[Moss 81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachussets Institute of Technology, April 1981.

[Munro *et al*. 94] D.S. Munro, R.C.H. Connor, R. Morrison, S. Scheuerl and D.W. Stemple. *Concurrent Shadow Paging in the FLASK Architecture* in M.P. Atkinson, V. Benzaken, and D. Maier, editors, Persistent Object Systems (Proc. of the Sixth Int. Workshop on Persistent Object Systems), Workshops in Computing, Tarascon, Provence, France, September 1994. Springer-Verlag in collaboration with the British Computer Society, pages 16--42.

[OSF 95] Open Software Foundation. *Java Mobile Code -- A White Paper*, December 1995.

[Pu *et al*. 88] C. Pu, G.E. Kaiser, and N. Hutchinson. *Split-Transactions for Open-Ended Activities* in Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pages 26--37, Los Angeles, California, August 1988.

[Rashid *et al*. 89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. *Mach: A Foundation for Open Systems* in Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2), September 1989.

[Stemple and Morrison 96] D. Stemple and R. Morrison. *Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach* in chapter 1.4.1, [Atkinson 96].

[White and DeWitt 92] S. White and D. DeWitt. *A Performance Study of Alternative Object Faulting in Pointer Swizzling Strategies* in Proc. of the 18th Int. Conf. on Very Large Database, pages 419--431, Vancouver, British Columbia, Canada, 1992.

[Gosling and McGilton 95] J. Gosling and H. McGilton. *The Java Language Environment -- A White Paper*. Sun Microsystems Computer Company, May 1995.

[Gosling *et al*. 96] J. Gosling, B. Joy and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gray and Reuter 93] J. Gray and A. Reuter. *Transaction Processing: Concept and Techniques*. Morgan-Kaufman, 1993.

[Haines *et al*. 96] N. Haines, D. Kindred, J. Morisset, S.M. Nettles, and J.M. Wing. *Composing First-Class Transaction* to appear in ACM TOPLAS, 1996.

[Härder and Rothermel 93] T. Härder and K. Rothermel. *Concurrency Control Issues in Nested Transactions* in VLDB Journal, 2(1):39--74, 1993.

[Jordan *et al*. 95] M. Jordan and M. Van De Vanter *Software Configuration Management in an Object-Oriented Database* in Proc. of the USENIX 1995 Conference on Object-Oriented Technologies, Monterey, California, June 26-29, 1995.

[Kaiser and Pu 95] G.E. Kaiser and C. Pu. *Cooperative Transaction for Multiuser Environments*. chapter 20, pages 409--433, in W. Kim, editor. Modern Database System -- The Object Model, Interoperability and Beyond. Addison Wesley/ ACM Press, 1995.

[Kaiser and Pu 92] G.E. Kaiser and C. Pu. *Dynamic Restructuring of Transactions*, chapter 8, pages 266--295, in "Database Transaction Models for Advanced Applications", Data Management Systems series, Morgan-Kaufman, San Mateo, CA, 1992. A.K. Elmagarmid, editor.

[Kim *et al*. 88] W. Kim, N. Ballou, H. Chou, J.F. Garza, and D. Woelk. *Integrating an Object-Oriented Programming System with a Database System* in OOPSLA '88 Int. Conf., pages 142--152, September 1988.

[Krablin 85] G.L. Krablin. *Building Flexible Multilevel Transactions in a Distributed Persistent Environment* Chapter 14,pages 213--234, in Atkinson, M.P. and Buneman, O.P. and Morrison, R., Data Types and Persistence -- Proceedings of the First Workshop on Persistent Object Systems, Appin, Scotland, August 1985.

[Martin and Ramamritham 95] C.P. Martin and K. Ramamritham. *ARIES/RH: Robust Support for Delegation by Rewriting History*. Technical Report 95--51, University of Massachusetts, Amherst, Massachusetts, June 1995.

[Mashburn and Satyanarayanan 94] H.H. Mashburn, and Satyanarayanan. *Recoverable Virtual Memory*. RVM Release 1.3, CMU, January 1994.

[Matthes and Schmidt 94] F. Matthes and J.W. Schmidt. *Persistent Threads* in Proceedings of the Twentieth International Conference on Very Large Databases, pages 403--414, Santiago, Chile, 1994.

[Morrison *et al*. 96] R. Morrison, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, D.S. Munro and M.P. Atkinson. *The Napier88 Persistent Programming Language and Environment* in chapter 1.5.3, [Atkinson 96].

[Moss 92] J.E.B. Moss. *Working With Objects: To Swizzle or Not to Swizzle?* IEEE Transactions on Software Engineering, 18(8):657--673, August 1992.

Bibliography

[Atkinson 96] M.P. Atkinson, editor. *Fully Integrated Data Environments* Springer-Verlag 1996.

[Atkinson *et al*. 96] M.P. Atkinson, L. Daynès and S. Spence. *PJava Design 1.2* Working Document available via http://www.dcs.gla.ac.uk/~susan/pjava/

[Atkinson and Morrison 95] M.P. Atkinson and R. Morrison. *Orthogonal Persistent Object Systems* in VLDB Journal, 4(3), pp319-401, 1995.

[Atkinson *et al*. 88] M.P. Atkinson, O.P. Buneman and R. Morrison. *Binding and Type Checking in Database Programming Languages* in Computer Journal, 31(2), pp99-109, April 1988.

[Atkinson *et al*. 83] M.P. Atkinson, K.J. Chisholm, W.P. Cockshott, and R.M. Marshall. *Algorithms for a Persistent Heap* in IEEE Software, Practice and Engineering, 13(3):259--272, March 1983.

[Bailey 96] P. Bailey. Personal communication, University of Glasgow, February 1996.

[Bancilhon *et al*. 85] F. Bancilhon, W. Kim, and H.F. Korth. *A Model of CAD Transactions* in Proc. of the 11th Int. Conf. on Very Large Database, pages 25--33, 1985.

[Chelliah and Ahamad 93] M. Chelliah and M. Ahamad. *Operating System Mechanisms for Distributed Object-Based Fault-Tolerant Computing* Technical Report GIT-CC-92-23, School of Information and Computer Sciences, Georgia Institute of Technology, Atlanta, Georgia, October 1993.

[Chrysanthis and Ramamritham 94] P. Chrysanthis and K. Ramamritham. *Synthesis of Extended Transaction Models using ACTA* in ACM TODS, 19(3), pp450-491, September 1994.

[Chrysanthis and Ramamritham 91] P.K. Chrysanthis and K. Ramamritham. *A Formalism for Extended Transaction Models* in Proc. of the 17th Int. Conf. on Very Large Database, pages 103--112, Barcelona, September 1991.

[Cockshott *et al*. 84] W.P. Cockshott, M.P. Atkinson, K. Chisholm, P. Bailey, and R. Morrison. *Persistent Object Management Systems* in IEEE Software, Practice and Engineering, 14:49--71, 1984.

[Daynès 95] L. Daynès. *Conception et réalisation de mécanismes flexibles de verrouillage adaptés aux SGBDO client-serveur*. Ph.D. thesis, Université Pierre et Marie Curie (Paris VI -- Jussieu). 29 September 1995.

[Daynès and Gruber 94] L. Daynès and O. Gruber. *Efficient Customizable Concurrency Control using Graph of Locking Capabilities* in M.P. Atkinson, V. Benzaken, and D. Maier, editors, Persistent Object Systems (Proc. of the Sixth Int. Workshop on Persistent Object Systems), Workshops in Computing, Tarascon, Provence, France, September 1994. Springer-Verlag in collaboration with the British Computer Society, pages 147--161.

[Daynès *et al*. 95] L. Daynès, O. Gruber, and P. Valduriez. *Locking in OODBMS clients supporting Nested Transactions* in Proc. of the 11th Int. Conf. on Data Engineering, pages 316--323, Taipei, Taiwan, March 1995.

[Garcia-Molina and Salem 87] H. Garcia-Molina and K. Salem. *SAGAS* in Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pages 249--259, 1987.

5. Conclusions and Future Work

The design for the system PJava has been presented. It is an alternative platform for the Java language with provision of completely orthogonal persistence for data, meta data (classes) and code (methods). One of the goals of PJava is to attract wider use of this form of persistence and another is to support a wide range of applications.

To achieve both of these goals a simple transactional model is provided by default and composable primitives are provided for describing complex transactional behaviour. Normally, these more complex behaviours will be programmed by specialising the class `TransactionShell`. As these shells, including their specialisations, present a standard external API, the majority of application programmers should be able to code, even using ready made sophisticated transaction models, without being aware of details of transaction implementation.

The syntax of Java is unchanged and only minimal changes have been made to the semantics of certain operations. In particular, there is no reduction in the safety afforded by Java. At present, we envisage PJava being used mainly for server software and not for applets. For example a crossword server might be written in PJava and a crossword presenter applet in Java. When the presenter is used, the remote user can select a crossword and then view clues, etc. The user may also ask the presenter to preserve a partial solution and return to it later. The crosswords and clues would be stored at the server in the stable store. As users progressed with crosswords their solution increments would be shipped back to the stable store to preserve them. After some failure, or when returning on another occasion, the user could resume an earlier puzzle and its partial solution would be retrieved from the store and shipped to the presenter. The communication between the Java applet and the PJava server could be any protocol available in Java.

The implementation of the prototype, $PJava_0$, which presents only the default transactional behaviour and omits several planned optimisations, is in progress. It is anticipated that a preliminary report on experience with this prototype and on its performance, will be available at the POS-7 workshop.

The prototype will be followed by a succession of releases leading to the functionality described here. Experience with earlier versions will be used to identify the need for design changes. Testing applications during this phase will be the development of CASE tools and some geographic information system (GIS) examples. Measurements with these loads will be used to identify which optimisations to focus on. An early task will be the development of a more integrated store management scheme, including disk garbage collection.

Design issues that are not easily resolved concerning the semantics of PJava include:

- when to re-initialise static variables in persistent classes;
- whether to take note of class relationships when organising static variable initialisation;
- whether to make top-level transactions of PJava analogous to UNIX processes (to maintain the analogy, each transaction would need to manufacture its own set of static variables to emulate the behaviour with respect to static variables of say two separate C++ programs run as transactions against a common database);
- whether to persevere with a combination of transactions and threads or to seek one construct that serves both purposes; and
- how to choose a good set of transaction primitives and implement them with minimal performance costs.

Our presently preferred solutions to these design issues are being implemented and will be tested for a few applications. It is hoped that other forms of persistent Java will be developed to exploit the novel opportunity of a language that is safe and which looks likely to become popular and commercially important. This will permit comparison of the various outcomes from these design choices, but much more importantly, it may enable many more application teams and industries to enjoy the benefits of persistent programming.

6. Acknowledgements

This work was supported by a grant from Sun under their collaborative awards program. It builds on research and technology developed in the European Community Basic Research Action $FIDE_2$, 6093.

To avoid these repeated object faults and internal residency checks, classes will be treated specially. During a stabilisation, the whole of the class including internal objects and all of the classes and objects on which it intimately depends will be marshalled together and shipped as one store object. When faulted in, the reverse process reconstructs the internal data structures and the indirect links via handles to all these closely associated objects. A generalisation of this, with a slightly more extensive set of commonly used classes and objects, will be used to achieve fast program initialisation by clustering the extensive set in the stable store and loading all of these with the `PJavaStore` object and class.

3.4 Promoting New Objects

When a PJava program stabilises, new objects in the garbage collected heap are promoted. First, they are found by performing an application of Java's garbage collection scanner from the handles in the ROT that are identified as having been updated since the last stabilise. These are then copied into the Object Cache and allocated handles in the ROT.

The next step is to allocate them PIDs at the end of the stable store, i.e. beyond the point where the size of the stable store, recorded in the `PJavaStore` object, says the store ends. They can then be written directly to the disk in a single transfer, by-passing RVM. There is no need to put any information in the log at this stage. If a crash occurs before the end of the next stage, the recovery mechanism will behave as if they had never been written and the store was still its old size with those PIDs unallocated, as the updated `PJavaStore` object has not been written back.

The PJavaStore object is now updated with the new size, highest PID, etc. and an RVM transaction is used to write back all of the modified objects in the Object Cache. This will be logged. If it commits, the whole stabilise succeeds and the new objects are visible. If it aborts or a crash occurs, then the modifications to these objects will not be visible at the next restart. Since these include the change to the store size, the new objects will also be ignored. During a database load there is a very high rate of creation of new objects. The machanism above avoids writing to the log and avoids complex allocation for these new objects. We believe that, in consequence, database load will be significantly accelerated with no loss of stable store integrity. Other features of the implementation will be discussed in a later paper which will include performance measurements for the mechanisms introduced above.

4. Related Work

{This section must be reconsidered and revised before *final* submission.}

The early work on PS-algol [Atkinson *et al.* 83] focused on providing orthogonal persistence and a default model of transaction and stabilisation similar to the default in PJava. Later, concurrent threads were provided in CPS-algol [Krablin 85]. The potential of primitives to support flexible transactions was discussed in that paper. The FLASK system of Napier 88 continued the tradition of persistent threads as a foundation for transactions [Munro *et al.* 94]. A similar scheme for Tycoon was reported recently [Mathes and Schmidt 94]. These systems did not support pre-emptive thread scheduling, and so did not require thread safe object faulting.

Several other groups have considered the provision of a flexible model of transactions. Chrysanthis and Ramamritham [Chrysanthis and Ramamritham 91] began with a formal model and have recently commenced a study of implementation mechanisms [Martin and Ramamritham 95]. Morrison and Stemple have proposed the partially automated synthesis of programs that have primitive synchronisation and stabilisation actions applied in the context of a store that accepts concurrent overlapping updates [Stemple and Morrison 96, Munro *et al.* 94]. A more detailed survey and analysis of the utility of our proposed transactional primitives in forming a wide range of transactions and a preliminary investigation of their feasibility has previously been presented [Daynès 95, Daynès *et al.* 95].

Perhaps the system closest in philosophy to this work is the persistent version of standard ML produced by the Venari project [Haines *et al.* 96]. Here, stabilisation of a snapshot is also the foundation of transactions. The clouds project also provides a set of primitives from which a variety of transaction models can be synthesised [Chelliah and Ahamad 93]. These systems use different primitives and work with languages that differ. Insufficient experience is available to support comparison at present.

There are several projects underway to produce persistent forms of Java. As far as we know this is the only one with both orthogonal persistence and persistence determined by reachability. Whether they aim for similar levels of transactional flexibility is also unknown.

from those for transient objects. These handles are collected together in the resident object table (ROT).

- The Object Cache is bootstrapped by loading the `PJavaStore` class and object instances. Thereafter, other objects are faulted into the cache whenever a PID that cannot be found in the ROT is dereferenced.
- Objects are promoted from the garbage collected heap and allocated new handles in the ROT, as well as PIDs, during the first part of a stabilise. They are then written to the stable store on disk.
- When an executing Java program requires a class that is not yet resident, a lookup in the Persistent Class Directory (PCD) is performed first. The standard class loading mechanism of Java is used only if the class is not found in the PCD.
- The Buffer Pool is implemented on top of RVM [Mashburn and Satyanarayanan 94] using a no steal policy (i.e. dirty pages are not written back until commit)[7]. The dirtying of pages only begins during the final phase of stabilisation so that this will not pin pages for long. Objects are copied from the Buffer Pool to the Object Cache as they are faulted in and are copied back to the Buffer Pool during stabilise if they have been updated. This object copying policy is chosen to increase the density of active objects in the Object Cache [Cockshott *et al*. 84, White and DeWitt 92, Moss 92, Kim *et al*. 88].

3.2 Swizzling & Faulting

Whilst from the Java application programmer's viewpoint, objects appear to reference each other directly, under the surface a level of indirection is used. We have been able to exploit this existing indirection via handles in our faulting and swizzling mechanism. We have adopted eager indirect swizzling [Moss 92] for objects and lazy swizzling for arrays and methods. A clustered technique, described below, is used for class data. The rationale behind these decisions is:

- objects are usually small, and therefore eagerly swizzling and allocating handles for their pointers is not expected to be expensive;
- arrays are potentially large and could require excessive handle space and processing time — it remains to be seen whether most accessed arrays then have all their elements accessed; and
- the code for operations within class data can be left unperturbed if we reconstruct the Java data structures as we fault a class before continuing.

When a program tries to *unhandle* a handle we need to test for residence and potentially fault in a non-resident object. As soon as an object is faulted into the Object Cache, all its PIDs are swizzled to point to handles in the ROT. If the referend hasn't been partially swizzled this far already, a new handle is added to the ROT. Each handle in the ROT holds the corresponding PID. Thus the swizzling horizon is maintained in the handles rather than in the objects.

For swizzling to be thread safe it is necessary to ensure that two threads do not concurrently fault in the same object as there would then be two images of that object potentially being updated independently. This is achieved by an optimistic policy. It is assumed that it will be relatively rare for two threads to attempt to unhandle the same handle simultaneously. The residency test is therefore performed once without any locking. Only if it is found that the object is non-resident is the object-faulting mutex acquired and the test re-performed. If the thread still finds the object non-resident it will be the only thread faulting that object. In this way all the residency checks for resident objects avoid the mutex overhead.

3.3 Clustered Class Loading

Observing the Napier88 system [Morrison *et al.* 96], we find that the same group of objects[8] are loaded at the start of every program. Without evasive action, this phenomena will be repeated in PJava, as the `PJavaStore` object and class and all the objects and classes on which they depend to perform the start up are brought in. We also require that when a class is loaded, all of its internal objects and related objects be loaded, so that we can avoid residency checks and object faults within the class `Class` implementation.

---

7. This is imposed by RVM which we are using for $PJava_0$.

8. About 670 according to some recent measurements [Bailey 96].

3. Implementation Issues

Space only permits a very brief sketch of our implementation plans. PJava requires changes to the Java abstract machine (JAM), to the Java run-time and to the special classes provided with Java listed above (section 1.2). In addition we need to implement classes: `PJavaStore`, `TransactionShell`, `UpdateBookKeeper`, `AccessBookKeeper`, `SnapShot` and `LockingCapability`. These are implemented partly in Java and partly as C extensions to the Java run-time. A tutorial on the general mechanisms required to incrementally load and stabilise a persistent store using persistence by reachability is given in [Atkinson and Morrison 95]. We use the same nomenclature here.

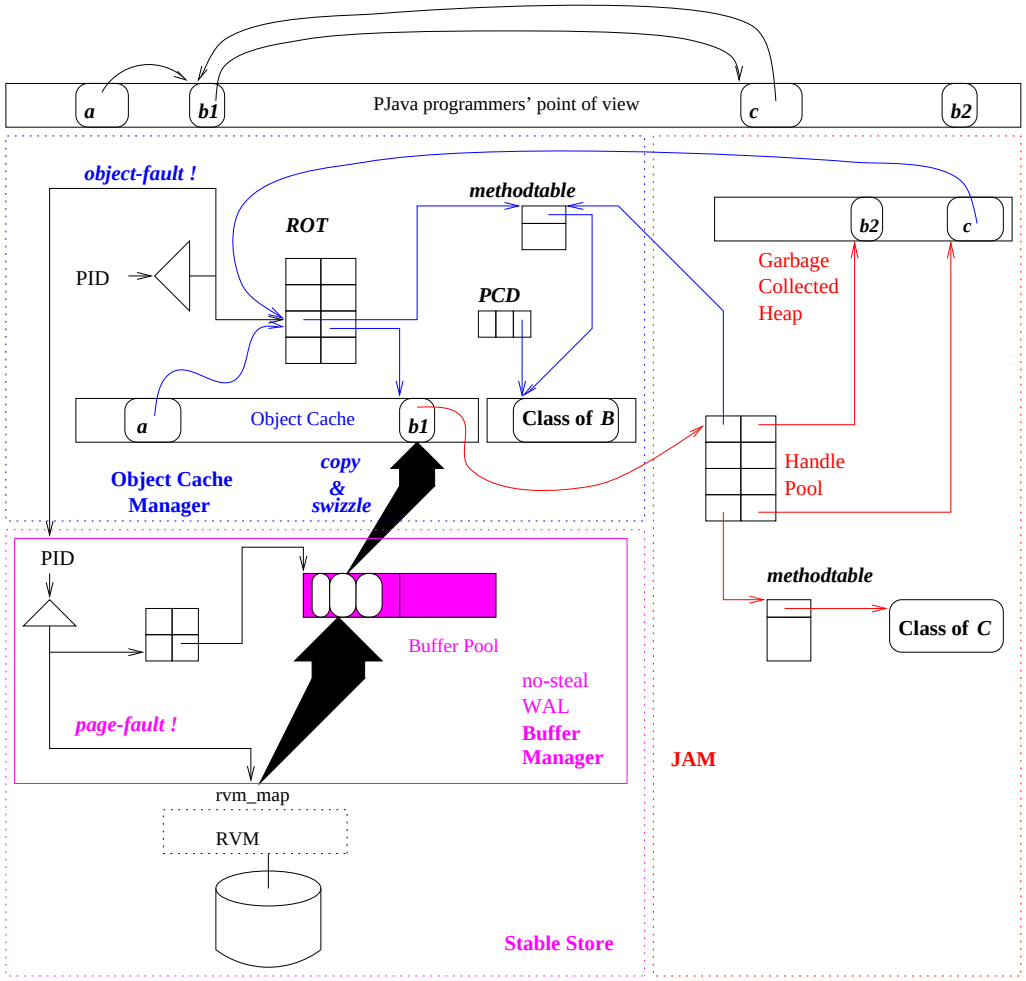

Figure 2: PJava$_0$'s Object Cache Architecture

3.1 Architecture of PJava$_0$

The overall architecture of PJava$_0$ is represented in Figure 2. The main features to notice are given below.

- The existing JAM has been disturbed minimally, in particular its garbage collected heap and the handles it uses to access transient objects on that heap retain the format they have in standard Java. Also, the changes to the interpreter are minimal.
- All of the persistence activities are handled via structures in the Object Cache and the Buffer Pool. Every object in the Object Cache is addressed by the JAM via handles that are distinct

able an identified set of the objects that have been accessed by a `Thread`. A typical use is to specify a set of objects whose locks and / or updates are to be delegated to another transaction.

`Snapshot` These are produced by `UpdateBookKeeper`s which retain their ordering. It is then possible to apply a method of a `Snapshot` to perform a set of updates that restore the saved state. This implies application of any intervening `SnapShot`s. After such an undo operation, application of the restore method of a later `SnapShot` would correspond to a redo operation. The `Snapshot` itself may be included in the set made durable as it is formed, or omitted from this set. `Snapshot`s may be disabled. If a Snapshot is made durable and then disabled, the permanence of an ACID transaction is achieved. `SnapShot`s are available for programmers wishing to build redo and undo in a command set as well as for transactional purposes.

`LockingCapability` These can be used within a transaction to represent locks accumulated by transactions. They provide a mechanism for specifying exclusion models other than the default of one writer or many readers and delegation. Delegation of locks can be used in a similar way to delegation of updates, in the definition of nested transactions for example. Normally delegation of updates and locks occur in step, but they may be separated; for example, in chained transactions a transaction that commits will pass its locks on, but not its updates which will have been stabilised [Gray and Reuter 93].

Figure 1 illustrates how two of these constructs, `SnapShot` and `UpdateBookKeeper` may be combined. Update operations U_i and SnapShot operations Sn_i are shown for an execution history. In addition, a `SnapShot` stabilise is shown as a large arrow. It is possible to disable a sequence of `SnapShot`s as a stabilisation is performed. The initial SnapShot, Sn_0, is formed implicitly as the `UpdateBookKeeper` is created and is the default point for a roll back. Rollbacks are automatically performed whenever a thread bound {e.g. by `B(ubk,th)` } to this `UpdateBookKeeper` fails to handle an exception. After `B(ubk,th)` every update made by `th` is logged by `ubk`, including updates to the threads stacks, to static and local variables, etc.

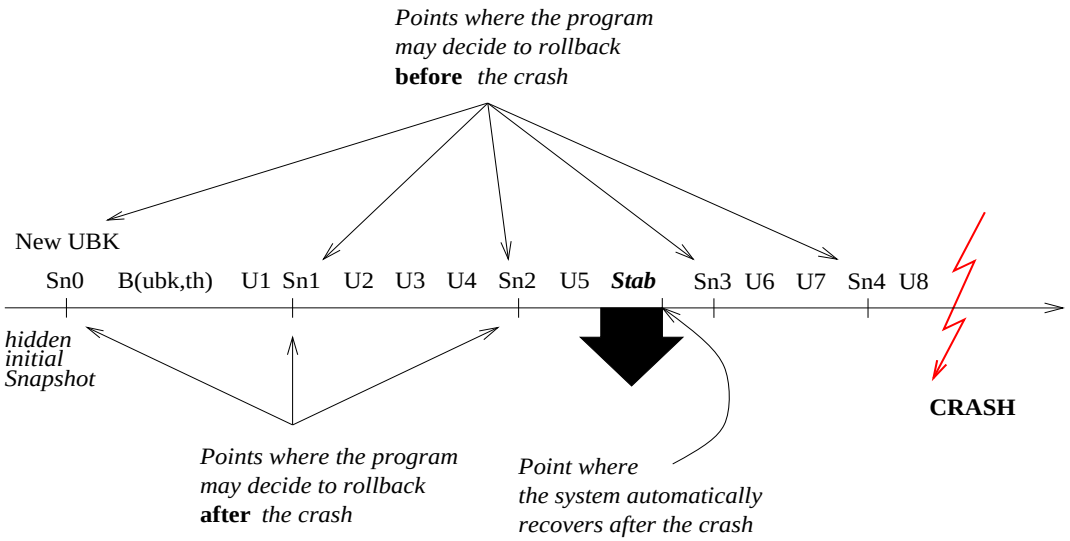

Figure 1 : a Program Execution Trace to show `UpdateBookKeeper`and `Snapshot` operations

This internal API is more complicated than the external API which we expect to be sufficient for most of the programmers who are not content with the default transactional model. The goal is to allow application programmers an easily negotiated ramp of increasing transactional sophistication. Only evaluation of the use made of these facilities by independent application programmers will show whether the separation of the default behaviour, the external and internal APIs of `TransactionShell`s has paid off. Reasonably efficient mechanisms to support the individual primitives are known, but our intuition that it is feasible to compose them has yet to be tested.

The present design supports sibling and parent-child concurrency [Härder and Rothermel 93], it also defaults to non-escaping children with durability achieved only by top-level transactions, which may themselves be nested. However, the primitives provided allow a programmer to arrange for earlier durability and for escaping transactions. These primitives are also capable of supporting various forms of long-running transactions, such as SAGAS [Garcia-Molina and Salem 87].

To achieve this flexibility, a `TransactionShell` class is provided. Specialisations of this class hold definitions of the transaction management policy to be used for transactions run under their control. Instances of these specialisations run individual transactions according to their policy. Transactions under different policy regimes may be executing concurrently with predictable behaviour. Two specialisations of `TransactionShell` are provided initially.

`OLTPTransaction` provides the typical flat ACID transaction model of many DBMS. It expects a brief transaction, suppresses the transaction operations of inner transactions. It makes all of its updates durable and releases all its locks at the terminal commit; and

`NestedTransaction` provides nested transactions with both parent-child and sibling concurrency. These transactions make their updates durable when the outermost transactions commit; inner transactions undo on abort and delegate their updates and locks to their parents on commit.

2.5.3 The External Transactional API

At present the division of the available transactional facilities into an “external” and an “internal” API is a matter of convention. In fact, all the transactional facilities are available to all programmers. We made need to review this reliance on convention.

It is intended that the standard, external transactional API provided by the instances of `TransactionShell` and its specialisations should satisfy most application programmers. This API hides implementation details from application programmers who program using a few `TransactionShell` methods. These methods allow the programmer to associate any object that satisfies the `Runnable` interface with an instance of a `TransactionShell`. Whenever the code in that `Runnable` object is activated, it is constrained to conform to the transaction model implemented by that `TransactionShell`. Programmers can then use its methods to:

- start its execution,
- enquire about the state of the execution (`running`, `aborted` or `completed`) and
- claim the results (synchronously or asynchronously).

By default transactions are run synchronously, that is, the transaction that performs the start operation is suspended until the initiated transaction completes. It is also possible to launch transactions asynchronously, making further methods useful. These can stop or kill a transaction and wait on an individual or group of transactions to achieve a certain state (all completed, one aborted, etc.).

2.5.4 The Internal Transactional API

Programmers who wish to define new transactional models by defining a specialisation of `TransactionShell` must use a more sophisticated internal transactional API which utilises a set of transaction building primitives.

`UpdateBookKeeper` Instances of this construct track the set of objects that have been updated by transactions associated with them. The internal API provides methods for creating `UpdateBookKeeper`s and for binding `Thread`s to a particular `UpdateBookKeeper`. `Thread`s are always associated with precisely one book keeper. The two most important functionalities of `UpdateBookKeeper`s are their ability to make a `Snapshot` and their ability to delegate some or all of their updates to another `UpdateBookKeeper`[6]. For example, this capability is used to pass updates to parents in nested transactions, so that they may later make durable, delegate or undo these updates. The `Snapshot` creation captures the state of this `UpdateBookKeeper`’s updated set of objects as a basis for undo and redo operations. This state can optionally be made durable as it is captured, thus guaranteeing its continued availability for two phase commit, long transactions, etc.

`AccessBookKeeper` These are similar to `UpdateBookKeeper`s except that they do not have the capability to make `Snapshot`s. They may be used where it is necessary to have avail-

6. The concept of delegation is formally described in [Chrysanthis and Ramamritham 94].

used from some other package that is also using transactions directly and they must compose correctly, i.e. with understandable and reasonable semantics.

Initially, we considered restrictive regimes, such as constraining a transaction to a single thread. At first sight, this appears to simplify the PJava programming model, but all the possible restricted models suffer the following drawbacks:

- they prohibit code that it seems reasonable to allow; and
- they will surely generate problems when independently written code is combined.

As an example of the former, consider a transaction that is required to obtain an edit to some complex value. That value might be displayed in several forms simultaneously in different windows, to allow the end-user to make changes to one form and visualise them in all views — e.g. a 3D manipulation or a graphical and textual view of a semantic data model schema. The edits and feedback would then be organised in terms of several co-operating threads. But the whole edit would be within one transaction. On the other hand, a thread supporting a long editing session might run this as a series of transactions, so that the end-user can choose when to secure the work so far.

Similarly, we expect that the code in one package may utilise threads and in another utilise transactions, so that composition of PJava programs requires that a transaction may contain several threads and a thread may contain a sequence of, possibly nested, transactions.

This issue is not fully resolved. The principle of parsimony in language design appears to be violated. Both transactions and threads support concurrent operations. There is a nuance that the thread model is more concerned with programming where programmers have an overall understanding of the set of co-operating threads and their interactions, whereas the transactional model is more concerned with programming where the programmers are not aware of all the other simultaneous activities. However, as programs get larger, programmers are less able to visualise thread interactions and so have to write thread-safe code protected by `synchronize` or explicit locks anyway. Similarly, as longer transactions are supported, programmers need to become more supportive of constructive interaction between transactions. It may eventually prove possible to elide the thread and transaction constructs. However, our goal of minimising perturbations to existing constructs currently results in two constructs.

Perhaps, as the number of application programmers grow, and consequently the number of independently-written concurrent threads grows, it will become mandatory to make all code thread-safe. At which point, the replacement of threads by transactions will have little execution cost and will relieve programmers of managing most locks and synchrony.

All code in PJava is run within some transaction. If an encompassing transaction has not been explicitly initiated, then it is the implicit transaction associated with the whole program (see 2.4). This is also a contentious issue. It means that there is some write barrier monitoring updates for all threads at all times. There are clearly threads for which this is useless work, for example a thread that monitors the system clock and displays the time — you would not wish to restore its previous state on a restart. However, so far, we haven't been able to devise a scheme that will allow non-transactional threads that doesn't also potentially breach the safety and consistency requirements.

2.5.2 TransactionShells

It is apparent that a wide range of transactional behaviours are useful in different applications [Bancilhon *et al.* 85, Garcia-Molina and Salem 87, Pu *et al.* 88, Chrysanthis and Ramamritham 91, Kaiser and Pu 92, Gray and Reuter 93, Kaiser and Pu 95]. The support for transactions in PJava is an experiment to test whether a wide range of these transactions can be made available by providing a small set of primitives that can be composed to achieve the transactional behaviours. The logical properties of these compositions have already been explored [Daynès 95, Daynès and Gruber 94, Atkinson *et al.* 96]. Only preliminary studies of their performance in real applications have been undertaken so far.

Because of the recursive structure of computations in a modern language like Java, some form of nested transaction [Moss 81, Härder and Rothermel 93] will be required as a result of program composition and calls of code that contain transactions. There is, however, a choice of nested transaction semantics, for example:

- the effects of inner transactions may be propagated to parents and only made durable when the top transaction commits or may be made durable immediately;
- only sibling concurrency may be supported or parent-child concurrency may be available, and
- children may be constrained within the lifetime of their parent or may be allowed to escape.

PJava supports both forms. By default, `static` variables are all persistently `static`. This supports well constants described in Java as `static final` variables. We contemplated allowing programmers to register in `PJavaStore` which classes should be transiently `static`.

This led to an unresolved concern as to whether the relationship between classes should be reflected in the static re-initialisation sequence. For example, suppose that classes `B` and `D` are specialisations of class `A` and that class `C` is a specialisation of class `B`. Suppose also that the application programmer includes `C` in the register of transiently `static` classes. Should this have any implications for classes `B`, `A` and `D`? For example, should classes `B` and `A` also be transiently `static` and should such inconsistency result in automatic registration of `B` and `A` or in an exception being raised? Such considerations led us to let the application programmers decide using the call-backs described next.[5]

2.4.4 Recovery and Initialisation Call-backs

PJava allows the application programmer to specify recovery and initialisation behaviour. As in Java, a PJava execution ultimately calls a `main` method of some class, often specified in the command-line. Before this call, PJava will run recovery code if the previous execution of PJava terminated unsuccessfully or there was a system or media failure during the final stabilise. After that, and before `main`, PJava will run an arbitrary sequence of call-backs supplied by the PAS programmer. PAS is emphasised here because those call backs must have been supplied during some previous successful execution.

There are two ways of setting the sequence of restoration call-backs. Both involve calling methods of `PJavaStore`. The method `setGlobalCallbacks(RestartCallback[] rc)` allows the application programmer to provide a sequence, `rc`, of call-backs, to replace any previous sequence. These call-backs are called by the PJava runtime in the order supplied in the array. The method `setClassCallbacks(RestartCallback[] rc, Class c)` provides a sequence of call backs for the specified class. These will be applied in order in any execution in which that class is loaded, after the global call-backs. Methods are provided to obtain the previous settings of these sequences.

Call-backs may be any implementation of the `RestartCallback interface`. They are called with a parameter which indicates whether PJava is performing a `restore`, a normal `restart` or a `retry`. A `restore` occurs after an incomplete stabilise or media failure. A `retry` occurs if, on a previous attempt at this bootstrap a call-back raised an unhandled exception or tried to perform an operation not allowed during the call-back sequence, such as `stabilizeAll`.

These call-backs can be used for many purposes, here are a few examples.

- To verify certain consistency constraints still hold in the store, and if they do not, to report this to an administrator, to warn the end-user and abort the execution. Call backs of this kind are likely to accumulate as successive classes are defined and may be generated by design tools.
- To establish the extent to which some previous execution completed a task.
- To create transient structures derivable from the stored structure. This might be “display and visualisation” data derived from the stable store and presented to suit the current user or context.
- To allow refinements of the initialisation policy for `static` variables.
- To permit an application team to develop a more sophisticated virtual machine on top of PJava and to initialise its state while working entirely within PJava.

2.5 Transactions in PJava

Three major issues were considered during the design of a flexible transaction model for PJava:

- how threads and transactions should compose,
- how to present a simple interface for programmers using *predefined* transaction models, and
- what primitives would best support programmers defining *new* transactional models.

2.5.1 Transactions and Threads

In order that code written independently can compose properly, it is necessary that nested transactions are supported. That is, some package may have been written using transactions. That package is then

5. We observe that the field modifier `transient` could be used in conjunction with `static` to indicate a reinitialised static variable.

life.

This has benefits beyond safety but also some costs. The additional benefits include avoiding repeated checks and re-utilising optimisations introduced in earlier executions. The schema, i.e. the collection of class definitions in the store, will grow incrementally as new classes are needed by the application. But other aspects of schema evolution and support platform evolution need special support. For example, a special loader is needed to replace method code with revised code, perhaps to repair a bug. Additional mechanisms will be needed to alter the class definition in other ways: new fields, new methods, changed signatures for methods, etc. These will not be provided in $PJava_0$ and initially PAS implementors will have to achieve these effects by store re-builds.

2.4 Stabilisation, Initialisation and Restoration

2.4.1 Global Stabilisation and Durability

Stabilisation is the process of ensuring that data is durable. After a successful stabilise, the data concerned will be available on restart even if there is a subsequent software failure. It should also be available after media failure, but this depends on the number of redundant copies of the data and the extent of the media failure. In this section only the default form of stabilisation, global stabilisation, will be discussed. Partial stabilisation is via `SnapShots` (see 2.5).

A call of `PJavaStore`'s `stabilizeAll`() method[4] will achieve a global stabilisation. This ensures that all changes since the start of the PJava program execution or the previous stabilise, are made durable. That is all of the objects that have been brought in from stable store and modified, all new objects that are reachable from these modified objects and any classes that are introduced via these new objects are all written atomically to disk. The `registerPRoot` call simply mutates the instance of `PJavaStore`. The `PJavaStore` object is then a modified object and the registered root object is the start of a graph of (possibly new) reachable objects.

2.4.2 Default Transactional Behaviour

The default transactional behaviour of PJava is an implicit call of `stabilizeAll` at the end of each successfully terminating program. A program that fails to handle an exception will terminate unsuccessfully and leave the stable store unchanged; thus an unhandled exception is, by default, equivalent to a transaction abort. Otherwise a PJava program behaves by default as a single transaction against the stable store with which it was associated.

Each object which was preserved by a stabilise holds its state at the last completed stabilise. This state determines the values of all its object variables when it is reloaded in a subsequent execution. However, in PJava an object may also have `static` variables which are effectively variables associated with the instance of that object's class.

2.4.3 Issues Arising with Static Variables

As their name suggests, `static` variables are only initialised once when the class is loaded, i.e. once per program execution in Java. In PJava we see the need for two different semantics for `static` variables:

1. persistently `static` variables, which are initialised when the class was loaded into some PJava program before it migrated to the stable store, or
2. transiently `static` variables, which are re-initialised according to their original initialisation code each time some execution of PJava causes them to be brought from stable store.

An example requiring a variable of the persistently static form would occur in a PAS that issues part numbers. A class `Part` would need a `static` variable in which to hold the last issued part number. A method would increment and issue the part number. If the part number variable were re-initialised each time a PJava program was run that used instances of class `Part`, the same sequences of part numbers would be re-issued repeatedly.

As an example requiring the transiently static form, consider an application that charges for the operations applied to instances of a particular class. Then it must start its variable holding the accumulated charge with the same initial value each time, and not the charge applied to the last customer.

---

4. The "ize" spelling form is used for all identifiers introduced by PJava to be consistent with Java. As a result of habit we use the "ise" form in our text, although both forms are usually correct English spellings.

not be predetermined if PJava is to meet its goal of being general purpose. A compromise is achieved by building-in some mechanisms that could have been modifiable (storing classes in the stable store, for example) and allowing application programmer control for other issues, such as re-initialisation after recovery and the form that transactions should take. However, application programmers are not burdened with supplying this code as default policies are provided. Only when the default does not meet an application's needs does an application programmer need to supply policy defining code.

2.2 Stores and Orthogonal Persistence

A PJava program is normally started in conjunction with an existing and populated stable store specified by an environment variable, command-line parameter, default or similar mechanism. In exceptional circumstances, PJava may create a new, empty, stable store and begin to populate it. Certain objects and classes are needed by the implementation of persistence and are consequently in a stable store by default. Every PJava store will contain one object of the class `PJavaStore` and its `Class` instance containing static variables and the code to manipulate the `PJavaStore` object. PJava$_0$ arranges that this `PJavaStore` object and `Class` instance are accessible to code in the executing PJava program. Other objects and classes are faulted in as needed.

As already mentioned, `PJavaStore` is the root of persistence by reachability in the stable store that contains it. The method `registerPRoot(aName, anObject)` of `PJavaStore` will make `anObject` behave as a persistent root. It can be re-accessed, normally in subsequent executions, by `getPRoot(aName)`. All objects reachable from `anObject` are preserved. All objects reachable from a result of `getPRoot` will be made accessible by object faulting as data structures are traversed using the standard repertoire of Java operations. This mechanism enables a programmer to make any graph of objects and the values they contain have a lifetime extending beyond the current execution.

Objects are retained as long as they are reachable from `PJavaStore` via at least one registered `PRoot`. Their life will be terminated if subsequent program executions make them unreachable. This may occur through assignment to objects in the graph replacing one object reference with another or a `null`. It may also occur because the last connection to the `PRoot` index is severed by a call of `discardPRoot(aName)`. Space will eventually be recovered by garbage collection.

Other operations on the persistent roots are available, to test if they exist, to obtain all their names, etc. Further methods associated with `PJavaStore` are described below. Full details of the `PJavaStore` methods and the whole of the PJava system may be found in [Atkinson *et al*. 96].

2.3 Classes and Code

When an object, `c`, an instance of class `C`, is made persistent it is necessary to ensure that all information that may be needed to operate on `c` that isn't already persistent is simultaneously made to persist. This additional information includes the class `C`, any super-classes of class `C`, their methods, any objects reachable from any of their `static` variables and the classes and current values of all the fields of `c` including those derived from `c`'s super-classes. This is illustrated by the following:

```
class C extends B {
  private D d;
  private E e[];
  static public F f;
  ...
}
C c = new C();
```

If `c` is made persistent then classes `C`, `B`, `D`, `E`, and `F` and the values of the fields `d`, `e` and `f` must also be made persistent. The promotion algorithm must then recurse over these additional classes and values until objects and classes that are already persistent are reached.

An index of the classes that have been made persistent is maintained in each `PJavaStore`. There are methods to interrogate this index and a method may be provided to remove a class that no longer has any instances.

Whenever `c` is used, during some subsequent execution, the runtime system ensures that only the methods associated with `C` and its super-classes are run on `c`. This extends the guarantee of safety that Java gives for one execution to multiple executions. Any checks and verifications that were performed by the compiler and class loader validated the code that was stored with `C`. Since we ensure that only that code can operate on `c` all the original guarantees for the integrity of `c` are perpetuated throughout its persistent

- `Class` — the class of all classes — instances of this class contain the definition of classes and provide access to the method code;
- `Thread` — the class that represents an independent thread of execution that may run concurrently with other threads;
- `Lock` — providing explicit locking when the implicit mutex provided by `synchronized` is insufficient; and
- a variety of classes concerned with IO, screen management, etc.

Other features of Java, such as packages, though relevant to our design and implementation, are not discussed in this paper.

2. Design Issues for PJava

This section presents some of the issues that were encountered during the design of PJava.

2.1 Principles and Desiderata

The three principles already well established for orthogonal persistence [Atkinson *et al*. 83, Atkinson and Morrison 95] are naturally re-applied in this context as follows.

The principle of **orthogonality** states that all values whatever their type have equal rights to persistence — longevity or brevity. In this context we need to identify carefully what constitutes all values. Clearly, all the base types and all values in the language constructed over those base types must be included. This naturally includes objects and arrays, and in our design it includes classes and code. JAM treats instances of the class `Class` specially, having special loading rules and different storage arrangements. We argue that instances of this class should be treated identically, for persistence purposes, with those of other classes so that if an object is in the stable store we can guarantee that the code to manipulate it is also available (see 2.3).

Treating classes consistently with other data results in a PJava store being self-contained and self-describing. In particular, there is sufficient information for garbage collection and schema edit. This can be compared with many OODBMS that have incomplete information about their contents; particularly those making C++ persistent which are usually forced to treat the class definitions (schema) and the code separately and specially.

The principle of **persistence independence** states that all code should have the same form irrespective of the longevity of the data on which it acts. This implies, for example, that transfers are not explicitly programmed. The essence of persistence independence is achieved for all PJava code as the semantics of the language does not vary as a consequence of variations in the longevity of data.

The principle of **persistence identification** states that there should be a straightforward and consistent mechanism for determining the longevity of values. In PJava, all values are either object instances or base-type values that are attributes of an object instance (`static` variables behave as attributes of instances of the class `Class`). Identifying persistence therefore reduces to identifying how long each object instance should persist as attributes of those instances will then persist as well. To maintain the pointer and space management safety of Java this has to be based on reachability [Atkinson and Morrison 95]. PJava ensures that objects continue to exist for as long as they are transitively reachable from a distinguished persistent root for each store, an instance of the class `PJavaStore`[3].

Two additional considerations have influenced the overall design:

Safety may not be compromised by the introduction of persistence. It has been pointed out that the safety of Java is an essential foundation for security [OSF 95] as well as being invaluable for programmer productivity. Type and pointer safety is not compromised in any way in PJava. This is a natural consequence of applying the principle of persistence independence.

Separation of policy and mechanism is already recognised as desirable in operating system design [Rashid *et al*. 89]. In the operating system context, this is generally achieved by allowing system programmers to supply code that defines policy. Not all application programmers can be expected to provide such code and the finer-grained mechanisms in a programming language implementation would be unacceptably costly if they made frequent referrals to application code. Nevertheless, some policies can-

3. In PJava_0 only one store can be used per program run, so that only one instance of `PJavaStore` is visible at one time.

structures to be preserved after one execution and to be re-used in later executions. If these data structures include bindings set up after verifying compliance with Java type rules and other safety mechanisms, then they are available for re-use in the subsequent executions without re-applying the checks. These and other re-uses of the results of earlier executions may provide performance gains.

Goal 3 has been met in its entirety, as no changes are required to the language definition. New classes are provided and the Java Abstract Machine, JAM, will be modified in order to provide orthogonal persistence and transactional facilities. Goal 4 derives from a desire to make the transition to persistent programming easy. Goal 5 supports expert programmers who may want to satisfy end-users requiring particular forms of transaction, for design work, what-if experiments, CSCW, etc.

1.2 Features of Java Relevant to PJava

The Java language is currently receiving much attention [OSF 95]. It therefore provides a golden opportunity to increase the industrial experience of using orthogonal persistence and hence to investigate the feasibility and benefits of orthogonal persistence in a wider range of applications, some of which may be expected to grow larger than the current generation of PASs built commercially. Java is particularly well suited to this because of the care that its designers have taken to make it safe [Gosling *et al*. 96]:

- the type system is well defined and rigorously enforced and
- all pointers and all space are managed *implicitly* by the Java abstract machine and never explicitly by the application programmer.

The existence of native methods (which can be written in C) threatens this safety, but they are already excluded from many contexts and they will be kept away from persistent stores. The class loader already performs dynamic binding, necessary to enable new code to access persistent data and code [Atkinson *et al*. 88] and performs security checks to restrict the code to the required safety levels. This offers safe incremental construction of large PASs.

The type system, which determines the range of values that PJava must support, is a nice combination of simplicity based on regularity and completeness achieved via powerful type constructor rules. The base types are similar to those of C with the exception that booleans are properly differentiated and there are no pointer types. The only constructors are `class, interface` and array (denoted by []). The `class` constructor implements single inheritance, and provides for the definition of a set of labelled attributes of the class, which can have any combination of the following properties:

- `public`, `protected`, `private` or package — determining visibility outside the class definition (when no visibility modifier is provided, the default of package protection is assumed — visible to all classes in the same package);
- `static` or dynamic — determining whether the value is associated with the class or with each instance of the class;
- method or value — determining whether there is code associated with the attribute;
- `final` — which when present prevents the attribute from being over-ridden, or in the case of `static` variables, updated;
- base type or constructed type — determining whether *internally* a pointer is used to represent the value or whether it denotes itself;
- `synchronized` — when present it indicates that other threads should be refused access to the object or class while the attribute given this property is in use; and
- `transient` — which is defined as a reserved field modifier in the language definition but not allocated a function[2].

Interfaces provide a general mechanism for polymorphism. They define the signature of a set of object attributes. Any class that includes the attributes so defined, irrespective of what other attributes it offers, can then be used in any context that requires that interface type.

An extensive and growing library of classes is available with the existing Java system. Those that have to be treated specially during the implementation of orthogonal persistence (because they are not implemented in Java and perform operations relevant to persistence) are:

---

2. We ignore the modifier `transient` but we suspect that the designers of Java intend it to indicate fields of an object that are to be set `null` as the object is stabilised or pickled because they can be reconstructed. We could support such semantics and we provide restoration call-backs that could be used for reconstructing derived data labelled in this way.

Design Issues for Persistent Java[1]: a type-safe, object-oriented, orthogonally persistent system

Atkinson, M.P., Jordan, M.J., Daynès, L. and Spence, S.

29 February 1996

Abstract

The object-oriented programming language Java is receiving much attention and is likely to become a popular commercial programming language because of its regular structure, safety features and modern constructs. It presents a novel opportunity, because of this safety and potential popularity, to make orthogonal persistence defined by reachability widely available. We report on a design for a system that provides such persistence with no changes to the Java language. The design includes ambitious goals for transactional flexibility but also includes simple transactional behaviour sufficient for many applications. We report on several issues that were encountered during the design which as yet have no obvious solution. An outline of the proposed implementation is also given.

1 Introduction

We report on the issues raised when designing the addition and implementation of orthogonal persistence to the Java language. The principles of orthogonal persistence are well known [Atkinson and Morrison 95]. The first orthogonally persistent language, PS–algol [Atkinson *et al.* 83] was conceived in order to add persistence to an existing language with minimal perturbation to its initial semantics and implementation. Our efforts to build a persistent system for Java have a similar motivation. Then the primary concern was to show that orthogonal persistence could be achieved at all and that it would have a beneficial impact on programmer productivity. Now we seek to achieve a much *broader set of facilities* in an *industrially supported language* to demonstrate that orthogonal persistence is beneficial for *large commercial programming projects*.

Typically such projects would currently be built from languages such as C and C++ and relational or object-oriented databases. This paper reports the first steps in an experiment to show that construction and maintenance of such persistent application systems (PAS) can be supported very effectively by an orthogonally persistent language.

We will support a general-purpose, orthogonally persistent system for the Java language [Gosling and McGilton 95, Gosling *et al.* 96]. This system is called PJava. We will evaluate PJava by re-coding in PJava a PAS that is already being developed and supported by an OODB [Jordan *et al.* 95]. This requires a prototype persistent Java-runtime, called $PJava_0$, which will later be superseded by a complete, robust and well-engineered run-time, called $PJava_1$. Measurement of, and experience with, a redeveloped version of the PAS running on $PJava_0$ will provide design data for $PJava_1$.

1.1 Goals for PJava

The design of PJava seeks to provide an orthogonally persistent system that meets the following goals.

1. No loss of safety when utilising persistence and ideally, an increase in consistency.
2. No impact on the performance of programs that do not use persistence; minimum performance penalties and, if possible, performance gains for those that do.
3. Minimal changes to the language.
4. Simple, almost subliminal, use of persistence for many programmers that can be satisfied by a default model of transactions.
5. Sophisticated and precise control of transactional programming for those that require it.

Orthogonal persistence requires that all values, whatever their type or class, should have equal rights to longevity and brevity. Longevity requires adequate mechanisms for recovery after software or media failure and adequate facilities for system evolution. The use of such persistence permits arbitrary data

[1] Java is a trademark of Sun Microsystems Inc.