Writing data-centric concurrent programs in imperative languages

Russell Sears
Yahoo! Research

Christopher Douglas
Yahoo! Research

Abstract

Declarative languages have been proposed for use in concurrent and distributed system development. In this work, we argue that the primary benefits of such languages come not from their declarative nature, but instead from the design patterns they espouse.

We explain how to apply these design patterns to C and Java and present two examples: a highly concurrent transactional storage system and a distributed storage implementation. We use case studies to highlight problems in current imperative and declarative approaches.

Compared to conventional imperative approaches, the additional rigor imposed by our approach improves testability and enables a wider range of systematic optimization and parallelization techniques. We inherit these properties from the declarative languages we mimic. The resulting implementations are structured as programs in those languages would be: they consist of view maintenance routines and asynchronous event handlers.

However, our use of manually generated code allows us to leverage the full range of imperative programming techniques. In particular, performance constraints sometimes force us to use techniques such as deadlock avoidance, invariant weakening, and lock-free updates. Such techniques are unavailable in current declarative runtimes; their correctness requires reasoning beyond the capabilities of current software synthesis systems.

Over time we expect higher level languages to improve dramatically, and we hope that some of our techniques will inform their designs. However, our concerns are more immediate: one of our systems is already in production, and development of the other is underway.

1 Introduction

Increased hardware concurrency presents a major challenge to software engineers. Hardware advances increase core counts and interconnect bandwidth, and therefore, the relative latency of interconnects throughout the memory hierarchy. Furthermore, solid state disks have increased I/O parallelism and bandwidth. Similarly, network bandwidth, but not latency, continues to improve.

These trends continuously expose software concurrency bottlenecks, leading to a vicious cycle: the frequency with which software must be revised has increased dramatically and the intellectual burden associated with scaling to current hardware (and testing the resulting system) increases each year. This exacerbates two problems with current development processes:

- Improved testing and verification are needed to cope with the increased complexity.
- Over the years, as new bottlenecks arise, developers must be able to modify the code and convince themselves of the correctness of the modifications.

We begin with a high-level overview of our approach. We then turn to a discussion of real-world case studies, in order of increasing sophistication. The first two describe our experiences rewriting legacy modules and focus on concurrency issues. The last two are based on our experiences building replication and caching services from scratch; we extend the ideas to incorporate error handling and software engineering issues. We conclude the paper with a discussion of open research directions.

2 Overview

A number of declarative languages target system development, and have been shown to be appropriate for state management [21], concurrent software development [33, 47] and network protocol implementations [2, 29]. Our work is based on the observation that programs written in such languages are structured differently than those written in imperative languages. Rather than adopt declarative languages, we apply *data-centric* design patterns to languages such as C and Java.

Data-centric designs have a number of distinguishing properties:

1. They are factored into state maintenance logic (*models*) and state transition functions (*controllers*) that are invoked by external code.[1] Together, models and controllers specify relational transducers that communicate by sending and receiving tuples.

2. The model consists of two types of relations: mutable base tables and read-only views. The views are formally specified [34] using relational calculus.

3. The controller consists of guards (*pre-conditions*) and state transitions (*post-conditions*) that are, again, specified in relational calculus [29, 2].

4. Finally, we specify weakened invariants that hold while the transition is executed, and therefore must be implied by both the pre- and post-conditions. We call such invariants *during-conditions*.

Having specified the system, we manually translate the specification into imperative code. If concurrency is of no concern we omit during-conditions and instead use coarse-grained locks. This leads to a tedious but straightforward translation process and yields code comparable to current declarative systems (Sections 3.1 and 3.2).

We have found that, for sufficiently complicated modules, this process is much faster than ad hoc approaches; the time taken to produce a specification and perform the translation by hand is dwarfed by the time needed to debug ad hoc code. Indeed, our early case studies were not implemented to validate the techniques presented here. Instead, we scrapped and rewrote existing ad hoc implementations after failing to track down bugs in our original attempts (Section 3.3).

The introduction of during-conditions allows us to leverage implementation techniques that are unavailable to existing declarative programs. However, it significantly complicates matters; otherwise language runtimes would incorporate such optimizations.

One must produce a specification that is both correct and implementable using the techniques at hand. In general, mapping from specification to code is no longer possible using current automated techniques, and may require extremely complex reasoning (e.g., [10, 30, 25, 8]). We cover simple approaches in Sections 3.1 and 3.2, and complex ones in Sections 3.4 and 4. It is the observation that automated techniques ultimately fail to generate such code that leads us to avoid their use, even for simple non-concurrent modules.

[1] These terms are borrowed from model-view-controller GUI programming frameworks, which are designed to simplify asynchronous program development [28].

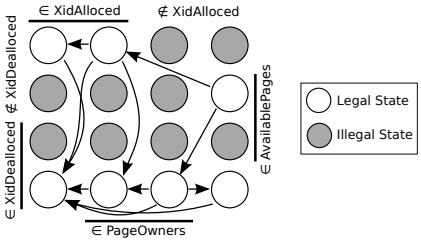

Figure 1: Partial allocation policy state machine. Compared to the complete relational specification (Equations 1 and 2), the state machine is unwieldy and uninformative.

This approach to software development works equally well with existing code. Instead of starting with a formal specification, we take existing modules, infer a specification (this often reveals bugs or fundamental problems), and reimplement the module using the above guidelines. We treat method invocations (and return values) as tuples. Depending on the legacy interface, we encode column values as method parameters or object fields.

3 Allocator case study

Our first case study is a record allocator from Stasis, a high-performance transactional storage system [35]. The allocator manages metadata for other, higher performance storage layouts [36, 37]. It consists of methods that manipulate pages and generate log entries, and a *policy* module that tracks space allocation and deallocation, places records on disk, and maintains invariants required by recovery. Although its correctness is crucial, concurrency and processing bottlenecks in the allocation policy have not yet become practical issues: it is protected by a single coarse-grained mutex.

Unlike traditional allocators, transactional allocators must isolate transactions from each other. In particular, space that was freed by in-progress transactions cannot be reused by other transactions.

3.1 State maintenance

We begin by specifying the allocation policy's state, which consists of three base tables:

```
AllPages: _pageid_, freespace
XidAlloced: _xid_, _pageid_
XidDealloced: _xid_, _pageid_
```

The _'s surrounding column names denote the primary keys of the tables. Furthermore, `XidAlloced` and `XidDealloced` are indexed both by `pageid` and by transaction id (`xid`). The `AllPages` relation contains the `pageid` and current `freespace` of all pages

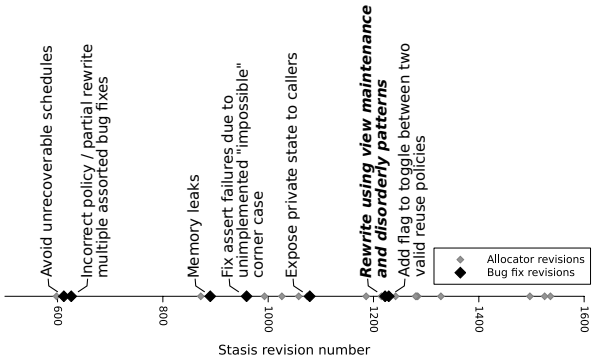

Figure 2: Six years of allocation policy revisions. Most bugs led to weeks or months of debugging effort.

known to the allocation policy. Each time a transaction allocates or deallocates a record on a page, a tuple is added to `XidAlloced` or `XidDealloced`, respectively. When a transaction commits, its entries are removed from `XidAlloced` and `XidDealloced`.

These tables are represented using red-black trees that implement an ordered set API, but any standard data structure would suffice. `XidAlloced` and `XidDealloced` each have two indexes, yielding two red-black trees per relation. `AllPages` is only indexed by `pageid` and is represented by one tree. In addition to the base tables, we maintain two materialized views:

```
AvailablePages: _pageid_, freespace
PageOwners: xid, freespace, _pageid_
```

`AvailablePages` is the set of pages with free space that can be reused by any transaction. In other words, it contains the rows of `AllPages` with no entries in `XidAlloced` or `XidDealloced`:

$$\{p \in AllPages : p \notin XidAlloced \wedge p \notin XidDealloced\} \quad (1)$$

`PageOwners` is a set of (page, transaction) pairs that tracks pages that transactions should continue to use (and can safely use) for future allocations:

$$\{p \in XidAlloced : \forall q \in XidDealloced \cup XidAlloced : p.page = q.page \Rightarrow p.xid = q.xid\} \quad (2)$$

`PageOwners` excludes pages with multiple entries in `XidAlloced` not for correctness, but to group related records (ones allocated by the same transaction) on disk, which reduces contention and improves locality [8].

To date, we have found that relational calculus expressions such as these clearly capture the runtime invariants of the systems we build, matching previous findings [2, 6, 42].

We have attempted to specify systems using state machines (Figure 1), but found that the results were too unwieldy to reason about or implement, and that they were not particularly amenable to verification techniques. The problem is that, for each additional table in our relational specification, the number of states in the analogous state machine doubles. In our allocator example pages can be in `XidAlloced`, `XidDealloced`, `AvailablePages` and `PageOwners`, leading to $2^4 = 16$ potential states. Even with the combinatorial explosion, these states do not capture sufficient detail about the state of the system, such as *which* transactions caused the page to transition to its current state.

A rich literature covers the techniques surrounding materialized view maintenance in relational settings [13, 34]. There, the primary challenge is to choose an appropriate update strategy given an expected workload. We have found that humans are reasonably good at choosing between various update strategies and manually translating view specifications into imperative code. We use coding conventions to facilitate the translation.

Helper methods maintain relations by atomically adding and removing tuples from the relevant trees. All modifications are performed against the base tables. Methods that modify base tables also perform materialized view maintenance, making it easy to confirm that the views are correctly maintained. For example, the following method adds a tuple to the `AllPages` relation:

```
allPages_add(allocation_policy * ap,
             pageid_t pageid, size_t freespace) {
  allPages_pageid_freespace tup = {
    pageid, freespace
  };
  int existed = set_add(tup, ap->allPages_pageid);
  if(!existed) {
    int existed2 =
      availablePages_add(ap, pageid, freespace);
    assert(!existed2);
  }
  return existed;
}
```

Note the use of naming conventions and types: relations' trees are named `relation_key1_key2_...` and `struct` type names encode the relation name and schema.

Furthermore, note that the method does *not* make any assumptions about the order in which it is called relative to other view maintenance functions. Section 3.3 explains why avoiding such assumptions is crucial.

The method relies on an invariant not mentioned above: `XidAlloced` and `XidDealloced` are subsets of `AllPages`.[2] Also, the `assert` verifies a logical consequence of `AvailablePages`'s specification. With these

[2]This invariant is maintained by the other allocation policy controllers, not by relying upon callers' good behavior.

observations, it is easy to see that the method implements the specification. Similar analysis of the other methods shows the views are correctly maintained.

3.2 State transition functions

This section explains how we specify and implement state transition functions that translate from module APIs to calls into the state maintenance code described above.

A number of declarative languages translate from state machine specifications to efficient implementations. Furthermore, state machine verification is well studied [17], as are problems of direct interest to us, such as state reachability in networks of communicating automata [12].

Rather than working directly with state machines, we prefer to specify systems using relational transducers [41]. This has a number of advantages: the relations and views we specify above correspond directly to transducer states, state transition specifications are written in relational calculus, the same language as view specifications, and, as above, translations from specification to imperative code are straightforward.

The method that handles completed transactions provides a concise example. Formally, and then in C:

$transaction\_complete(xid)$:

$$XidAlloced := \{p \in XidAlloced : p.xid \neq xid\}$$

$$XidDealloced := \{p \in XidDealloced : p.xid \neq xid\}$$

```
void transaction_complete(allocation_policy * ap,
                          transaction_id xid) {
  pageid_t *pids; int count; // OUT params
  xidAlloced_select_by_xid(ap, xid, &pids, &count);
  for(int i = 0; i < count; i++) {
    xidAlloced_remove(ap, xid, pids[i]);
  }
  free(pids);
  // Same for xidDealloced.
}
```

3.3 Experiences

Figure 2 summarizes the changes made to the allocation policy since it was introduced in Stasis revision 598. In some sense, the initial version was data-centric; it maintained sets at runtime and inferred the states of pages based on the sets they were present in. However, it mixed view maintenance and event handler logic, and was written in an *orderly* style; the calling code never performed certain transitions, so these transitions were left unimplemented. Other transitions were explicitly disallowed.

This led to three of the five bugs discovered before the design was scrapped, and created problems in the rest of the allocator implementation.

As one might imagine, allocator bugs are extremely difficult to track down. Each revision was tested extensively before being committed to source control. Most bugs that made it past testing persisted for years and took weeks to track down after manifesting.

Revision 1222 is a rewrite of the allocation policy. It implemented a correct policy, but not the policy required by Stasis' logging discipline. This was detected and fixed almost immediately, in revision 1227.

Ultimately, the rewrite was not due to a bug in the allocation policy, but instead due to a bug in a related module. At that point, we were unable to reason about the allocation policy's correctness, and opted to rewrite it from scratch. Doing so took a few days, and (more importantly) we have not had significant problems with this module since the rewrite. The current code is easier to read and understand than the previous iteration, and it helped expose the related bug.

The new allocation policy consists of three relations, two views and seven controller methods, and is implemented with red-black trees and 500 lines of code. Each method can be verified in isolation; the longest method is 32 lines of code.

3.4 Introducing concurrency

This section explains how we could modify the allocation policy to make use of finer-grained concurrency control. Rather than use a mutex to protect the allocation policy's state, we rely upon test-and-set operations exposed by concurrent data structure implementations [20, 32]. The difficulty is that each change to the allocation policy's state results in multiple index operations, allowing other threads to observe partial updates.

For instance, `XidAlloced` is backed by two indexes. We could implement updates by carefully ordering index operations. One index (say the one keyed by `xid, pageid`) would represent ground truth; the other would accelerate lookups into the first, yielding the following during-conditions:

$$XidAlloced = XidAlloced\_xid\_pageid$$

$$XidAlloced\_xid\_pageid \subseteq XidAlloced\_pageid\_xid$$

Of course, the controller implementation can rely on facts implied by the specification:

$$p \notin XidAlloced\_pageid\_xid \Rightarrow p \notin XidAlloced$$

We would maintain these weakened invariants by inserting into `XidAlloced_pageid_xid` before `XidAlloced_xid_pageid` and performing deletions in the opposite order.

Operations that manipulate multiple tuples require similar attention. Removing the mutex that protects

`transaction_complete` would prevent it from being atomic, leading to the following during-condition: *"The absence of a page from* `XidAlloced` *and* `XidDealloced` *implies that no active transaction has allocated / deallocated from the page."*

The analogous statement in the current specification is an if-and-only-if; we have reduced a set equality to a logical implication.

4 Buffer manager case study

The allocation policy deals exclusively with in-memory state. In contrast, the Stasis buffer manager interacts with disks and allows many I/O operations to be outstanding at once while preventing requests for unrelated data from interfering with each other. Furthermore, accessing cached data should be as fast as possible.

The mismatch in performance between disk and in-memory operations significantly complicates the design: short-running operations must not be blocked by unrelated long-running operations and, therefore, observe the system's state when such operations are in process.

We could deal with this by introducing additional states ("reading" and "writing" in the case of page buffers), further complicating the specification.

Instead, we opt to hide the additional transitions behind fine-grained mutexes. A bucket mutex protects the hashtable bucket that maps from `pageid` to the buffer. While holding the bucket lock, we atomically manipulate various pieces of in-memory state associated with the page, and then lock the page. We release the bucket, and, if necessary, initiate page I/O.

Although this technique is general-purpose, it requires non-standard functionality from the underlying data structure: all data structure operations are split into two phases. The first phase locates any existing data and obtains the appropriate mutex; the second either completes or cancels the operation, and releases the mutex.

Our buffer manager also incorporates protocols that vary lock orders depending on related page states. Determining that such protocols are correct requires careful reasoning about the related state invariants and transition functions. Automatic generation of such protocols is still beyond the state of the art (Section 6).

5 Network case studies

Our final two case studies are the replication protocol and metadata caching service of Walnut [11], a distributed storage system under development at Yahoo!. Unlike our other case studies, these systems are asynchronous: in order to make a request, the system sends a message. When a response arrives, a callback is invoked. As in the underlying network, message delivery is best effort (at most once), and most failures are treated as timeouts. We implement this by disallowing controller (and network transport) methods that return values or throw non-fatal exceptions. Such asynchrony eliminates the major problems with RPC: error-handling at the call site, and the latency of synchronous remote method invocations [43].

It also allows us to manipulate scheduling of computation during testing and at runtime without changing the event handling logic. This simplifies the use of automated testing tools [38, 39], and enables optimizations such as SEDA's thread pool sizing [44] and Cilk's work-stealing [9]. Furthermore, it hides framework-specific interfaces such as futures [7] behind clean APIs, preventing them from obfuscating the protocol implementation.

These network protocols are the first data-centric systems that we have built within multi-member teams. Although the specifications provide detailed descriptions of system semantics, they are unapproachable by people unfamiliar with the overall system design. One solution is to provide *sequence diagrams* that illustrate the ordering and flow of messages and events in various scenarios. Indeed, we naturally arrange controller methods in such orders, as doing so improves program readability.

6 Related work

A number of languages, libraries and programming language tools inspired our work.

Dataflow systems such as MapReduce [16], Dryad [23] and Click [27] achieve parallelism by restricting programs to conform to predefined dataflows and concurrency models, making them inappropriate for system implementation work. Concurrency control mechanisms such as lock managers and multi-version concurrency control allow SQL queries to safely perform arbitrary reads and writes. These mechanisms rely on transaction rollback and cope poorly with contention, leading to unpredictable latencies and throughput collapse [1, 19, 33].

A recent study of existing imperative programs finds that most systems resort to *ad hoc* synchronization primitives, and that, unlike mutexes, such techniques are extremely error prone. In particular, it is difficult to locate, let alone verify, the use of many such primitives without proper documentation. This leads to maintainability problems and bugs [46]. We address these concerns by isolating the use of such primitives to modules with well-defined concurrency semantics.

Languages such as Deputy [14] and CQual have explicit support for pre-conditions and post-conditions, and statically guarantee that functions have desirable properties such as memory safety. Such techniques are closely

related to our work; one possibility is to extend their analysis to support concurrency control annotations.

Orth's [45] authors propose dividing program state into "orthogonal" relations, views and state transition logic, which would be statically optimized by the compiler. Later representation synthesis languages [21] use relational view definitions and application traces to generate optimized C code. We extend this to concurrent software, and borrow the idea of *disorderly programming* [3] to correctly manipulate the underlying state, regardless of application control flow. Furthermore, our approach does not require specialized languages or development tools.

The Overlog [29], Dedalus [4] and Bloom [3] languages extend Datalog for asynchronous distributed system development. Although their syntax is beyond the scope of this paper, the following Overlog rule is equivalent to the definition of `AvailablePages` above:

```
AvailablePages(pageid,freespace) :-
   AllPages(pageid,freespace),
   notin XidAlloced(_,pageid),
   notin XidDealloced(_,pageid);
```

Like our work, programs written in these languages are naturally factored into view and controller logic. Techniques that parallelize their evaluation fall into three categories: concurrency control (discussed above), static partitioning and static monotonicity checks.

Partitioning finds updates that cannot conflict. When applied to databases [15, 18, 24], these techniques leverage runtime data, aggressively partition, and fall back to more expensive synchronization when necessary. Such techniques are workload dependent, and often rely upon transactional or other non-standard primitives, making them difficult to apply outside of database environments.

Monotonicity checks such as CALM [3] examine the structure of logic computations, and infer that certain computations (such as those without negation or aggregation) are embarrassingly parallel [5]. Such techniques can be applied directly to our specifications.

Each of these three techniques is promising, but none is applicable in all circumstances. We know of no declarative environment that incorporates the range of available techniques. Building such a runtime would be challenging, to say the least.

We believe the close relationship between these logic languages and our approach opens up the possibility of improved testability for imperative programs. Recent work on unit testing in Bloom demonstrates that thorough tests can be implemented with minimal effort [22]. Furthermore, a wide range of automated testing techniques for imperative languages are based upon theorem provers with knowledge of first order logic, and have been applied to small programs [38]. We suggest a two-tier approach to testing, where the specification is proven correct, and then each imperative method is shown to implement the corresponding portion of the specification. This mirrors the approach taken by existing large scale software verification efforts [26].

Finally, *software synthesis* takes declarative approaches one step further, and uses techniques such as constraint solvers and theorem provers to generate executable programs from incomplete specifications [31, 40]. Such systems could be used to "flesh out" specifications of our during-conditions, and also to automatically generate portions of the view and controller logic that are beyond the capabilities of other declarative techniques. Since the focus is on generating method bodies during compilation, such tools can be applied to computationally hard program generation tasks.

7 Conclusion

Our data-centric imperative software designs are based on formal specifications that mimic programming styles from declarative networking. Our specifications extend those languages with *during-conditions* which provide for weakened invariants while updates are in progress.

Manual translation of specifications to code provides developers with the full range of imperative concurrency control primitives. Our specifications are written with varying degrees of rigor, as we have focused our development efforts on the most problematic modules.

This "pay as you go" approach to data-centric software development has greatly improved our development process. For sufficiently complicated modules, the up front cost of writing specifications is more than paid for during testing and debugging. Furthermore, we have successfully retrofitted existing systems with data-centric modules, and have interfaced data-centric servers with a number of event and threading frameworks.

Although we currently develop without the aid of specialized tools, automatic verification of data-centric code should be significantly easier than verification of traditional system implementations: method implementations are extremely stylized, and abstract away complicated control flow and other intra-module dependencies. Furthermore, though our specifications are currently written for humans, they could easily be written in machine-readable forms for the purposes of formal verification and automated testing. Similarly, it may be possible to build software synthesis tools to generate sets of feasible during-conditions, or to translate from our concurrency control sketches to imperative code.

Data-centric programming has greatly aided our efforts to build highly concurrent, production quality systems; we believe that further research and improved tooling will yield significant improvements.

8 Acknowledgments

We would like to thank Philip Bohannon, Tyson Condie, Raghu Ramakrishnan and the anonymous reviewers; their suggestions greatly improved the presentation of this work. Much of this work was done during our interactions with Joe Hellerstein's group during the development of Overlog and Dedalus. Markus Weimer coined the term *during-condition*. Patrick Quaid reverse-engineered sequence diagrams from our code. Michi Mutsuzaki and Jianjun Chen graciously tolerated our sometimes pedantic objections to idiomatic language constructs. Daniel Wilkerson, Ras Bodik and Eran Yahav provided feedback and references to related work.

References

[1] AGRAWAL, R., CAREY, M. J., AND LIVNY, M. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems* (1987).

[2] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys* (2010), pp. 223–236.

[3] ALVARO, P., CONWAY, N., HELLERSTEIN, J., AND MARCZAK, W. R. Consistency analysis in bloom: a CALM and collected approach. In *CIDR* (2011), pp. 249–260.

[4] ALVARO, P., MARCZAK, W., CONWAY, N., HELLERSTEIN, J., MAIER, D., AND SEARS, R. Dedalus: Datalog in time and space. *Datalog Reloaded* (2011), 262–281.

[5] AMELOOT, T. J., NEVEN, F., AND DEN BUSSCHE, J. V. Relational transducers for declarative networking. *CoRR abs/1012.2858* (2010).

[6] ASHLEY-ROLLMAN, M., DE ROSA, M., SRINIVASA, S., PILLAI, P., GOLDSTEIN, S., AND CAMPBELL, J. Declarative programming for modular robots. *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS* (2007).

[7] BAKER, H., AND HEWITT, C. Incremental garbage collection of processes. Tech. Rep. 454, MIT AI Lab, Dec. 1978.

[8] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Notices 35*, 11 (2000), 117–128.

[9] BLUMOFE, R., ET AL. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing 37*, 1 (1996), 55–69.

[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *OSDI* (2010), pp. 1–8.

[11] CHEN, J., DOUGLAS, C., MUTSUZAKI, M., QUAID, P., RAMAKRISHNAN, R., RAO, S., AND SEARS, R. Walnut: A unified cloud object store. In *Sigmod* (2012).

[12] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8* (April 1986), 244–263.

[13] CONDIE, T., CHU, D., HELLERSTEIN, J., AND MANIATIS, P. Evita raced: metacompilation for declarative networks. *Proceedings of the VLDB Endowment 1*, 1 (2008), 1153–1165.

[14] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems* (2007), ACM, pp. 205–218.

[15] CURINO, C., ZHANG, Y., JONES, E. P. C., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *PVLDB 3*, 1 (2010), 48–57.

[16] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004).

[17] DEUTSCH, A., SUI, L., AND VIANU, V. Specification and verification of data-driven web services. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (2004), ACM, pp. 71–82.

[18] GANGULY, S., SILBERSCHATZ, A., AND TSUR, S. A framework for the parallel processing of datalog queries. In *SIGMOD* (ACM Press, 1990), pp. 143–152.

[19] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (1996), pp. 173–182.

[20] HANKE, OTTMANN, AND SOISALON-SOININEN. Relaxed balanced red-black trees. In *CIAC: Italian Conference on Algorithms and Complexity* (1997).

[21] HAWKINS, P., AIKEN, A., FISHER, K., RINARD, M., AND SAGIV, M. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (2011), ACM, pp. 38–49.

[22] HELLERSTEIN., P. BloomUnit: Declarative testing for distributed programs. In *DBTest* (2012).

[23] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[24] JONES, E. P. C., ABADI, D. J., AND MADDEN, S. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Conference* (2010), A. K. Elmagarmid and D. Agrawal, Eds., ACM, pp. 603–614.

[25] JOUKOV, N., TRAEGER, A., IYER, R., WRIGHT, C., AND ZADOK, E. Operating system profiling via latency analysis. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 89–102.

[26] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)* (Big Sky, MT, Oct. 2009), ACM SIGOPS, pp. 207–220.

[27] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (August 2000), 263–297.

[28] KRASNER, G. E., AND POPE, S. T. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming* (1988).

[29] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *SOSP* (2005), pp. 75–90.

[30] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS) PDS-15*, 6 (June 2004), 491–504.

[31] MORGENSTERN, A., AND SCHNEIDER, K. Program sketching via CTL* model checking. In *SPIN* (2011), A. Groce and M. Musuvathi, Eds., vol. 6823 of *Lecture Notes in Computer Science*, Springer, pp. 126–143.

[32] PUGH, W. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, University of Maryland, College Park, June 1990.

[33] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems*. McGraw Hill, 2003.

[34] ROSS, K., SRIVASTAVA, D., AND SUDARSHAN, S. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Record* (1996), vol. 25, ACM, pp. 447–458.

[35] SEARS, R., AND BREWER, E. Stasis: Flexible transactional storage. In *OSDI* (2006).

[36] SEARS, R., CALLAGHAN, M., AND BREWER, E. Rose: compressed, log-structured replication. *VLDB* (2008).

[37] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *SIGMOD* (2012).

[38] SEN, K. Concolic testing. In *22nd International Conference on Automated Software Engineering (ASE 2007)* (Nov. 2007), pp. 571–572.

[39] SIMSA, J., BRYANT, R., AND GIBSON, G. A. dBug: Systematic testing of unmodified distributed and multi-threaded systems. In *SPIN* (2011), A. Groce and M. Musuvathi, Eds., vol. 6823 of *Lecture Notes in Computer Science*, Springer, pp. 188–193.

[40] SOLAR-LEZAMA, A., ARNOLD, G., TANCAU, L., BODIK, R., SARASWAT, V., AND SESHIA, S. Sketching stencils. *ACM SIGPLAN Notices 42*, 6 (June 2007), 167–178.

[41] SPIELMANN, M. Verification of relational tranducers for electronic commerce. In *PODS* (2000), ACM, pp. 92–103.

[42] SZEKELY, B., AND TORRES, E. A Paxon evaluation of P2, 2005.

[43] TANENBAUM, A. S., AND VAN RENESSE, R. A critique of the remote procedure call paradigm. In *Proc. of the EUTECO 88 Conf.* (Vienna, Austria, Apr. 1988), R. Speth, Ed., North-Holland, pp. 775–783.

[44] WELSH, M., CULLER, D., AND BREWER, E. SEDA: an architecture for well-conditioned, scalable Internet services. *Operating Systems Review 35*, 5 (Dec. 2001), 230–243.

[45] WILKERSON, D. S., AND GOLDSMITH, S. F. Orth: Orthogonal programming. Unpublished, July 2005.

[46] XIONG, W., PARK, S., ZHANG, J., ZHOU, Y., AND MA, Z. Ad hoc synchronization considered harmful. In *OSDI* (2010), vol. 10, pp. 163–176.

[47] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.