Persistent Oberon
A Programming Language with Integrated Persistence

Luc Bläser

Computer Systems Institute, ETH Zürich, Switzerland
blaeser@inf.ethz.ch

Abstract. This paper presents the programming language Persistent Oberon, which offers persistence as a naturally inbuilt concept. Program data is automatically kept durable and stored in non-volatile memory, without the programmer having to write explicit code for the interactions with an external database system. In the case of a system interruption or failure, the program can directly continue from its latest consistent state. In contrast to other existent persistent programming languages, this language does not need any explicit or artificial programming interfaces or commands to use persistence. The programming language is completely implemented and offers a high scalability and performance.

1 Introduction

As a consequence of the traditional computer architecture with volatile main memory, programming languages also only support a volatile memory model. Unless the programmer takes extra efforts, the state of a program is lost when the system or application is terminated. As a result of performance advantages on current machines, this design may be reasonable for applications which only perform temporary computations. However, many practical programs work on data that should be persistent and remain present even if the system is interrupted. For this area of application, programming languages currently leave the programmer unsupported and require them to explicitly employ a separate persistence system, such as a database, a serialization framework, or a file system. Even with the help of existing software tools, the programming work and error proneness for managing persistent data within a program are still immense. Complicated and time-consuming work is typically involved in the effective mapping of the program data to the persistent secondary memory (e.g. a disk) and in programming the necessary interactions with the persistence system, for storing and loading the data at the right moments. Especially for object-oriented programs, the intricacy is particularly high, as the dynamic reference-linked object structures need to be efficiently represented in the persistent storage and a memory-safe runtime support with garbage collection has to be provided on all levels of the memory (main memory and disk).

In order to improve the support of persistent data in a program, various approaches have been taken to directly provide persistence as an inbuilt feature of the programming language [3–6, 12, 14, 15, 19, 20, 22, 24, 34]. Although this approach seems to be the most obvious step towards simple and efficient programming with persistent data, none of these programming models have received widespread recognition in practice. A reason for this fact is certainly, that various fundamental problems are still open in this field, such that the programmer may decide against using a persistent language:

- *Language-support*
  To the best of our knowledge, no programming language exists that really features data persistence as a fully integrated concept. Existing persistent programming languages still require artificial programming interfaces when working with persistent data.
- *Concurrency*
  Concurrency is generally not sufficiently supported by persistent programming models, though programmers increasingly use concurrency for modern applications.
- *Interoperability*
  The range of practical applicability can be substantially widened for a persistent programming language by introducing a more general data model, which facilitates consistent and uniform interoperability with data of arbitrary longevity and already existing software.
- *Safety*
  The runtime support for object-oriented persistent programs is often not fully memory safe. Many persistent languages for example, require that garbage collection has to be performed when the system is turned off.
- *Efficiency*
  Persistent programming languages are often less efficient than a conventional solution which uses customized interactions with a database or a persistent storage.

A sustainable solution to these issues seems to be a prerequisite for a potential successful prevalence of the persistent programming vision. For this purpose, we have developed the new programming language Persistent Oberon that aims to address these open problems. The language offers the following key features:

- *Language-integrated persistence*
  The programming language supports data persistence as an elementary feature, without requiring any persistence-related programming interfaces and thinking about a separate external persistence system.
- *General data longevity*
  The programming language is based on a data model, which uniformly covers data of arbitrary longevity, i.e. persistent, volatile and cached data can be used in a consistent way.
- *Effective and safe memory management*
  The runtime system incorporates effective non-disruptive and complete garbage collection with simultaneous caching in volatile main memory. To our knowledge, none of the existing systems is capable of such effective caching for persistent garbage-collection, which works for this general programming model.

While Persistent Oberon has already been very briefly presented in a poster session [11], here we describe the language in more detail, explain its rationale and also report on its implementation. The programming language has been completely implemented on the basis of Active Oberon [16, 28, 32], which is the object-oriented descendant of Oberon [37, 38]. The system supplies the entire infrastructure that is necessary for persistent programming, including a compiler and runtime system, as well as a disk storage and program evolution facility. By means of an experimental evaluation, we also show that the new language offers a high scalability and performance.

The remaining paper is organized as follows: Section 2 motivates the idea of persistent programming and identifies the main shortcomings of existing persistent languages. Section 3 then describes the new programming language Persistent Oberon. In Section 4, the design and implementation of the runtime system is presented together with a performance evaluation. Section 6 reports on related work, before we conclude this paper in Section 7.

2 State of the Art

For object-orientation, concrete criteria of a seamless integration of persistence within a language have already been postulated by the principles of *orthogonal persistence* [8, 7]:

- *Persistence independence*
  All program operations look the same irrespective of the lifetime of the accessed data.
- *Type orthogonality*
  An object type does not predetermine the lifetime of its instances.
- *Persistence identification*
  The concepts of object identification and implicit object lifetimes remain unchanged.

The main idea of orthogonal persistence is to avoid any special handling, which is only required or applicable to persistent data and to fully preserve the philosophy of the underlying programming paradigm. Although many persistent languages [3, 12, 24, 19, 6] (including non object-oriented ones) are claimed to be orthogonally persistent, regrettably none of them fulfils this goal of language-institutionalized persistence:

- Special program functions, interfaces and explicit textual identifiers are required in these languages, to query and fix a root of the persistent object graph, something that is clearly contradicting the principle of persistence independence. Persistent roots have to be handled entirely differently in comparison to the transient ones (such as static variables in Java, module variables in Persistent Modula-3). Figure 1 illustrates how cumbersome it is to set up the initial persistent state in these languages. As an implication of the special persistent roots, a program also has to explicitly determine whether it is started for the first time or is simply resumed after interruption.

– To maintain consistency for the interruptible execution, the abovementioned languages require explicit stabilization (checkpointing) or transactions via dedicated persistence interfaces. These mechanisms also form persistence-specific artifacts that are quite unnatural and complicated to use. More especially, the approach of global checkpoints necessitates the knowledge over the entire program, in order not to prematurely save the temporary modifications of a non-completed logical transaction.

Persistent Modula 3 (PM3)

```
INTERFACE MyModule;
  IMPORT Database;
  DataEntry = OBJECT  (* … *) END;
  VAR myData: DataEntry; p: Database.Public
BEGIN
    TRY
      p := NARROW(Database.Open(“MyModuleData”),
                             Database.Public);
      myData := NARROW(p.getRoot(), DataEntry)
    EXCEPT
     Database.DatabaseNotFound =>
       Database.Create(“MyModuleData”);
       p := NARROW(Database.Open(“MyModuleData”,
                              Database.Public)
       NEW(myData); p.SetRoot(myData)
    END
END MyModule.
```

PJama

```
import org.opj.store;
class DataEntry { /* … */ }
class MyProgram {
  static DataEntry myData;
  public static void main(String args[]) {
    PJStore p = PJStoreImpl.getStore();
    if (p.existsPRoot(“MyProgramData”)) {
      myData =
            (DataEntry) p.getPRoot(“MyProgramData”));
    }
    else {
      myData = new DataEntry;
      p.newPRoot(“MyProgramData, myData);
    }
   }
}
```

Fig. 1. Explicit accesses to the persistent state

3 Persistent Oberon

Persistent Oberon is based on a modular object-oriented programming model, which combines the notion of conventional objects with the concept of modules, as they are known in Oberon [37, 38] and Modula [36]. Modules thereby turn out to be a key concept for introducing persistence in a natural way. Besides being a static compilation and deployment unit, a module represents a singleton instance at runtime that maintains an individual data state. A module is dynamically loaded by the system, as soon as it is used for the first time by the user or another importing module. In the following sections, we explain the main language concepts that are related to persistence support.

3.1 Modules

In our language, a module is designed to live infinitely long in the system. Once loaded and initialized, the module and its contained state stays permanently alive and survives all system restarts. Naturally, references also belong to this persistent state and by default, remain valid at system restart. In other words, modules constitute the persistent roots, implicitly making all transitively reachable objects of the modules persistent. To illustrate the meaning of this, Figure 2 outlines a persistent bank system, together with an exemplary runtime topology of the corresponding object instances. Notably, the code of the module is identical to a conventional transient program and no persistence-specific programming constructs are involved here. In Persistent Oberon, a module can only be unloaded for the reason of changing the program definition. In this case, the runtime system provides an evolution facility that supports the programmer to migrate the persistent data of the former module version to the newer one.

```
MODULE Bank;
  TYPE
    Account = OBJECT
      VAR
        customer: Customer;
        balance: REAL;
    END Patient;
    Customer = OBJECT (* ... *) END Customer;
    AccountList = OBJECT (* ... *) END AccountList;
  VAR accounts: AccountList
END Bank.
```

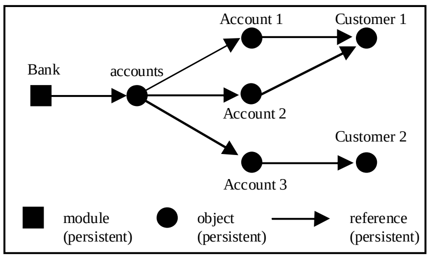

Fig. 2. A persistent program

3.2 Transactions

As the persistent state should always be available in a consistent way when the system is resumed after an interruption or failure, the program execution has to necessarily reflect the states of consistency. For this purpose, the language uses the concept of transactions, which define statement sequences that change the program from one consistent state to another. In Persistent Oberon, a statement sequence (BEGIN-END block) is to be annotated by the TRANSACTION-attribute if it represents a transaction, see Figure 3. All modifications, which are performed by the execution of a transaction (including the code of directly or indirectly called procedures), are either completely applied or not at all. During the uncompleted transactions, these changes are only temporarily valid and are discarded at a system interruption. A transaction may also be prematurely stopped by the programmer by way of the ABORT-statement. In this case, the corresponding transaction statement block is immediately exited and none its modification to the program state become effective.

Naturally, a transaction may also execute statement sequences which are defined as transactions. Such transactions (executed as part of another transaction) are called sub-transactions [25]. A sub-transaction can be aborted without terminating its surrounding transaction. However, an abort of the surrounding transaction always cancels all the sub-transactions and discards all the effects that have been performed by the sub-transactions. Therefore, the effects of a sub-transaction only become durable when the surrounding procedure is also successfully finished. In other words, only the changes of a successfully finished top-level transaction (not enclosed by another transaction) are made persistent. In Persistent Oberon, a single modifying operation automatically forms an implicit transaction if it is not enclosed by an explicit transactional statement sequence.

Figure 3 explains the use of transactions by way of the bank example. The *Transfer* procedure contains a transactional statement block, which runs as a top-level transaction. The transactional statement blocks within the procedures *Withdraw* and *Deposit* are called by the *Transfer* procedure and hence only represent sub-transactions.

```
Account = OBJECT
  VAR balance: REAL;

  PROCEDURE Withdraw(amount: REAL): BOOLEAN;
  BEGIN {TRANSACTION}
    IF balance >= amount THEN
      balance:= balance - amount; RETURN TRUE
    ELSE RETURN FALSE
    END
  END Withdraw;

  PROCEDURE Deposit(amount: REAL);
  BEGIN {TRANSACTION}
    balance:= balance + amount
  END Deposit;
END Account;
```

```
PROCEDURE Transfer(from, to: Account; amount: REAL);
VAR success: BOOLEAN;
BEGIN
   BEGIN {TRANSACTION}
     success := from.Withdraw(amount);
     IF success THEN to.Deposit(amount)
     ELSE from.customer.Inform
     END
   END;
   ReportStatus(success)
END Transfer;
```

Fig. 3. Transactions

Both top-level and sub-transactions feature isolation with respect to serializability[9], of read- and write-accesses on the granularity of objects and modules. This means that concurrent transactions can only see effects of others as if the transactions were executed in a strictly serial order. A sub-transaction is however not isolated from its enclosing transactions, as it has access to the temporary state of the surrounding transactions.

3.3 Interoperability

To allow interoperability with existing non-persistent programs, Persistent Oberon also supports references to objects that do not necessarily have to be persistent but can be of shorter longevity. For example, this could be transient objects, which are only available during an uninterrupted system phase, or cached objects, which can even vanish during the running system when memory space becomes scarce. To enable such shorter object lifetimes, a reference can be declared as *transient* or *weak*, to deviate from the default semantics of a usual persistent reference. The meaning of a transient reference is that the target data does not need to be retained at system interruptions. Analogously, a weak reference permits the disposal of the target reference at any time during program execution. However, transient or weak references do not force shorter lifetime for the referenced objects but merely figure as a suggestion for the runtime system. The value of transient references is safely reset to NIL on system restart and a weak reference is cleared on removal of the referenced object. Significantly, object lifetimes are still determined by transitive reachability, such that in combination with an appropriate runtime system, memory safety can be completely ensured. This may be illustrated by Figure 4, showing an extension of the previous bank example. Everything that is not explicitly declared as transient or weak should be persistent, that is particularly true for all data associated with accounts. The list of account managers, which are currently logged in the system, can be maintained as transient, since they have to logon again after a system interruption. Furthermore, the module also maintains an object cache of the least recently accessed accounts, which are only retained as long as free memory space is not sought by automatic garbage collection. The right-hand side of Figure 4 shows the potential states of the program object graph in different stages, the initial topology, after garbage collection and at system restart. Thereby, the object lifetimes are specified as follows: All objects being transitively reachable from a module via persistent references, are persistent. The other non-persistent objects are transient, if they are reachable via persistent or transient references from a module or the transient state of a running procedure (or transaction). All remaining objects form garbage, which are possibly used as cached data, and are eventually removed from the system.

As a result, the introduced reference semantics enable a general data model, safely interoperable with other preexisting transient programs, such as with low-level operating system modules. Modules written in the persistent programming language may then import classical transient modules and reuse the therein provided logic, with the restriction that the persistent program part only interacts with the data of the imported module by using transient (or weak) references.

3.4 Particular Functionality

We deliberately do not provide the same amount of functionality as a database system offers. The presented model is rather designed for general-purpose programming with a minimum set of fundamental concepts for persistence. Advanced functionality, such as special querying languages, automatic transaction processing, mechanism of data distributions and security policies, can be individually provided by customized program logic.

4 Runtime System

We have implemented an entire execution platform Persistent Oberon, to provide evidence that the proposed persistent programming model can be efficiently realized on conventional computer machines. As a fundament, we have chosen

```
MODULE Bank;
  TYPE
    Account = OBJECT
      VAR
        customer: Customer;
        balance: REAL;
    END Account;

    Customer = OBJECT (* ... *) END Customer;
    AccountList = OBJECT (* ... *) END AccountList;
    BankManager = OBJECT (* ... *) END BankManager;
    ManagerList = OBJECT (* ... *) END ManagerList;
  VAR
    accounts: PatientList;
    managers: ManagerList;

    loggedInManagers: {TRANSIENT} ManagersList;

    leastRecentlyAccessed: {WEAK} AccountList;
END Bank.
```

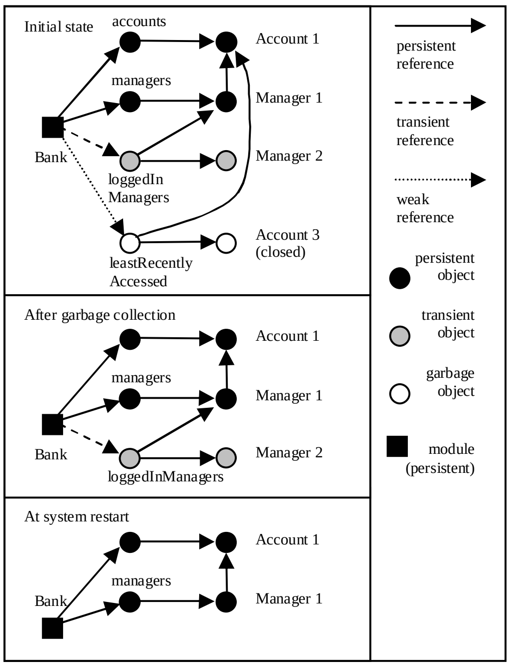

Fig. 4. Using data with shorter lifetimes

the operating system AOS [28], which employs Active Oberon as the native programming language.

4.1 Memory Management

The basic infrastructure of the memory system is the persistent object store (POS), managing the non-volatile memory heap on a disk and enabling fault-tolerant atomic updates or allocations. Furthermore, the system supports main memory caching with a lazy-loading mechanism, where an object is only loaded into memory, when requested for the first time after a system restart. As the normal main memory addresses depend on a system run, synthetic unique object identifiers are used for the reference values, also allowing flexible memory movement of the objects. Consequently, these identifiers need to be mapped to both main memory addresses and locations in the POS. This translation is realized by a residency object table, implemented as a high-scalable and efficient hash table with splay-trees as table entries [17]. Automatic garbage collection is another decisive issue, in continuously ensuring the following memory safety requirements:

– Durability of the latest committed states of each persistent object and module at any time. (Only the top-level transactions change the stable state).

- Exclusion of dangling pointers, i.e. a reference pointing to an object with shorter lifetime than the source object or module.
- Absence of memory leaks, where each non-persistent object is eventually freed from the persistent store and each garbage object is removed from all memory spaces with finite delay.

For this purpose, modules and objects are conceptually classified into two disjoint sets P and T. The set P contains at least all modules and persistent objects and furthermore, forms a transitive closure of reachability via persistent references in the stable states. All transient objects, which are not contained in P, belong to T. The union of P and T represents the transitive closure of reachability via non-weak references in all states (persistent store and main memory). It is essential for correctness that the latest stable states of modules and objects of P always reside in the POS, whereas for all other objects, the POS does not hold a value state. The accuracy of these two sets can be established by two independent automatic garbage collectors: One, called the POS garbage collector, removes objects from P and frees the occupied space in the POS; the other is only responsible for disposing of garbage in the main memory and is hence named the main memory garbage collector.

At system startup, P is initialized with the set of all objects in the POS and T is empty. When a module is activated for the very first time, an empty state is immediately allocated for this module in the POS and the module is added to P. Subsequently, each state modification has to be performed within a transaction. Each transaction has an associated set, called the write-object-set (WOS), recording all objects and modules, which have been modified or allocated during the execution of the transaction. The write-object-set is implemented as a combination of a bucketed list for rapid iteration and a hash-splay [17] for fast searching.

On the commit of a top-level transaction, the system collects all object states that have to be propagated to the persistent store, as specified on the left-hand side of Figure 5. Thereby, it tracks the stable states for the entities unmodified by the current transaction and otherwise, the current states in the transaction's WOS. Transactions cannot commit concurrently, implying that the commit process always maintains a coherent view of the stable states. Conversely, the commit of a subtransaction (in the context of nested transactions [25]) causes each entry of its WOS to be transferred to the super-transaction, if the entry is not yet contained in the super-transaction's WOS. In addition, each transaction maintains a backup of the original states of its modified objects or modules. In the case of a transaction abort, the corresponding backup states are restored in main memory.

The POS garbage collector detects non-persistent objects in the POS and safely reclaims the corresponding free space. Therefore, the collector also has to correctly interact with the simultaneous main memory object cache. To do so, all objects that are detected as non-persistent by the POS garbage collector, are atomically moved to set T under exclusion of intermediate concurrent transactions. As the non-persistent objects may still have transient lifetime, they are

```
PROCEDURE Commit(top-level transaction $A$);
  AcquireLock($TopLevelTransactionCommit$);
  $NewP$ := {}; $MarkStack$ := Empty;
  FOREACH $x \in$ WOS($A$) with $x \in P$ DO
    $RefSet$ := persistent references in current state of $x$
    FOREACH reference in $RefSet$ pointing to $y \neq$ NIL DO
      $MarkStack$.Push($y$)
    END
  END;
  WHILE $MarkStack$ is not empty DO
    $x$ := $MarkStack$.Pop();
    IF $x$ is in main memory and $x \notin P$ and $x \notin NewP$ THEN
      $NewP := NewP \cup \{x\}$;
      IF $x \in$ WOS($A$) THEN
        $RefSet$ := persistent references in current state of $x$
      ELSE
        $RefSet$ := persistent references in stable state of $x$
      END;
      FOREACH reference in $RefSet$ pointing to $y \neq$ NIL DO
        $MarkStack$.Push($y$)
      END
  END;
  Begin atomic POS update;
  FOREACH $x \in$ WOS($A$) with ($x \in P$ or $x \in NewP$) DO
    Store current state of $x$ in POS and set it as the stable state
  END;
  FOREACH $x \in$ ($NewP \setminus$ WOS($A$)) DO
    Promote stable state of $x$ to POS.
  END;
  $P := P \cup NewP$;
  End atomic POS update;
  ReleaseLock($TopLevelTransactionCommit$);
END Commit;
```

```
PROCEDURE RemoveNonPersistentData(object set $S$);
AcquireLock($TopLevelTransactionCommit$);
FOREACH object $x \in S$ DO
  IF $x$ is not present in main memory THEN
    Load $x$ into main memory
  END
END;
$P := P \setminus S$; $T := T \cup S$
Delete $S$ in the POS
ReleaseLock($TopLevelTransactionCommit$);
END RemoveNonPersistentData;
```

Fig. 5. Cache interaction mechanism

loaded to main memory before removal from the persistent storage. The right-hand side of Figure 5 shows the detailed cache interaction by the disposal process of the persistent garbage collector. The described cache interaction mechanism is combinable with any correct (and thus necessarily complete) persistent garbage collector. Because of the characteristics of the non-volatile disk storage, such a collector should specifically support incremental execution, fault-tolerance, minimal I/O-overheads and maximum progress on each collection run [23, 2]. For these requirements, we have chosen the persistent mature object space (PMOS) [26], as a suited underlying garbage collector. It allows incremental and complete collection by using a partitioned object space. A disadvantage of PMOS is however the overhead involved in storing small partitions, because each partition records all incoming references. On the other hand, longer disruptions of the POS result from larger partitions, since the POS is locked during a collection and thus blocks concurrent transaction commits. We have abandoned this trade-off by using larger partitions and only blocking the POS during the evacuation of a small amount of objects per partition. The disposal of non-persistent objects is done within a single blocking period.

The main memory collector has only the task of reclaiming garbage objects in the set T by immediately removing them from the main memory space. As for garbage objects of P, the POS collector first moves them to T, before they can be definitively discarded. This two-step disposal process is necessary because objects in P may still be reachable from a root in the POS, even though they are not so in main memory. Therefore, objects of P are considered as additional root elements for main memory garbage collection. All states must be traced for references and the collector can ignore references pointing to not yet loaded objects. Before garbage is finally deleted, weak references on these objects are reset to NIL, to avoid dangling references.

Transactional isolation is currently realized by serial transaction scheduling. More relaxed execution could be enabled in the runtime system by *strict two-phase locking* or *optimistic concurrency control*[9]. However, these mechanisms possibly abort transactions to prevent deadlocks and effectiveness of unserializable transactional execution, respectively. In such a case, the unexpectedly aborted transaction needs to be restarted. The problem of this approach (and our reason not to use it) is, that long running transactions may suffer from starvation, since they could be continuously aborted by the transaction scheduler.

4.2 Experimental Evaluation

To give an impression of our system's functionality and efficiency, we have measured the performance by the OO7 benchmark [13]. The results are compared to a classical approach, which a user would probably take if they have to develop this persistent application (i.e. the benchmark) within a similar time frame. Such an alternative could be JDO [35], which has been recently advertised as a transparent persistence framework for Java, interacting with a normal database system. Regrettably, the framework does not entirely fulfill this ideal: A programmer has to interact with special persistence API's and needs to provide additional XML-metadata for database mapping[1]. An even greater drawback is that objects are not automatically managed by the runtime system but must be explicitly deleted or made transient. This disagrees with the conventional Java programming model and allows violations of the referential integrity. All tests were run on a PC with Intel Pentium 4, 3GHz, 8KB L1 and 512KB L2 cache, as well as 1GB main memory. The hard disk was a Seagate ST3200822AS with 200GB capacity, 8.5ms average read seek time, 7200 rpm and about 16MB/s transfer-bandwidth. In Persistent Oberon, the data store space resided on a 10GB partition with POS-partition size of 4KB. The garbage collector was continuously active to measure the real efficiency. The JDO system is based on Windows XP with JDK 1.5SE, JDO 1.0, JPOX 1.0.4 vendor implementation, and MySQL 4.0 database[2]. Initially, the measurements should be performed with the small OO7 configuration (about 53,000 objects inclusive collection entries). However, this amount already exceeds the capabilities of the JDO implemen-

[1] For some JDO implementations, one must even account for foreign key constraints in the database: An acyclic data topology first needs to be allocated in the database, before cyclic references may be set in the program.

[2] Many other database systems and JDO vendor implementations could not be used because corresponding license contracts forbid the publication of performance results.

tation, whereas our system runs perfectly with this configuration[3]. Therefore, we had to scale down the configuration to make the comparison possible (8,300 objects inclusive collection entries, see the right-hand side of Figure 6). The results are restricted to the most interesting traversals, each forming a single top-level transaction. The remaining tests gave no further information nor did they show up any contradictions. The left-hand side of Figure 6 summarizes the average execution times including the commit overheads, rounded to two significant figures. T1 is a read-only traversal, whereas T3C updates the data set. Both traversals are distinguished by whether the transaction operates on a cold main memory cache (meaning it is empty) or a warm cache, which already contains all needed objects. CU resembles the costs of solely updating the warm main memory cache. As a result, our system is not only scaling well for higher data loads but also greatly outperforms the JDO system by a factor of about 30 to 80. As for the cache updates, the discrepancy is not that high but this time only accounts for a small part of the total runtime cost. More details about the benchmark implementation, as well as the complete experimental results can be found in [10].

	Persistent Oberon		JDO	
	cold	warm	cold	warm
T1	91 ms	23 ms	3400 ms	1800 ms
T3 C	390 ms	300 ms	13000 ms	11000 ms
CU		81 ms		115 ms

NumAtomicPerComp	10
NumConnPerAtomic	3
DocumentSize bytes	200
ManualSize bytes	1024
NumCompPerModule	30
NumAssmPerAssm	3
NumAssmLevels	3
NumCompPerAssm	3
NumModules	1

Fig. 6. OO7 performance comparison

5 Related Work

Persistent programming has a long tradition and therefore, our language is related to various existing works.

Persistent programming languages. One of the earliest programming languages with support of persistence is PS-algol [3]. It already features persistence by referential reachability, as well as a transactional execution model. Napier88 [24] is a successor of PS-algol. Pointer type annotations were already introduced in object-oriented Persistent Modula-3 [19] for fine-granular specification of persistence reachability [18]. In Persistent Modula-3, a reference can also be declared to refer to an object that ought to be always transient, even if it could be reached over persistent references [27]. This is different to our model,

[3] The JDO system fails with stack overflow errors even for very high stack sizes, or runs out of connection ports (increasing the system parameters only helped to a certain degree).

where a chain of persistent references from a persistent root cannot be broken by an object explicitly forced to always be transient, which implies the risk of dangling persistent references in Persistent Modula-3. [21] shows the integration of transitive persistence into classical non-concurrent Oberon but does not support any transactional features. Optional custom internalization and externalization functions are proposed by [21], to ignore certain references for persistence reachability, which is solved in our work using transient or weak references. PJama [5, 6] is a system providing persistence for Java, also based on reachability from a persistent root. Checkpoints may be performed under PJama, to update modifications of a program in the persistent store but no fine-grained transaction model exists for threads within an application. All of the mentioned languages do not offer a fully language-integrated persistence. They still require artificial programming constructs to deal with persistent roots or to define transactions or checkpoints.

Caching-aware garbage collection. We have designed our own cache mechanism for simultaneous garbage collection in the persistent store, because we could not find another such cache mechanism, which is applicable to our general data model. A series of work on garbage collectors for persistent object systems points out this issue of cache-coordination but does not address it [31, 23, 29]. An interesting collector is reported by [1, 2], which allows concurrent modifications in main memory. The system is however not designed (and does not work) in the presence of non-persistent references, since only reference cuts and newly allocated objects are recorded. In our model, a transient or garbage object can become persistent again, by converting a transient reference to a persistent one. The collector of [30] manages both a transitory and persistent memory heap but does not discard objects from the disk space without system restart. The copying collector of Persistent Modula-3 [20] works in the presence of caching but computes all persistent objects for the entire system with a global stabilization. This is unsuited for our system, since an atomic transaction should only update its own modified objects and should not save the temporary state of other objects used by a different concurrent transaction. To ensure that no flaws are possible in our system, we have formally proved the memory safety of our caching algorithm [10].

6 Conclusion

We have demonstrated that data persistence can be featured as a naturally inbuilt concept of a programming language, enabling the uniform, flexible and safe use of data with arbitrary longevity. Such a language eventually facilitates the development of persistent applications without bothering programmers to write cumbersome and vulnerable code for database interactions. The programming model is intentionally kept to a minimum of fundamental concepts and therefore, does not provide inbuilt mechanisms for special purposes. Instead, one can individually augment this functionality by using customized logic or interoperating with classical non-persistent program modules. The runtime system and its source code are available at [10].

Acknowledgments

I am especially grateful to Prof. Dr. Jürg Gutknecht for many helpful discussions during the entire work, leading to several improvements of conceptual aspects. Many thanks also go to Raphael Güntensperger and Dr. Thomas Frey for their important remarks regarding the design and implementation of the runtime system.

References

1. L. Amsaleg, M. Franklin, and O. Gruber. *Efficient Incremental Garbage Collection for Workstation/Server Database Systems*, Intl. Conf. on Very Large Data Bases (VLDB), Sept. 1995.
2. L. Amsaleg, M. Franklin, and O. Gruber. *Garbage Collection for a Client-Server Persistent Object Store*, ACM Transactions on Computer Systems, 17(3): 153-201, Aug. 1999.
3. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. *PS-algol: A Language for Persistent Programming*, Austrian National Computer Conference, Sept. 1983.
4. M. P. Atkinson, M. J. Daynès, and S. Spence. *Design Issues for Persistent Java. A Type-Safe Object-Oriented Orthogonally Persistent System*, Intl. Workshop on Persistent Object Systems (POS), May 1996.
5. M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. *An Orthogonally Persistent Java*, ACM SIGMOD Record, 25(4):68-75, Dec. 1996.
6. M. P. Atkinson and M. J. Jordan. *A Review of the Rationale and Architecture of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform*, Sun Labs Technical Report TR-2000-90, Sun Microsystems Laboratories, June 2000.
7. M. P. Atkinson and R. Morrison. *Orthogonally Persistent Object Systems*, VLDB Journal, 4(3):319-402, July 1995.
8. M. P. Atkinson. *Programming Languages and Databases*, VLDB Journal, 408-429, 1978.
9. P. A. Bernstein, V. Hadzillacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.
10. L. Bläser. *The Persistent Oberon System*, http://www.jg.inf.ethz.ch/persistence
11. L. Bläser. *A Programming Language with Natural Persistence*, Poster Session, In the Companion of the Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Oct. 2006.
12. C. Boyapati. *JPS: A Distributed Persistent Java System*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1998.
13. M. J. Carey, D. J. DeWitt, J. F. Naughton. *The OO7 Benchmark*, ACM SIGMOD Conference, Washington, D.C., May 1993.

14. J. Carey, D. J. DeWitt, M. J. Franklin, et al. *Shoring up persistent applications*, ACM SIGMOD Record, 23(2):383-394, June 1994.
15. A. Dearle, R. di Bona, J. Farro, et al. *Grasshopper: An Orthogonally Persistent Operating System*, Computing Systems, 7(3): 289-312, 1994.
16. J. Gutknecht. *Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon*, Joint Modular Languages Conference (JMLC), March 1997.
17. Z. He, S. M. Blackburn, L. Kirby, and J. Zigman. *Platypus: Design and Implementation of a Flexible High Performance Object Store*, Intl. Workshop on Persistent Object Systems (POS), Sept. 2000.
18. A. L. Hosking and J. E. B. Moss. *Towards Compile-Time Optimisations for Persistence.* Intl. Workshop on Persistent Object Systems (POS), Sept. 1990.
19. A. L. Hosking and J. Chen. *PM3: An Orthogonally Persistent Systems Programming Language Design, Implementation, Performance*, Intl. Conf. on Very Large Data Bases (VLDB), Sept. 1999.
20. A. L. Hosking and J. Chen. *Mostly-Copying Reachability-Based Orthogonal Persistence*, ACM SIGPLAN Notices, 34(10), 1999.
21. M. Knasmüller. *Adding Persistence to the Oberon System*, Joint Modular Languages Conference (JMLC), March 1997.
22. B. Lewis, B. Mathiske, and N. Gafter. *Architecture of the PEVM: A High-Performance Orthogonally Persistent Java(tm) Virtual Machine*, Intl. Workshop on Persistent Object Systems (POS), Sept. 2000.
23. U. Maheshwari and B. Liskov. *Partitioned Garbage Collection of a Large Object Store*, ACM SIGMOD, 313 - 323, 1997.
24. R. Morrison, R. C. H. Connor, Q. I. Cutts, et al. *The Napier88 Persistent Programming Environment*, School of Mathematical and Computational Sciences, University of St. Andrews, Scotland, 1999.
25. J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge Mass, 1985.
26. J. E. B. Moss, D. S. Munro, and R. L. Hudson. *PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores*, Intl. Workshop on Persistent Object Systems (POS), May 1996.
27. J. E. B. Moss and A. L. Hosking. *Expressing Object Residency Optimiza-tions Using Pointer Type Annotations*, Intl. Workshop on Persistent Object Systems (POS). Sept. 1994.
28. P. J. Muller. *The Active Object System. Design and Multiprocessor Implementation*, PhD thesis 14755, Department of Computer Science, ETH Zurich, 2002.
29. D. S. Munro, A. L. Brown, R. Morrison, and J. E. B. Moss. *Incremental Garbage Collection of a Persistent Store using PMOS*, Intl. Workshop on Persistent Object Systems (POS), Sept. 1998.
30. J. O'Toole, S. Nettle, D. Gifford. *Concurrent Compacting Garbage Collection of a Persistent Heap*, ACM Symposium on Operating System Principles (SOSP), Dec. 1993.
31. A. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*, PhD Thesis, Department of Computing Science, University of Glasgow, May 2000.
32. P. Reali. *Active Oberon Language Report*, Institute of Computer Systems, ETH Zurich, March 2002.
http://www.bluebottle.ethz.ch/languagereport/ActiveReport.pdf
33. J. Seligmann and S. Grarup. *Incremental Mature Garbage Collection Using the Train Algorithm*, European Conference of Object-Oriented Programming (ECOOP), 235 - 252, August 1995.

34. E. Skoglund, C. Ceelen, and J. Liedtke. *Transparent Orthogonal Checkpointing through User-Level Pagers*, Intl. Workshop on Persistent Object Systems (POS), Sept. 2000.
35. Sun Microsystems. *Java Data Objects (JDO)*, http://java.sun.com/products/jdo
36. N. Wirth. *Modula: A Language for Modular Multiprogramming*, Software - Practice and Experience, 7(1): 3-35, 1977.
37. N. Wirth. *The Programming Language Oberon*, Software - Practice and Experience, 18(7): 671-690, July 1988.
38. N. Wirth and J. Gutknecht. *The Oberon System*, Software - Practice and Experience, 19(9): 857-893, Sept. 1989.