *V* as the secondary.

There are several potential problems in this scheme (Note that some of the deficiencies of DB/FS solutions are shared by WebLogic™, as mentioned below):

D5 **Performance is degraded on secondaries**. D5 is related to D1. Instead of only providing application processing, secondary application servers face contention from session state updates.

D6 **Recovery is more difficult (special case code for failure and recovery).** The middle-tier is now stateful, which makes recovery more difficult. Special-case failure recovery code is necessary. In Case 1, a server *A* receiving a valid cookie stating that *B* as primary and *C* as secondary must realize that it must now become primary since *B* failed, and in Case 2, a secondary must realize that it should now become the primary. Special-case code makes the overall system harder to reason about, harder to maintain and less elegant.

D7 **Poor failure/recovery performance for Case 2.** Assuming equal load across all servers, upon failure of a primary *A*, the secondary *B* will have to serve double load – *B* must act as primary for all of *A's* requests as well as its own. Similar logic applies to *B's* secondary, which experiences twice the secondary load.

D8 **Lack of separation of concerns.** The application server now provides state storage, in addition to application logic processing. These two are very different functions, and a system administrator should be able to scale each separately.

D9 **Performance coupling.** If a secondary is overloaded, then users contacting a primary for that secondary will experience poor performance behavior as well. Because of the synchronous nature of updates from primary to secondary, if the secondary is overloaded, e.g. from faulty hardware or user load, the primary will have to wait for the secondary before returning to the user, even if the primary is under-loaded [4]. An industry expert has confirmed that this is indeed a drawback [6].

Note that *any* replication scheme requiring synchronous updates will *necessarily* exhibit performance coupling**.** That is, whenever a secondary is slow for any reason, any primary served by that secondary will block. In Case 1, after a failure of a single server, the entire cluster should be coupled, if we assume that the load balancer load balances correctly. To be specific, each request for *S* will be assigned to a random server *U*, and all of the nodes in the cluster will be performance coupled to the secondary *T*. This is particularly worrisome for large clusters, where node failures are more likely because of the number of nodes. Furthermore, application servers often use shared resources such as thread pools, and slowness in the secondary will hold resources in the primary for longer than necessary.

Proposed solution: A "middle-tier storage" layer

The support of industry in J2EE, together with the special qualities of session state and the failure of current solutions to address it properly, presents a viable opportunity to innovate and explore new state storage solutions. We present a solution that focuses on the following principles:

P1 **Avoid special case recovery code**. This addresses deficiencies D2 and D6. Avoiding special case recovery code reduces total cost of ownership, by facilitating easier administration and allowing programmers to reason about the system more easily, since the "special failure case" is not special, but rather the normal case [13].

P2 **Design for separation of concerns**. A system administrator should be able to scale the DB/FS separately from the session state store, and scale the session state store separately from the application processing tier. This addresses D1 D5, and D8.

P3 **Session cleanup should be easy**, and not require extra work. This addresses D3.

P4 **Graceful degradation upon failure**. Unnecessary performance degradation effects should not be seen, such as cache warming effects in DDS [4], and uneven load distribution in BEA. This addresses D7.

P5 **Avoid performance coupling**, as seen in DDS and which is present in the in-memory replication scheme. This addresses D9.

In addition, the system should be able to tolerate N simultaneous faults, where N is configurable by the system administrator. Unless N simultaneous faults occur, the system should continue operating correctly.

We assume a physically secure and well-administered cluster, along with a commercially-available high throughput, low latency redundant system area network (SAN) that can achieve high throughput with extremely low latency. High redundancy in the SAN enables us to assume that the probability of a network partition is arbitrarily small, and we need not consider network partitions. An uninterruptible power supply reduces the probability of a system-wide simultaneous hardware outage. The middle-tier storage has two components: bricks and stubs.

A brick stores session state objects by using a hash table. Each brick sends out periodic beacons to indicate that it is alive.

The stub is used by applications to read and write session state. The stub interfaces with the bricks to store and retrieve session state. Each stub also keeps track of which bricks are currently alive.

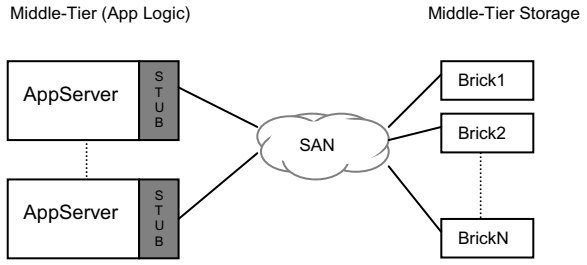

Figure 1

The write interface exported by the stub to the application is `Write(HashKey H, Object v, Expiry E)` and returns a cookie as the result of a successful write, or throws an exception if the write fails. The returned cookie should be stored on the client. The read interface is `Read(Cookie C)` and returns the last written value for hash key H, or throws an exception if the read fails. If a read/write returns to the application, then it means the operation was successful. On a read, we guarantee that the returned value is the most recently written value (recall that the type of session state we are dealing with is accessed serially by a single user).

The stub propagates write and read requests to the bricks. Before we describe the algorithm describing the stub-to-brick interface, let us define a few variables.

Call W the write group size. A stub will *attempt* to write to W of the bricks, and read from R bricks. Define WQ as the write quota, which is the minimum number of bricks that must return "success" to the stub before the stub returns to the calling application. We use the term quota to avoid confusion with the term quorum; quorums are discussed in section on related work. $WQ - 1$ is the number of **simultaneous brick failures** that the system can tolerate before losing data. Note that $1 \le WQ \le W$, $1 \le R$, and $R \le W$. Lastly, call t the timeout interval, an amount of time that the stub waits for bricks to reply to its requests.

The stub handles a write by doing the following:

1. Calculate checksum for object and expiration time.
2. Create a list of bricks *L,* initially the empty set.
3. Choose W **random** bricks, and issue the write of {object, checksum, expiry} to each brick.
4. Wait for WQ of the bricks to return with success messages, or until t elapsed. As each brick replies, add its identifier to the set *L.*
5. If t has elapsed and the size of L is less than WQ, repeat step 3. Otherwise, continue.
6. Create a cookie consisting of H, the identifiers of the WQ bricks that acknowledged the write, and the expiry, and calculate a checksum for the cookie.
7. Return the cookie to the caller.

The stub propagates the read to the bricks:

1. Verify the checksum on the cookie.
2. Issue the read to R random bricks chosen from the list of WQ bricks contained in the cookie.
3. Wait for 1 of the bricks to return, or until t elapses.
4. If the timeout has elapsed and no response has been returned, repeat step 2. Otherwise, continue.
5. Verify checksum and expiration. If checksum is invalid, repeat step 2. Otherwise continue.
6. Return the object to the caller.

The set of bricks that have the most recent write for key X changes on each write. While omitted in the algorithm for simplicity, a second-level timeout can be added for the case when writes/reads to the stubs continually time out; an exception can be thrown when this occurs.

For garbage collection of bricks, we use a method seen in generational garbage collectors [10]. Earlier we described each brick as having one hash table, for simplicity. In reality, it has a set of hash tables; each hash table has an expiration. A brick handles writes by putting state into the table with the closest expiration time after the state's expiration time. For a read, the stub also sends the key's expiration time, so the brick knows which table to look in. When a table's expiration has elapsed, it is discarded, and a new one is added in its place with a new expiration.

What happens on failure?

If a node cannot communicate with another, we assume it is because the other node has stopped executing. As discussed earlier, we assume that a network partition is not possible. We assume that components are fail-stop.

On failure of a client, the user perceives the session as lost, i.e. if the OS crashes, a user does not expect to be able to resume his interaction with a web application.

On failure of an application server, a simple restart of the server is sufficient since it is stateless. The stub on the server detects existing bricks from the beacons and can reconstruct the table of bricks that are alive. The stub can immediately begin handling read and write requests.

On failure of a brick, a simple restart of the brick is necessary. All state on that brick is lost; however, the state is replicated on *(WQ – 1)* other bricks, and so no data is lost. Furthermore, on a subsequent write of that data, *WQ* copies are made, and the system can once again tolerate *(WQ – 1)* faults without losing data.

An elegant side effect of having simple recovery is that clients, servers, and bricks can be added to a production system to increase capacity. For example, adding an extra brick to an already existing system is easy. Initially, the new brick will not service any read requests since it will not be in the read group for any requests. However, it will be included in new write groups because when the stub detects that a brick is alive, the brick is a candidate for a write. Over time, the new brick will receive an equal load of read/write traffic as the existing bricks.

When the system is under high load, and latency exceeds *t*, new requests will be generated while old ones have not yet been serviced, potentially increasing the load even more. To address this, bricks can discard a request if *t* has elapsed by the time the brick begins processing it. Secondly, we insert a random exponentially delay between each retry.

Middle-tier storage vs. design principles

We believe the design of this middle-tier storage system achieves the design principles outlined, and avoids the drawbacks of previous solutions while achieving good performance and exhibiting good failure/recovery behavior.

P1 **Avoid special case recovery code.** The system as described has no special case recovery code.

P2 **Allow for separation of concerns.** A middle-tier storage layer can be scaled separately from persistent storage usage and separately from application processing.

P3 **Session cleanup is easy.** Bricks can easily expire session by removing them from the hash. No extra process to scrub old data is needed.

P4 **Node failure results in graceful degradation.** Since multiple copies are available for any given write, a single node failure does not affect correctness, only the capacity of the system. Furthermore, multiple node failures do not affect correctness as long as the number of simultaneous failures is less than *WQ.*

P5 **Performance coupling is avoided by two strategies:**

Change the brick replica group on each write. In schemes where a key is mapped to a fixed set of replicas (i.e. Key X is always mapped to replicas A, B, and C), performance for key X is limited by the slowest in the replica group. This implies that if the entire cluster is functioning correctly, with the exception of a single node, all requests served by the faulty replica group will experience poor performance. It is important to note that *any scheme requiring synchronous replies from a fixed set of nodes will experience negative performance coupling* – namely, performance is limited by the slowest in the group.

Issue more writes than necessary and wait for a few of them to return. By issuing a write to more replicas than are required, we avoid performance coupling caused by faulty nodes. Say bricks *A, B, C* compose the write group for a given write of key *X*, and *B* is faulty or overloaded. If *WQ* is set to 2, the faulty node does not hamper performance of the higher-level request. This is especially important since the replica group for key *X* changes on each write – if we do not issue the write to more bricks than necessary, in the presence of a faulty brick, eventually all keys will experience poor performance at one point or another.

The ratio of the tunable parameters *WQ* to *W* allows the administrator to determine the performance/cost tradeoff.

Interesting properties of solution:

An interesting property is a negative feedback loop involving the stubs and bricks. For example, let *W* be 3, *WQ* be 2, and the write group be *A, B, C.* If *B* is faulty or overloaded, *A* and *C* will reply first. A subsequent read will not involve *B*. In general, *B* will fail to reply to most write requests in time. By monitoring the number of operations handled by a brick, an administrator can detect and replace faulty nodes.

If *B* is temporarily overloaded, it can start shedding writes, to bring load back to a reasonable level. Furthermore, since subsequent reads for the state will not be addressed to *B*, load is reduced. This may result in writes failing under high load, but it is often better to shed load early, as evidenced in Internet workloads.

Related Work:

A similar mechanism is used in quorum-based systems [11, 12]. In quorum systems, writes must be propagated to *W* of the nodes in a replica group, and reads must be successful on *R* of the nodes, where $R + W > N$, the total number of nodes in a replica group. A faulty node will often cause reads to be slow, writes to be slow, or possibly both. Our solution obviates the need for a quorum system, since the cookie contains the references to up-to-date copies of the data; quorum systems are used to compare versions of the data to determine which copy is the current copy.

DDS [4] is very similar to the proposed middle-tier storage layer. However, one observed effect in DDS is performance coupling – a given key has a fixed replica group, and all nodes in the replica group must synchronously commit before a write completes. DDS also guarantees persistence, which is unnecessary for session state. Recovery behavior also exhibits negative cache

warming effects; when a DDS brick is added to the system, performance of the cluster first drops (because of cache warming) before it increases. This effect is not present in our work, since recovered/new bricks do not serve any read requests.

From distributed database research, Directory-Oriented Available Copies [15] utilizes a directory that must be consulted to determine what replicas store valid copies of an object. This involves a separate roundtrip, and the directory becomes a bottleneck. In our work, we distribute the directory by sending the directory entries to the browser, leveraging the fact that for a given key, there is a single reader/writer.

We share many of the same motivations as Berkeley DB [14], which stressed the importance of fast-restart and treating failure as a normal operating condition, and recognized that the full generality of databases is sometimes unneeded.

The need for graceful degradation upon failure was recognized in Petal [16]. However, Petal uses chained declustered; nodes are logically chained, and upon failure of a node, the node's predecessor and its successor can service requests for it. Similar to DDS, Petal allows reads to be serviced by either a primary or secondary, but writes must occur at the primary, and data is locked on a write. Hence Petal shares some of the same disadvantages as discussed earlier under DDS. Another key difference between our work and Petal is that although both systems address storage, we address different levels of the storage hierarchy. Petal attempts to present the image of virtual disk to its clients, and hence must maintain and keep consistent metadata for disk block information; in our system, we deal with in memory objects, and need not maintain and keep consistent such metadata.

There are some interesting settings of *W, WQ,* and *R* that correspond to work done in previous research. Setting all the variables to 1 is the equivalent of single-copy memory. Setting *W* and *WQ* to 2 is roughly equivalent to the in-memory replication scheme adopted by BEA, and setting *W* to the total number of bricks and *R* to 1 is the equivalent of "write all, read any."

Future Work:

We hope to explore the effect of faulty or overloaded bricks on overall performance. We expect that the system will degrade gracefully in the presence of *n* faulty bricks when $W > n + WQ$.

An interesting point to investigate is "shooting bricks," either for garbage collection, or to restart a brick that is performing poorly because it has been running too long. With respect to GC, since session state for a client is rejuvenated on each request, it may be possible simply "shoot" bricks that are reaching capacity, and restart them proactively to avoid garbage collection. A large Internet portal employs a similar strategy, by proactive rebooting its web servers to avoid out-of-memory errors caused by memory leaks. We do not expect rebooting to have a significant impact on performance of the system, given a sufficient number of bricks and an appropriate setting for *WQ*.

We hope to investigate the effects of the ratio of *W to WQ* and how performance is affected. We expect that as *W* grows larger than *WQ* that the system will perform without bottlenecks, until system capacity is reached.

Conclusion:

This paper argues for a new middle-tier storage layer that handles session state. We believe the storage system avoids the pitfalls of previous solutions, in particular, poor failure/recovery behavior and performance coupling.

References:

[1] Sun Microsystems. Java2 EnterpriseEdition. http://java.sun.com/j2ee/.

[2] U. Singh. Personal communication. E.piphany, 2002.

[3] C.G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 2002-210, Litchfield Park, AZ, 1989.

[4] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for iInternet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation,* San Diego, CA, Oct. 2000.

[5] D. Jacobs. Distributed Computing with BEA WebLogic server. In *Proceedings of the Conference on Innovative Data Systems Research,* Asilomar, CA, Jan. 2003.

[6] D. Jacobs. Personal communication, BEA Systems, December 2002.

[7] J. Gray. The Transaction Concept, Virtues and Limitations. In *Proceedings of VLDB,* Cannes, France, Sept 1981.

[9] Network Appliance. http://www.networkappliance.com.

[10] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. SIGPLAN Notices, 32(5):97—108. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* May 1997.

[11] Robert H. Thomas: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. TODS 4(2): 180-209(1979)

[12] David K. Gifford: Weighted Voting for Replicated Data. Proceedings 7th Symposium on Operating Systems Principles: 150-162, 1979.

[13] G. Candea and A. Fox, Crash-Only Software, Submitted to HotOS 2003.

[14] M. Seltzer and M. Olson. Challenges in embedded database system administration. In Proceeding of the Embedded System Workshop, 1999. Cambridge, MA

[15] Concurrency Control and Recovery in Database Systems, by P.A. Bernstein, V. Hadzilacos and N. Goodman.

[16] Petal, Distributed Virtual Disks, Edward K. Lee and Chandramohan A. Thekkath. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, 1996), pp. 84--92.