# Concurrency—The fly in the ointment?

Stephen M. Blackburn and John N. Zigman*

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia
{Steve.Blackburn,John.Zigman}@cs.anu.edu.au

**Abstract**

Concurrency is a central pillar of the Java programming language, is implicit in the transactional model of computation adopted by most persistent systems, and has been widely studied in the context of orthogonal persistence. We argue that despite the substantial literature on concurrency control and transaction models for orthogonal persistence, a basic question as to the interaction between concurrency and orthogonal persistence has yet to be adequately addressed.

The demands of orthogonality appear to place orthogonal persistence at odds with more general approaches to concurrency control. Given its stated objective of providing a 'complete' computational environment, the future of the orthogonal persistence vision in general, and persistent Java in particular, will depend, to some extent, on the release of this tension through the realization of new approaches to the integration of concurrency and persistence.

This paper addresses both strict transactional and more 'open' approaches to managing concurrency in face of persistence, and examines difficulties with each approach. A simple transaction model that relieves some of these difficulties is presented and its limitations briefly examined.

## 1 Introduction

The challenge of building orthogonally persistent systems is highlighted by the ambitious design rules prescribed by Atkinson and Morrison [1995]—parsimony of concepts; orthogonality of concepts and completeness of the computational environment. The magnitude of this challenge is reflected by the absence of any orthogonally persistent system that properly fulfills these design rules. Of particular note is the failure of orthogonally persistent systems to adequately accommodate concurrency. This is made conspicuous by the sophistication of modern database systems and programming languages with respect to concurrency, because it is database systems and programming languages that orthogonal persistence brings together.

The role of concurrency as part of any computational environment is well established. Concurrency facilitates speedup by hiding latencies; is fundamental to scaleup (as product of distribution); and is intrinsic to the semantics of cooperative computations. Inadequate support for concurrency therefore constitutes a clear violation of the third of the above design rules: 'completeness of the computational environment'. The importance of the orthogonal persistence research community resolving this inadequate support for concurrency is therefore quite clear.

The key focus of this paper is identifying the problems that arise from the intermix of persistence and concurrency, and establishing why many of these problems are exacerbated by the *orthogonality* of persistence. Both strict transactional and more 'open' approaches to managing concurrency in the face of persistence are addressed, and difficulties with each approach are examined. A simple transaction model that relieves some of these difficulties is then presented and its limitations briefly examined.

Before addressing the problems that arise from the intermix of persistence and concurrency, we will briefly review persistence and concurrency in the following sections.

*The authors wish to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government's Cooperative Research Centers Program.

## 1.1 Persistence

The problem of managing data that outlives computations that operate over it is significant in both theoretical and practical dimensions. The binding of short-lived computations to long-lived persistent data raises fundamental questions about identity and type. On the other hand, a range of significant engineering issues are raised by the problem of efficient and coherent movement of data between volatile and stable storage. Among the various approaches to persistence, *database systems*, *database programming languages*, and *orthogonally persistent systems* are prominent. Each of these are briefly characterized below.

**Database Systems** The database field's approach to persistence has, since the early 70s, been dominated by the relational model [Codd 1970], in which the basis for data storage and management is the mathematical concept of relations. That the database industry is now worth around $US10 billion annually and is growing at about 35% p.a. [Silberschatz and Zdonic 1996] is evidence that the relational model has served it very well. However, there exists an enormous and rapidly growing volume of data in need of management to which the relational model is ill-suited. The database community responded to this need in the 80's and 90's with a new data model based on object-oriented programming principles—a response that led to the development of object-oriented database management systems (ODMS). However, the success of the ODMS push has been equivocal [Carey and DeWitt 1996]. More recently a hybrid of the two approaches has emerged. The object-relational model adds object-oriented features such as inheritance, complex object support, and an extensible type system to the relational data model. Current expectations among some in the database community are that the object-relational model is set to become 'the next great wave' in the database world [Stonebraker and Moore 1996].

**Database Programming Languages** Since the early days of database research, attempts have been made to reconcile the divergent programming language and database paradigms. One approach has been to try to integrate database functionality into programming languages, either by adding to existing languages or by designing new languages with database functionality. The product of such marriages, which, for want of a better term will be referred to here as *database programming languages* (DBPLs), is something that provides programmers with a more integrated approach to persistent data, but typically does so at the cost of data type orthogonality (only some types can be made persistent) and/or persistence identification (the persistence of an object is not defined in terms of reachability—leading to the problem of dangling pointers) [Atkinson and Morrison 1995].

**Orthogonal Persistence** The third member of the persistence family is *orthogonal persistence*. Orthogonally persistent systems are distinguished by an orthogonality between data use and data persistence. This orthogonality comes as the product of the application of the following principles of persistence [Atkinson and Morrison 1995]:

**Persistence Independence** The form of a program is independent of the longevity of the data which it manipulates.

**Data Type Orthogonality** All data types should be allowed the full range of persistence, irrespective of their type.

**Persistence Identification** The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system.

Some of the challenges that confront the designer of an orthogonally persistent programming language include the unification of type systems and data models, the development of mechanisms for binding to persistent data, and dealing with system evolution. The efficient engineering of orthogonally persistent languages raises a separate set of challenges, including: transparent and efficient movement of data between stable media and memory; efficient implementation of persistence by reachability; and efficient implementation of flexible recovery and concurrency control with respect to persistent data.

## 1.2 Concurrency

The study of interacting, concurrently executing computations remains fundamental to computer science, intersecting with and impacting upon a diverse range of fields within the discipline. Key issues faced by those researching concurrency relate to *control* over the interactions between concurrently executing computations, and efficient and correct facilitation of concurrent computation through *provision*

*of appropriate resources*. The first of these issues leads to the development of mechanisms by which interactions can be conducted in a meaningful (as opposed to haphazard or chaotic) way. The second relates particularly to the management of shared resources.

The first key issue, the control of interactions between concurrent computations, has traditionally been approached very differently by the database and programming language communities. On one hand, the database community has tended to focus on what could be termed 'throughput' solutions, where concurrency is primarily a means for speeding up the processing of otherwise serialized, independent computations. The focus has therefore been on controlling interactions between concurrent computations so as to facilitate concurrent access to resources while maintaining the appearance of serializable independence and isolation. On the other hand, the programming language community used concurrency to address a broader set of problems and has included more strongly a role for concurrency as a means of improving the responsiveness of logically unified, concurrent computation. A product of the view of concurrency that has tended to dominate in the database community is the prominence of the transactional computational framework, which emphasizes *atomicity* and *isolation*.

**Concurrent Programming Models** A range of programming constructs have been developed to facilitate coordination between cooperating concurrent processes. Included among these are constructs such as *monitors*, *semaphores*, and *mutexes*, which are used to control interactions between computations through shared values. Mechanisms such as *remote procedure call (RPC)* and *message passing* facilitate interactions between computations in the absence of shared values.

**Transactions** Transactions have been and continue to be the mainstay of the database community. Given the (simplistic) view of concurrency as being a means of speeding up the processing of otherwise serialized independent computations, it is natural that *serializability* and *isolation* are the key properties, from a concurrency standpoint, of (simple) transactions. Härder and Reuter [1983] characterized the most fundamental transactional model in terms of *ACID* properties—atomicity, consistency, isolation, and durability.

Atomicity is a powerful device for altering the granularity of interaction. In the absence of atomicity, the maintenance of coherency between concurrent computations requires continuous fine-grained interaction. With atomicity, coherency checks need in principle only occur at the transactional grain (temporally and morphologically). Atomicity also facilitates optimistic computation. Optimism is a powerful computational device—amongst other things it is a means of combating latency.

To the extent that isolation is desirable in a concurrent system, the ACID transaction model has served the database community well. However, such an approach is often too limiting. For example, long lived transactions, intra-transaction parallelism, and hierarchical transactional relationships are all curtailed by the rigidity of the ACID model [Elmagarmid 1992]. The database community has responded to this inflexibility with a range of transaction models that selectively break the ACID properties [Moss 1981; Elmagarmid 1992; Anfindsen 1997].

Intra-transactional concurrency may arise either as a product of a single transaction being distributed across multiple processes/processors, or through the use of an intra-process concurrency mechanism such as threads. By allowing concurrent intra-transactional computations to be isolated from each other by means of ACI (non-durable) sub-transactions, advanced transaction models provide one approach to intra-transactional concurrency. Alternatively, 'cooperative' programming language constructs such as monitors, semaphores and mutexes can be used, giving a quite different intra-transactional programming model.

# 2 Integrating Concurrency and Persistence

The task of *efficiently* and *coherently* moving data between volatile and stable storage, which is central to persistence, is complicated by concurrency. This is manifest in two distinct domains: concurrency between data movement and computation over the volatile data; and concurrency between separate computations operating over the volatile data. In the case of the former, the possibility exists of the coherency of the stable image of the data being corrupted by updates made by the computation while the transfer is underway. In the case of the latter the consistency of the volatile image of the data is a function of all of the concurrent computations, so there exists a need for coordination between the various computations in order for a consistent image of the volatile data to be determined for capture on stable storage.

Given the centrality of concurrency as a computational tool, persistent systems must provide effective means for dealing with the complexity that arises from the intermix of concurrency and persistence. The many approaches that have been taken in addressing this issue can be classified in terms of those that have the requirement that all computation over persistent data occur within a transactional context, and those that do not have this constraint. We will refer to these as 'transactional' and 'open' approaches respectively. In the following sections we will outline these alternatives, and will argue that neither has addressed the concurrency/persistence issue adequately to provide a general solution for *orthogonal* persistence.

## 2.1 Transactional Approaches

The bulk of the literature on transactions and transaction models has been driven by the problems of persistence and concurrency, so it should be no surprise that transactions provide a good solution to these problems. The key to this solution is the use of *atomicity* and *serializability* with respect to (concurrent) computations over persistent data so as to guarantee the consistency of the persistent image. All computations over persistent data must occur within a transactional context, and their results are only made visible outside of that transactional scope upon termination and only if the computation can be serialized with respect to prior 'successful' computations over the persistent data. An unserializable computation is 'aborted', any changes made by it being discarded. Transactions may execute concurrently as long as their effects are serializable—true inter-transactional concurrency is thus only seen between independent computations.

While the basic transactional model described above is somewhat restrictive, various 'advanced' transaction models have been developed that allow controlled relaxation of various constraints while maintaining guarantees about the coherency of persistent data, thereby providing much greater opportunities for concurrency without sacrificing coherency (see section 1.2).

The semantic impact of *atomicity* and *serializability* along with the requirement that all computation over persistent data occur within a transactional context, has meant that any use of transactions within a programming language has tended to impinge heavily on the underlying programming model[1]. On the other hand *isolation*, which is implicit in the atomicity property of transactions, allows the programmer to write transactional code in the knowledge that it will be isolated from the effects of any concurrent computation, thereby relieving the programmer of the need to explicitly account for concurrency of computation. The almost universal adoption of the transactional model for database systems and database programming languages suggests that these tradeoffs with respect to programmability have been accepted as reasonable in those domains.

By contrast, the transactional model has not been widely adopted within the orthogonal persistence community, most orthogonally persistent systems opting instead for 'open' approaches based on checkpointing and explicit synchronization (see section 2.2). The best known counter example is PJama [Atkinson et al. 1996], which embodies a sophisticated transaction model [Daynès et al. 1997].[2]

### 2.1.1 Transactions and Orthogonality of Persistence

Perhaps the reason for this aversion to transactional approaches within the orthogonal persistence community stems from the challenge to orthogonality inherent in the transactional model. Implicit to the transactional model is a notion of *two worlds*, one a world of persistent data, and the other an (external, non-persistent) world that issues transactions over the first. This dichotomy between persistent and external worlds is manifest in most database systems in the distinction between client application computation and embedded database computation, the latter typically taking the form of queries which are dispatched to a database server.

This duality is a fundamental product of the impossibility of the basic transactional construct—the ACID transaction—being invoked from within (atomic) transactions, and more particularly, the incapacity for the ACID transaction to be the basis for its own (nested) invocation. The property of *atomicity* ensures that any transaction (ACID or not) invoked within a transaction will, by definition, be subject to the atomicity of its parent. The *durability* (irrevocable stability) of any child transaction is thus at

[1] The proposed transactional framework for PJama[Atkinson et al. 1996] introduces the concept of a `TransactionShell` class, which allows 'runnable' objects to be invoked transactionally by simply passing them as an argument to a `TransactionShell` instance. The result is what Daynès, Atkinson, and Valduriez [1997] describe as a high degree of 'transactional transparency'.

[2] It is important to distinguish between PJama as presented in [Atkinson et al. 1996; Daynès et al. 1997], and PJama as currently implemented (versions $\leq$ 0.4), which provides only a global stabilization method (`stabilizeAll()`) and so fits better in the 'open' approaches category.

odds with the *atomicity* of the parent. Consequently, ACID transactions must be invoked not within transactions, but from within some external non-transactional computation.

The premise that all computation over persistent data occur within a transactional context and the need for a non-transactional context for the invocation of transactions leaves few alternatives for transactional orthogonally persistent systems.

One approach is to use computation over non-persistent data as the platform for invoking transactions over persistent data, the non-persistent invocation environment remaining non-transactional. However, such a demarcation of persistent and non-persistent computation is in clear violation of the principle of *persistence independence* (see section 1.1). Furthermore, we do not believe it would be possible to prevent leakage of references to persistent data into the non-transactional computation without changes to the Java language.

The alternative is that *all* computation occur within a transactional context, the operating system providing the non-transactional invocation environment. This analysis, when considered in light of the abovementioned restrictions of ACID transactions, gives rise to the following conundrum for those wishing to develop transactional orthogonally persistent systems:

> The principle of *persistence independence* and the impossibility of invoking the basic transactional construct—the ACID transaction—from within transactions appears to bind the granularity of transactions to process invocations. However, such a limitation would severely curtail opportunities for concurrency and so is clearly at odds with a key design objective for orthogonally persistent systems, '*completeness of the computational environment*' [Atkinson and Morrison 1995]. Transaction based concurrency control and orthogonal persistence thus appear to be at odds.

This conundrum appears not to have been addressed by proponents of transactional orthogonally persistent systems (other than by disregarding the concurrency issue and opting for the single transaction per invocation approach). The authors know of three proposals for orthogonally persistent systems that aim to support multiple transactions per invocation. The first, from the University of Massachusetts, is not confronted by the problem because it is essentially the single transaction per-process approach with an optimization whereby the cost of process invocation is saved through the reuse of process resources for the execution of a subsequent (identical) transaction. Although this model is not general, it is likely to find applications in the commercially important domain of (bulk) 'transaction processing'. The second is proposed by Garthwaite and Nettles [1996], who reported that they had not yet begun implementing 'multiple top level transactions' and omitted any discussion of the design for such a facility. The third system is PJama [Atkinson et al. 1996], and although a sophisticated transactional environment has been proposed [Daynès et al. 1997], there is no evidence that the above conundrum has been addressed in either [Atkinson et al. 1996] or [Daynès et al. 1997], although the conundrum is clearly present, as 'all code must be run within the scope of a transaction' [Daynès et al. 1997] and the objective of supporting multiple transactions per program invocation is fundamental to the PJama design.

## 2.2 'Open' Approaches

While the transactional model was developed with the express purpose of managing persistent data in the face of concurrency and has become the dominant approach in database systems, more open approaches with a closer affinity to concurrent programming models have a strong appeal in the context of orthogonal persistence, where persistence is an implicit facet of a programming language rather than explicit adjunct. Prominent examples of the use of open approaches include the Napier88 persistent programming language [Morrison et al. 1996] and the Grasshopper persistent operating system [Dearle et al. 1994].

The key challenge for proponents of open models lies in providing practical and efficient means for identifying consistent cuts through computation in order to facilitate coherent stabilization of data. The problem of identifying consistent cuts has been the focus of considerable attention in the distributed systems community, and for which general solutions are difficult [Schwarz and Mattern 1994].

Of the solutions available, the most simple is to provide a global stabilization mechanism and leave the problem of synchronization to the programmer. This approach is used in implementations of Napier88 [Morrison et al. 1996], Tycoon [Matthes et al. 1995], and current implementations of PJama [Atkinson et al. 1996][3]. While it is quite adequate for persistent applications such as those that require

[3]See footnote 2.

only coarse-grained stabilization guarantees, the complexity of synchronization and the global scope of stabilization make it an inappropriate choice for many applications.

A more refined approach is used in Grasshopper [Jalili and Henskens 1995], where causality dependence is used to minimize the effects of synchronous stabilization—only those computations and data which are causally interdependent are checkpointed synchronously, thus bounding the temporal and morphological scope of the checkpoint operation. Dearle and Hulse [1995] build upon this approach by using timestamps to facilitate an 'optimistic' causal dependency checkpoint scheme that does not require the synchronization of all interdependent computations. However, this approach depends on the concurrent processes taking their checkpoints at times that can provide consistent cuts—it will only find consistent cuts, not enforce them [Hulse 1998]. It is therefore necessary for the programmer or some other agent, such as an operating system daemon, to ensure that checkpoints are taken in such a fashion as to guarantee consistent cuts and therefore guarantee recoverability. Both approaches are implemented within the operating system kernel and are page-grained and therefore subject to various 'phantom' effects that are the product of morphological grain mismatch.

Stemple and Morrison [1992] foreshadow what is perhaps the most general and flexible 'open' approach, the communicating actions control scheme (CACS). CACS is a method for specifying systems of concurrency control that allows multiple models, including transactional models, to be specified. Stemple and Morrison [1992] do not address the issue of how CACS can, in general, be used to provide users with efficient and practical means of identifying consistent cuts through computation for the purposes of coherent stabilization. In the absence of such a capability or reports of implementation experience with CACS, it is hard to see CACS as a solution in itself to the problems that arise from the intermix of persistence and concurrency.

### 2.3 Persistence and Concurrency: Conclusions

It is clear that unlike cousins in the database and database programming language domains, orthogonal persistence research has struggled to integrate persistence and concurrency.

Transactional solutions appear to be limited by the conundrum resulting from the requirement that computation only occur within (ACID) transactional contexts and the impossibility of invoking ACID transactions from within a transaction. This appears to severely curtail the possibilities for concurrency in transactional persistent computation and so limits the generality of any such approach to orthogonal persistence.

Despite the attraction of approaches that do not impose the rigidity of the transactional model, there is yet to be published an account of an implemented system that provides an efficient framework for exploiting the flexibility of the 'open' approach while offering a sufficiently general solution to the problem of providing consistent stabilization of data in the face of concurrent computation.

Although these analyses of the two major approaches to dealing with persistence and concurrency suggest a rather bleak outlook for orthogonal persistence with respect to concurrency, the following section outlines a transaction model that, although not providing an 'ultimate' solution, addresses the pertinent issues in the transactional domain, and therefore provides a way forward.

## 3 A workable model

The most straightforward response to the problem of providing a control environment for the initiation of (ACID) transactions, where all computation in that control environment is itself contained within a transactional context, is to allow new ACID transactions to be spawned at the point of termination of another ACID transaction. The non-transactional 'glue' that launches new transactions could, in such a solution, be captured within an atomic call to `CommitAndSpawn()` and the requirement that all user computation occurs within a transactional context would thus be met. Such a `CommitAndSpawn()` call could take an arbitrary number of new transactions as arguments and be matched by an `AbortAndSpawn()` call for user-initiated aborts.

Inter-transactional isolation induces a dislocation of control flow which is at odds with familiar programming language semantics. Thus a simple linkage of an ad hoc series of transactions by such calls would not in general lead to a particularly usable programming model. However, by combining the `CommitAndSpawn()` mechanism with the *chain transaction* model [Gray and Reuter 1993][4], more ele-

[4]The 'chain transaction' presented by Gray and Reuter [1993] and referred to here is quite distinct from the 'chain transaction' described by Chrysanthis and Ramamritham [1994] as a special case of the joint transaction model. The latter does not give each transaction in the chain ACID properties, which is important to our model.

gant solutions begin to emerge.

The chain transaction model allows a series of transactions to be 'chained' together, the environment held by each transaction passed on to its successor along with any locks held. Modeling a (long) computation as a chain transaction is thus similar to modeling it as a single ACID transaction with checkpoints. Both have continuity and a relatively fine-grained recovery capacity, but they are distinguished by the former being composed of *N* discrete atomic parts in contrast to the latter's complete atomicity. Consequently the chained transaction foregoes rollback capacity and atomicity of the whole, but opens up the possibilities for more fine-grained concurrency and the possibility of invoking new transactions at each of the 'windows' delivered by `CommitAndSpawn()`.

Within the broad family of chain transactions, concurrency opportunities can be traded off against programmability. At one extreme, strict two-phase locking can be maintained across the life of a chain of transactions—thus guaranteeing read-write semantics with respect to all operations within the scope of the chain. Locking conflicts can be resolved in favor of chain transactions, ensuring that no chain is broken for the sake of a non-chained transaction. With such a model, the chain transaction approaches the behavior of a single ACID transaction with checkpoints. At the other extreme, all locks could dropped at each chaining point, forsaking read write semantics within the chain for increased concurrency opportunities.

The combination of `CommitAndSpawn()` with the chain transaction model offers a workable solution to the problem at hand. The *chain and spawn* transaction model trivially extends the chain model [Gray and Reuter 1993] with a `ChainAndSpawn()` construct that replaces the `Commit()` implicit in `ChainWork` with `CommitAndSpawn()`.

```
public static void main(String[] args ) {
        // implicit tx begin
        ...
        TXRequest ra = null;
        while( (rq = nextRequest()) != null) {
                switch (rq.rid) {
                case TXRequest.DEPOSIT:
                        tx.start(new Deposit(rq));
                        tx.claim();
                        break;
                case TXRequest.WITHDRAWAL:
                        tx.start(new Withdrawal(rq));
                        tx.claim();
                        break;
                case TXRequest.BALANCE:
                        tx.start(new Balance(rq));
                        tx.claim();
                        break;
                }
        }
        ...
        // implicit tx commit
}
```

Figure 1: The invocation of (ACID) transactions from within the program main().

The chain and spawn transaction model should fit well into the concurrency control framework for PJama proposed by Daynès et al. [1997][5]. This is illustrated by the simple example in figure 1, which places the aforementioned conundrum in the PJama context. The control code that comprises the main program must exist within a transaction, and yet each of the transactions to be invoked by the control code are classic examples of transactions that must be ACID. If the `tx` object were of a chain transaction type, then each `tx.start()` would in fact be a `ChainAndSpawn()`, the new transaction being invoked and the control computation being broken up into many small ACID transactions—one for each loop

[5]In keeping with the framework outlined by Daynès et al. [1997], when applying the chain and spawn model to Java, we assume that transactions are Java objects and are composed with Java code via `runnable` objects.

iteration. If two phase locking were ensured across the transactions composing the chain, normal read write semantics would be preserved throughout the execution of the outer loop. If, as in the above example, the control computation is primarily over non-persistent data, the cost of the `Commit`, `Begin` pair implicit in each `ChainAndSpawn()` will be small.

### 3.1 Limitations of the Chain and Spawn Model

The 'chain and spawn' transaction model provides a *workable* solution to the problem at hand. Although the example in figure 1 suggests that via the PJama transaction framework 'chain and spawn' transactions can provide a syntacticly transparent solution, in fact the semantic impact on the 'control code' is significant. The basis for decomposing the control computation into multiple transactions is *imposed* by the placement of `ChainAndSpawn()` calls, rather than being *derived* from the natural semantics of the computation. The simplicity of the example given renders this imposition barely noticeable, however if more general examples of control computation are considered, the potential for difficulty should be clear. The objective of *transaction independence* [Daynès et al. 1997] is therefore met syntactically, but perhaps not semantically.

## 4 Conclusions

Concurrency is a fundamental computational tool, it is central to the Java programming language, and it is implicit to the transactional model of computation that underpins most persistent systems. Yet despite the sophistication of modern database systems and programming languages with respect to concurrency, orthogonally persistent systems, which bring together database and programming language principles, are underdeveloped in this respect. This underdevelopment can be explained in part by the problem of the intermix of concurrency and persistence, which is largely avoided by programming languages through the absence of persistence mechanisms, and is limited in database systems by the relatively restrictive notions of concurrency and persistence they embody.

The requirement that all computation over persistent data occur within a transactional context and the impossibility of invoking ACID transactions from within transactions produces a conundrum for those wishing to develop transactional orthogonally persistent systems (where all computation may, by definition, be over persistent data).

On the other hand, allowing computation outside of any transactional context introduces the challenge of providing practical and efficient means of identifying consistent cuts through concurrent computation in order to facilitate coherent stabilization of data. The absence of any published account of orthogonally persistent systems that have provided an efficient and general solution to the problem through either transactional or 'open' approaches suggests problems for the future of orthogonal persistence as a platform for general computation.

Having identified a stumbling block for the development of generalized concurrency support in orthogonally persistent systems, we have outlined *commit and spawn* as a simple means of unlocking the conundrum facing the transactional approach to the problem. While on its own *commit and spawn* does not deliver a particularly usable programming model, its combination with the chain transaction model [Gray and Reuter 1993] leads to the *chain and spawn* transaction model, which facilitates the maintenance of control-flow across ACID transactions and so delivers a *workable* model. This solution, despite its limitations, indicates that the problem is not insoluble, and therefore provides some hope for a future, more general, resolution of this quite fundamental problem.

## Acknowledgements

The authors wish to thank Robin Stanton, Graham Kirby, Ron Morrison, and David Sitsky for helpful suggestions and comments on this work.

## Bibliography

ANFINDSEN, O. J. 1997. *Apotram - an Application-Oriented Transation Model*. PhD thesis, University of Oslo. Available online at: `http://www.fou.telenor.no/apotram`.

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent java. *SIGMOD Record 25*, 4 (Dec.), 86–75.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

CAREY, M. J. AND DEWITT, D. J. 1996. Of objects and databases: A decade of turmoil. In T. M. VIJAYARAMAN, A. P. BUCHMANN, C. MOHAN, AND N. L. SARDA Eds., *VLDB'96, Proceedings of the 22th International Conference on Very Large Data Bases* (Mumbai (Bombay), India, Sept. 3–6 1996), pp. 3–14. Morgan Kaufmann.

CHRYSANTHIS, P. AND RAMAMRITHAM, K. 1994. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems 19*, 3 (Sept.), 450–491.

CODD, E. F. 1970. A relational model for large shared databanks. *Communications of the ACM 13*, 6 (June), 377–387.

DAYNÈS, L., ATKINSON, M. P., AND VALDURIEZ, P. 1997. Customizable concurrency control for Persistent Java. In S. JAJODIA AND L. KERSCHBER Eds., *Advanced Transaction Models and Architectures*, Chapter 7. Kluwer.

DEARLE, A., DI BONA, R., FARROW, J., HENSKENS, F., LINDSTRÖM, A., ROSENBERG, J., AND VAUGHAN, F. 1994. Grasshopper: An orthogonally persistent operating system. *Computing Systems 7*, 3 (Summer), 289–312.

DEARLE, A. AND HULSE, D. 1995. On page-based optimistic process checkpointing. In *Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems* (Lund, Sweden, Aug. 14–15 1995), pp. 24–32. IEEE.

ELMAGARMID, A. K. Ed. 1992. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann series in Data Management Systems. Morgan Kaufmann, San Mateo, CA, U.S.A.

GARTHWAITE, A. AND NETTLES, S. 1996. Transactions for Java. In M. JORDAN AND M. ATKINSON Eds., *First International Workshop on Persistence and Java* (Drymen, Scotland, Sept. 16–18 1996).

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo.

HÄRDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.

HULSE, D. 1998. *Store Architecture in a Persistent Operating System*. PhD thesis, University of Adelaide.

JALILI, R. AND HENSKENS, F. 1995. Using directed graphs to describe entity dependency in stable distributed persistent stores. In H. EL-REWINI AND B. D. SHRIVER Eds., *Proceedings of the 28th Hawaii International Conference on Systems Sciences*, Volume 2 (Hawaii, U.S.A., Jan. 3–6 1995), pp. 665–674. IEEE Computer Society Press.

MATTHES, F., SCHRÖDER, G., AND SCHMIDT, J. 1995. Tycoon: A scalable and interoperable persistent system environment. In M. ATKINSON Ed., *Fully Integrated Data Environments*. Springer-Verlag.

MORRISON, R., BROWN, A. L., CONNOR, R. C. H., CUTTS, Q. I., DEARLE, A., KIRBY, G. N. C., AND MUNRO, D. S. 1996. The Napier88 reference manual (Release 2.2.1). Technical report, University of St Andrews. Available online at: `http://www-ppg.dcs.st-andrews.ac.uk/Publications/`.

MOSS, J. E. B. 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, U.S.A.

SCHWARZ, R. AND MATTERN, F. 1994. Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distributed Computing 7*, 3, 149–174.

SILBERSCHATZ, A. AND ZDONIC, S. 1996. Strategic directions in database systems—breaking out of the box. *ACM Computing Surveys 28*, 4 (Dec.), 764–778.

STEMPLE, D. AND MORRISON, R. 1992. Specifying flexible concurrency control schemes: An abstract operational approach. In *15th Australian Computer Science Conference* (Hobart, Australia, Jan. 1992), pp. 873–891.

STONEBRAKER, M. AND MOORE, D. 1996. *Object-Relational DBMSs : The Next Great Wave*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann.