Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience[*]

Florian Matthes Rainer Müller Joachim W. Schmidt

Department of Computer Science
Hamburg University
Vogt-Kölln-Straße 30
D-2000 Hamburg 54

`matthes,mueller,schmidt@dbis1.informatik.uni-hamburg.de`

Abstract

A promising approach to improve the quality of next-generation database programming languages is to strictly separate data modeling and data manipulation tasks from data storage issues. This separation greatly simplifies the development of tailor-made database languages and data models based on the generic services of a data model neutral object store protocol that provides efficient, transactional access primitives to persistent, shared and possibly distributed data objects.

As a first step towards this goal, this paper investigates the specific services which *language clients* require from their supporting object stores. It gives insight into the process of selecting a suitable object store for a polymorphic higher-order programming environment (P-Quest) and sketches the mapping of higher-level language objects like functions, environments and polymorphic aggregates, onto more primitive storage structures. The functionality of three candidate object stores (O_2, Mneme and Napier POS) is evaluated based on the requirements imposed by the Quest language.

1 Introduction and Motivation

Database programming languages (DBPLs) like *Pascal/R* [SM80], *Modula/R* [KMP$^+$83], *DBPL* [SEM88]) allow programmers to write applications using the same type system and abstraction mechanisms for both, volatile and persistent data. Typically, the type systems of DBPLs and their persistent storage systems are both oriented towards a specific data model.

Traditionally, the choice of bulk types offered by a DBPL (e.g., sets, relations, lists) heavily influenced the design and implementation of persistent storage systems. Therefore, substantial changes or extensions on the type level disallowed any significant reuse of services for mass data storage and evaluation. With relational storage systems, for example, severe semantic and technical difficulties are encountered when allowing arbitrary nesting of tuple and relation type construction or tuple construction over function types.

As a consequence, most DBPLs (e.g. DBPL [SM91], E [Ric89], O_2 [LR89], Galileo [AGO89]) come with their own tailor-made persistent storage system, a situation which

[*]This work was supported in part by the European Commission, ESPRIT, Basic Research, Contract # 4092 (FIDE)

severely restricts re-usability and interoperability and is therefore unsatisfactory for both, system users and implementors.

Some modern persistent programming languages, however, are based on archtectures which clearly separate modelling from data manipulation and data storage issues. Data modelling tasks are captured by the type system of the PPL, data manipulation is performed by an abstract machine that uses a narrow software interface to accomplish a rather primitive, element-oriented access to objects held in a persistent object store.

The purpose of our work reported here is to develop generic (i.e. data model independent) object stores that are capable of supporting efficient, transactional access to (distributed) multi-user object stores by *multiple* database programming language, possibly with different object-oriented, relational or functional data models. Finally, such generic stores are intended as *servers* for a multiplicity of language *clients* in open distributed environments.

As a first step towards this goal, this paper investigates the specific service requirements of polymorphic higher-order languages (PHOLs) on their supporting object stores. More specifically, we report on the experience gained in a FIDE project subactivity to implement P-Quest, a persistent version of the polymorphic, higher-order language Quest. This language is being used for the initial implementation of the *Tycoon*[1] database programming environment. We choose Quest [Car90], a functional language with imperative features, explicit type quantification and subtyping rules inductivley defined over all type constructors as the starting point of our work, since it comprises already many of the features considered relevant for next-generation database programming languages (object identity, parametric and inclusion polymoprhism and higher-order functions).

This paper is organized as follows: Section 2 presents the characteristics of three object stores available as potential servers for the P-Quest language environment and summarizes their externally visible functionality (object addressing, object layout, persistence model, transaction model etc.). In section 3, this functionality is compared against the specific, rather low-level, requirements of the P-Quest Abstract Machine. This section also justifies the choice of the Napier object store for the P-Quest system implementation. Details of this implementation are subject of section 4 (data representation, portable data import and export, garbage collection, control transfer between object store and P-Quest evaluator). The paper ends with a performance comparison between the persistent and non-persistent Quest system and a summary of desirable features to be supported by object stores serving database language clients.

2 Object Storage Systems

The following subsections give an conceptual overview of three existing Object Storage Systems that are considered as candidates for the integration with the Quest system. Because of the experimental nature of the P-Quest project, we are mostly interested in object storage concepts that are actually implemented, taking the future plans of the storage system suppliers as a hint for our future work.

[1]Tycoon = Typed Communicating Object in Open eNvironments.

2.1 The Mneme Object Store

The overall goal of the Mneme project is the integration of programming languages [HM91, BHM90] and database features to provide better support for cooperative information-intensive tasks. The Mneme Object Store is a fundamental component of this project. As stated in [Mos89c, MS88], the Mneme Object Store is designed to be a generic persistent object management system that should be usable from a varity of tools and programming languages and portable across a wide range of systems. The store functionality is made available to clients as a set of C-library subroutines [Mos89b].

Mneme provides a simple object format. Objects consist of a set of attributes, an identifier section and a byte section. Attributes can be used by clients to model concepts like access rights. All object references must be stored within the identifier section. In addition, a client is allowed to store tagged immediate values in the identifier part. The reason for this design is to allow the easy distinction between identifiers and values.

Every object has a logical identifier (32 Bit) associated with it, used to identify an object in memory. Furthermore, an object identifier can be used to control the placement of newly allocated objects. That is, an object identifier of an existing object can be passed to the object creation function to give a hint where to place the new object. As argued in [Mos89a], 32-bit identifiers seem to be adequate for todays computer architectures because they allow easy object manipulation and support an object space of sufficient capacity for most applications.

To provide fast access to resident objects, Mneme supports two addressing mechanisms; a handle mechanism and a pointer mechanism. Both of them allow a client to abstract from the movement of objects form and to secondary storage. Using the first method, a client first has to create a handle for the to be accessed. Subsequently, the object can be accessed by passing the handle to Mneme object functions. This method eliminates the lookup costs over a series of manipulations of the same object. When access is no longer desired, the handle must be destroyed explicitly since creation and destruction actions are used to trigger buffer management, locking and unlocking actions within the storage system.

The actions to be taken before an object can be addressed via memory addresses (pointers) are conceptually the same (create pointer, destroy pointer). Subsequently, the object can be accessed circumventing the Mneme interface, but some abstraction and saftey is lost. First of all, the main memory layout of objects must be known to the client. Second, no range checks can be done by the Mneme system. Third, the client must be aware of the existence of persistent and session object identifier formats. Fourth, the client must explicitly mark changed objects that have been stored persistently. The advantage of this type of addressing is a possible increase in performance. For integration with programming languages this kind of addressing seems to be feasible because the compiler or the abstract machine can take care of safety aspects.

Up to now the physical organisation of objects has not been discussed. Mneme groups objects together into files which can be seperatly named and located within a network. Files are the basic unit of persistence, they can be created and destroyed dynamically. Persistence within a Mneme file is defined by reachability from a client-defined root object. The object space a client can access within a session is conceptually unbounded. The objects space a client can address at a given point in time is restricted to a maximum of 2^{28} objects, which does not seem to be a real restriction. The maximum number of objects that can be stored within a Mneme file is 2^{20}. An automatic forwarder protocol to transparently handle inter-file references is currently not implemented. The Mneme file concept addresses several issues that are important for the implementation of large, possibly distributed persistent prorgamming environments, such as autonomy of subcollections and, if an automatic forwarder mechanism is added, locality transparency.

The last concept relevant for language clients is the Mneme pool concept. Pools are logical subcollections of objects stored within a Mneme file. A strategy vector must be associated with each pool. The vector contains object management routines, such as cluster strategies, specific to this pool. Different pools might have different strategy vectors. This concept allows a client to optimize the management of groups of objects. For further information on Mneme see [MOS91, Mos90].

2.2 The O_2 Object Manager

O_2 is a database system and an object-oriented system. The target applications of the system are business applications, transactional applications, office automation, and multimedia applications. The O_2 Object Manager handles persistent and temporary complex objects with identity. As stated in [VBD89], the Object Manager (OM) was designed to be as canonical as possible, so that other clients should be able to use the Object Manager as a back-end.

The implementation of the object management system is based on a Client/Server architecture [DFMV90]. That is, there is one Server Object Manager shared among all Client Object Managers. The main difference between the server and the client version is that the first is disk-based and multi-user and the second is memory-based and single-user. The lowest layer of the Server OM is the Wisconsin Storage System (WiSS) [CDKK85] which provides the basic persistence primitives and the basic transaction primitives used by higher levels of the O_2 system.

The O_2 object concept is quite simliar to the O_2 data model [LR89] at the O_2 language level. The O_2 object manager supports tuple, list and set-structured objects and provides a rich set of interface functions to manipulate those objects. At object creation time the client has to specify the structure of the object, i.e. the types of all fields of a tuple object, the element type of a set, etc.

In contrast to Mneme, O_2 object identifiers are physical (they describe the physical location of objects) and are twice as long (64 Bit), to support a large address space. Physical object identifiers are choosen to support fast retrieval of persistent objects and to avoid large object tables that map object identifiers to memory locations. The contents of an object is accessible through the identifiers associated with objects and offsets describing the position of the data to be extracted. O_2 does not support direct access to memory resident data. Like in Mneme, a client can fully abstract from memory management tasks such as movement of objects from and to secondary memory and the implementation of persistent structures.

O_2 objects are stored in O_2 databases, implemented as logical WiSS partitions that must be generated before use. The size of a database is fixed and cannot be extended if necessary. Persistence is defined by reachability from a set of client-defined named (root) objects, which are the entry points to the persistent object graphs. Aside from the above mentioned abstractions, an Object Manager client can fully abstract from storage reclamation tasks. That is, an object must not be explicitly deleted because the Object Manager keeps track of the set of non-reachable objects and recovers the space at suitable time intervalls (garbage collection).

While the transaction concept on the Server Object Manager is based on WiSS transaction with pessimistic concurrency control, the client/sever interaction is implemented using an optimistic approach. An OM client can abstract from the implementation details, which therefore are not discussed here. It is sufficient to say that a client is able to start, to commit and to abort a transaction. A client may switch between different transaction modes, turn concurrency control on/off, run transactions in resident mode (all required objects are prefetched and swizzled) and can activate, respectivly deactivate recovery measures. For a detailed description of the O_2 System see [BDK91].

2.3 The Napier Object Store

The Napier Object Storage System [Bro89] was developed at the University of St. Andrews and designed to support the persistent programming language Napier88 [MBCD89, DCBM89].

As described in [BM91, BMM$^+$92], the Napier Object Store architecture was designed to be an open system architecture to be used for building peristent programming languages. The layered architecture of the system allows clients to adapt the system to their needs and to select an appropriate layer to build their system on. Because of space limitations, only the highest layer, the Stable Heap, is disussed. The functionality of the Stable Heap layer is made available to clients as a set of C-subroutines.

The object concept of the Napier system is quite simliar to the Mneme concept. A Napier object is divided into three parts; an object header, an identifier and a data part. All object references must be stored in the identifier part. In addition, the client can store tagged immediates within the identifier part. The data part can be used to store data of any type. The reason for the segregation of pointer and non-pointers is to support garbage collection. Different from Mneme, Napier allows to specify the number of identifiers stored within an object dynamically. This feature can be used to reduce the amount of work to be done by the garbage collector. Object identifiers (32 Bit) are logical and can change over time. Like in Mneme, Napier supports two object addressing methods. Objects can be accessed by passing an object identifier and an offset to the object functions of the Napier Stable Heap layer or via virtual memory addresses that can be determined by the *keyToAddress* function of the Stable Heap interface. Whereas in the first case a client can abstract from the actual position of an object (main memory, secondary memory) and the main memory layout of an object, the latter abstraction is lost using the second method of object addressing while supporting faster access.

A shadow copy is created before a persistent object is modified. Shadow copies support recovery from system crashes and transactional object manipulation. Any modifications to an existing object store made since the last call to the *stabilise* function of the Stable Heap layer can be undone by canceling the session with the Stable Heap. If the stability function is called within a session, the actual state of the Stable Heap is made persistent. The simple stability mechansim of the Napier system is intended to enable clients to model various kinds of transaction concepts.

All Stable Heap objects are stored in an operating system file, called a Napier object store. This file has to be created before use by the client. Persistence is defined through reachability from the Napier root object defined at store creation-time. The size of an object store and its shadow store (which is part of the object store) must be determined at creation time and cannot be extended dynamically. Further, a clients cannot access different object stores concurrently.

To support a conceptually unbounded object space, the Stable Heap layer provides a garbage collector. Both, the garbage collector and the stabiltiy mechanism are never invoked automatically by the storage system. This enables clients to cache objects outwith the Stable

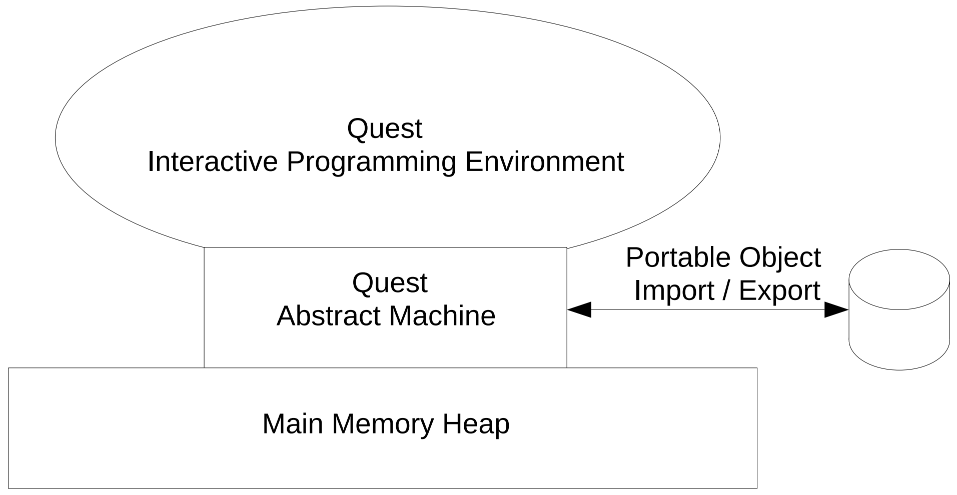

Figure 1: The non-persistent Quest architecture

Heap. Prior to a garbage collection and a stabilization the Stable Heap layer calls a client-supplied *save* function. The purpose of the *save* function is to write all cached objects back to the Stable Heap. On successful execution of these functions, a client-supplied *restore* function is executed. Both functions (save and restore), must be passed to the Stable Heap layer when a session is started.

Another handler function that must be supplied by the client is invoked if the Napier system runs out of shadow space. This situation occurs, if an already persistent object is to be modified and there is not enough shadow space to copy the object prior to its modification. In this case, a client can choose from two possible solutions. He can either cancel the session or automatically invoke the stabilise function.

Like Mneme, Napier is a single user system and does therefore not support concurrent access to a given object store.

3 Selection of a Storage System

The selection of the object storage system for P-Quest is based on the following specific assumptions. First, the P-Quest prototype is intended to be used as a single-user persistent programming environment. As a consequence, the multi-user facilities of the O_2 system that are essential for database applications have not influenced our choice.

The selection is also affected by the requirements of the Quest abstract machine (QM), specifically by those modules that implement the data model of the QM. Fig. 1 shows a simplified version of the QM. Every time the Quest system is activated, the Quest language processor (a pre-compiled Quest program) is copied into the main memory heap. In a next step, the internal (executable) representation of this program is reconstructed and executed. Some of the operations of the export/import modules are also available to Quest programmers. These functions enable programmers to store any kind of Quest data in an encoded format on file and to import these data in subsequent sessions.

The data structures created by the QM during a session are stored within the main memory heap. There are two modules that are responsible for accessing heap structures. The module Value implements the structured data types of the QM and the module Store provides low level access to the heap. An important issue is the efficient mapping of these P-Quest data representations onto the data structures offered by the three candidate object stores.

Most of the data structures created by the QM are instances of generic, polymorphic data types. A *tuple[n]* or a *stack[n]*, for example, are data structures consisting of n fields of type Poylmorph. On creation of such an object only the number of fields but not the types of the fields are known to the QM. That is, it is not known whether a field is used to store an immediate value or a refernence of another data structure.

The mapping of Quest data structures to Napier or Mneme objects does not seem to be too difficult because both systems use 32 bit identifiers and allow clients to store any type of data in the identifier part of objects. Mapping Quest structures on O_2 objects does not seem to be as easy, because object creation is based on object type descriptors that exactly specify the type of each field of an object. Therefore, it is unclear whether polymorphic Quest structures likes stacks, tuples, code-objects, etc. can be efficiently mapped onto O_2 objects, if at all. Further, the QM has no built-in bulk data types like list and sets. Hence, the O_2 bulk types and bulk operations cannot be used. Therefore, only a few routines of the rich set of interface functions of the O_2 Object Manager are required to implement the P-Quest system. The O_2 OM does not seem to be the right candidate for the integration with the QM, mainly because of the differences in the data models and therefore it is excluded from the selection list.

In general, an object management system used to build a persistent programming language should not support a special data model because this might complicate the above mentioned mappings, if at all possible, reduce the performace of the target system and might also restrict future extensions of the client (PL) data model.

Another strong demand of the QM is that the persistent object storage system should support fast access to object data as well as reliable access. As described in sections 2.1 and 2.3, both the Napier and the Mneme system provide fast access (via memory addresses) as well as reliable access methods. This criterion can therefore not be used to select one of the two systems.

A client of a persistent storage system should have the ability to define stable and consistent persistent states of an object store. As described in section 2.3, the Napier system provides a mechanism to stabilise the actual state of a pesistent heap and, combined with the shadow storage concept, supports recovery from system crashes. By canceling a session with the Napier system, it is also possible to undo the changes to an existing object store made since the last call of the stabilise function.

The Mneme system currently does not provide a stability and a recovery mechanism. The only way to save the modifications to a Mneme file is to close the file. After a Mneme file has been closed, there is no way to undo the modifications to the file. Mneme currently supports the creation of pools, but there is only one strategy vector with two management routines available. Further, and in contrast to the Napier system, Mneme has no garbage collector, which is to be reguarded as an important component of a persistent storage system.

The advantages of the Mneme system are more conceptual ones: partitioned object spaces, pools with variable object management strategies, etc. To summarize, the Napier object storage system is chosen for the integration with the Quest system.

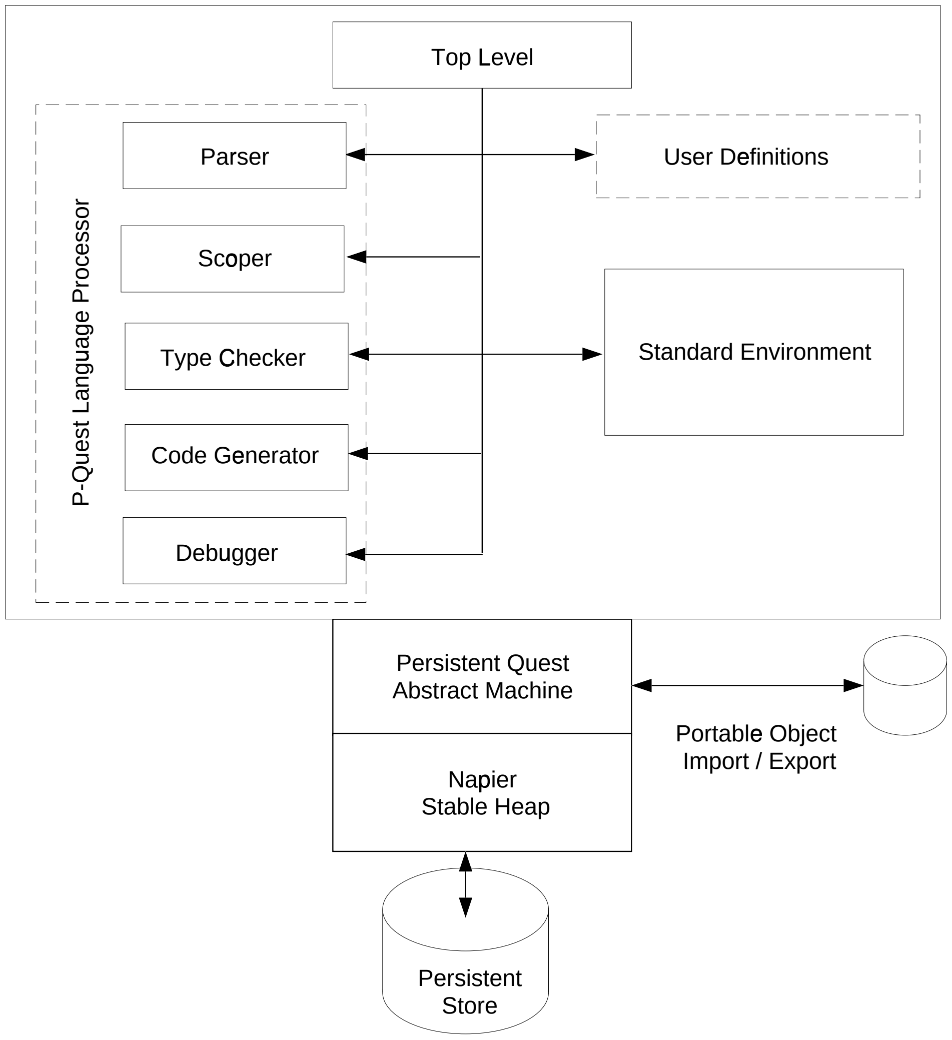

Figure 2: The P-Quest Architecture

4 The P-Quest Programming Environment

Fig. 2 shows the architecture of the P-Quest Environment. Conceptually, there are three layers. The first layer contains the application programming environment, a set of thirteen standard Quest library modules and the Quest language processor: interactive top level, compiler front-end, compiler back-end and a runtime debugger. All of these parts are written in Quest.

The byte code generated by the compiler back-end is interpreted by the portable persistent Quest abstract machine (PQM), the second layer. The Napier Object Store forms the lowest layer.

The persistent address space supported by the Napier Store is the only address space available to the PQM. Hence, all objects generated by the PQM and thus indirectly by higher levels of the system are stored in a Napier Store. There is no volatile, local heap like in Napier88.

4.1 Persistence in P-Quest

Persistence in the P-Quest system is based on reachability from a pre-generated root object. A main task is to map the Quest data structures like functions, programs and type definitions onto the Napier Store model.

A typical P-Quest program usually consists of a set of function definitions and a set of identfiers used to store atomic or structured values. One of these functions serves as the main function, simliar to the main function of a C-program. For every function there is a special data structure, a closure object that contains the global static environment of the function (top level values, references to structured values and closure objects of functions used by that function). In addition, a closure object contains a reference to an object which holds the byte code of the function generated by the Quest compiler-backend. The function code is relocatable and does not contain pointers into the stable heap. Thus, a garbage collector can freely move objects without invalidating programs. The closure object of the *main*-function is the root object of the program. Everything defined within the program is directly or indirectly reachable from the *main*-function. Thus, a P-Quest program can be made persistent by making the closure object of the *main*-function reachable from the P-Quest root object.

Another question is how a P-Quest programmer can control the persistence of his defintions and modifications to the persistent store. At each point in time within a session with the interactive P-Quest environment or within a P-Quest program, the user can call the P-Quest library function *stabilise*, which writes all changes to the stable heap to persistent storage. If a user ends a session without calling this function all new definitions and modifications made since the last call of this function are lost.

An important point is the fact that not only new definitions and modifications but also the actual machine state of the abstract machine is made persistent on a call of the stabilise function. Thus, if a system crash occurs, the P-Quest system can be restarted with the machine state recorded by the last call of the stabilise function. Because the global state and the actual machine state of the persistent abstract machine are important for understanding the persistence implementation, they are described in turn.

The global state of the PQM is given by the tuple:

GlobalState< *StableHeap, PQuestRoot, LevelStack, LevelStackPointer, MachineState*>

The *StableHeap* is provided by the Napier Object Storage System. The *PQuestRoot* is a special object generated by a format program that is part of the P-Quest system. The root object contains some place holders for persistent subroots and some P-Quest system data. The *LevelStack* can store machine states to keep track of recursive invocations of the machine. Typically, the P-Quest system, which is a Quest program runnning at level zero, compiles another Quest program and recursivley starts a machine at level one to execute the compiled program. When the execution at level one stops, the control returns to the machine at level zero. The *LevelStackPointer* points to the last frozen machine state.

A machine state can also be described as a tuple:

MachineState< *Stack, TopLevelPointer, TrapPointer, FramePointer,*
Closure, ProgramPointer, ExceptionObject, ErrorFlag, ShadowFP >

As can be seen from Fig. 3 there are four pointers associated with a machine stack. The *TopLevelPoiner* points to the top of a large bottom frame which contains the values declared at the top level. Above the top level frame are the frames resulting from function activations. The topmost (current) frame is indicated by the *FramePointer*. The component *shadowFP* points to the result slot of the last activation frame. This prevents the garbage collector from reclaiming a result that might be referenced by the function that is currently being executed.

The *TrapPointer* indicates the topmost exception handler. A P-Quest programmer is allowed to write exception handlers which are evaluated if an exception is raised. A trap frame contains the machine state of the exception which should be executed if the related exception occurs and a pointer to the next trap frame below. If an exception is raised which cannot be handled by the topmost exception handler, the exception handling is delegated to the next exception handler. If an adequate handler could not be found, a standard exception handler is activated. Hence, exceptions propagate along the dynamic call chain, but not through machine level boundaries.

The *ExceptionObject* contains an exception name and an exception value and is also used for error handling. The *ErrorFlag*, a boolean value, indicates the termination state of a machine execution, normal or exceptional execution.

As can be seen from Fig. 3 that describes the global state of a P-Quest system including an abstract machine state, there is an additional object, the *LiteralObject*, that is reachable from the closure of a function. The *LiteralObject* typically contains strings defined as values within a function and references to code objects arising from statically nested functions. The last component is the *ProgramPointer* that indicates the next instruction of the program to be executed.

The P-Quest root object (*PQuestRoot*) is generated by a special P-Quest store format program that must be called with the location of a Napier Object Store as parameter before the store can be used by a PQM. The first field of the root object is a place holder for the closure object of the main function of the program to be stored in the object store, typically the top level-function of the interactive P-Quest language processor. The next field is reserved for a reference to a machine state to be recorded when the stabilise function is called. As can be seen from figure 3, the closure object of the machine state and the closure object of the P-Quest root might not be the same, because the stabilise function can be called at any point within a P-Quest program. The next two fields (LS and LP) contain the reference to the *LevelStack* and the relative *LevelStackPointer*. Both of them must be saved because the stabilise function might be called on higher levels of machine activations (e.g. level 1).

The following parts contain references to P-Quest system data, such as format descriptors, built-in error messages, etc. This information is created by the P-Quest store format program. The main purpose of this design is to have unique identifiers for format descriptor objects as well as error message objects.

4.2 Implementation of the P-Quest System

The persistent abstract machine is realized by a module tree shown in Fig. 4. The modules can be divided into three groups: data modules, storage modules and execution modules. The tasks of these three groups are described below.

4.2.1 Storage Modules

The modules POS, Memory and Store belong to the storage group. The Modula-3 interface module POS contains the Napier Stable Heap interface functions as external declarations

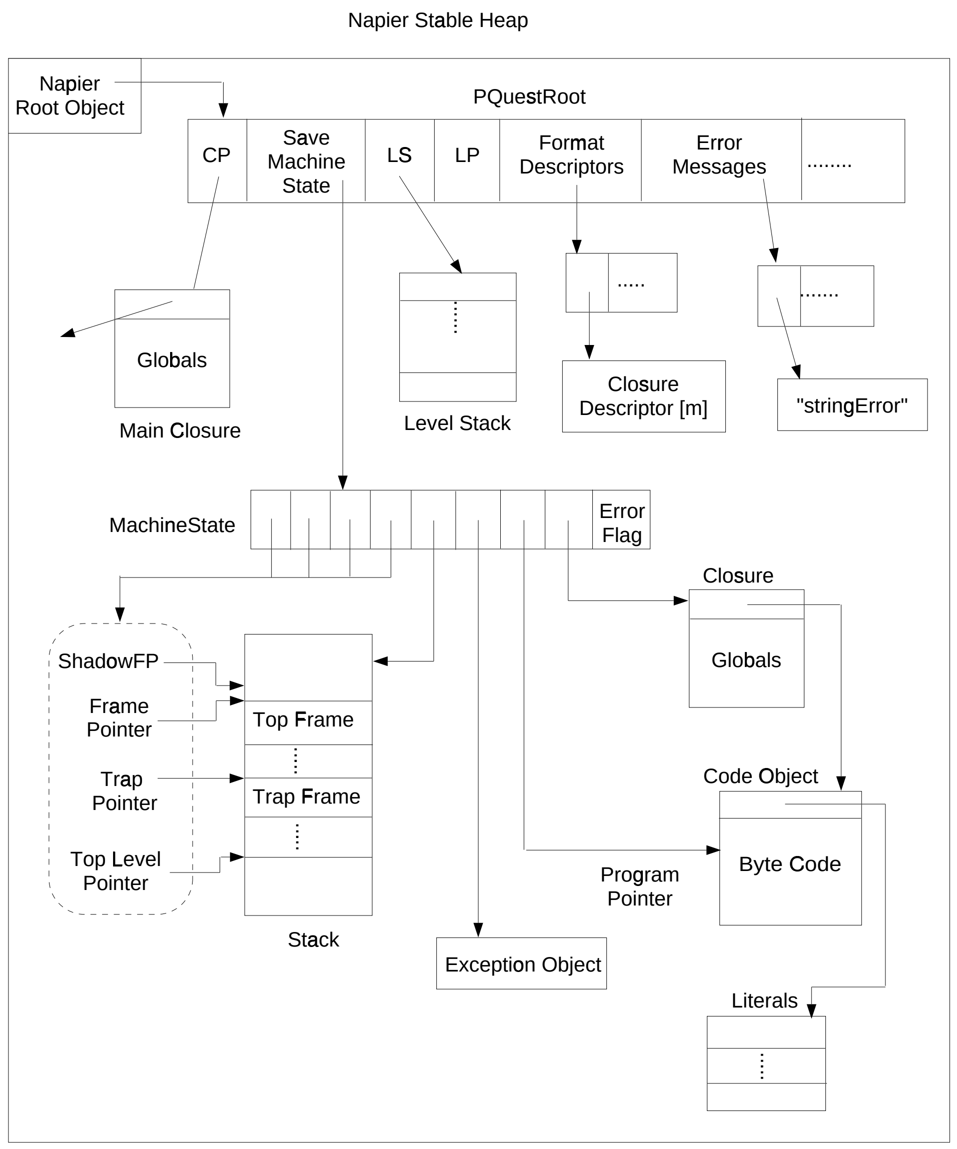

Figure 3: The global state of the P-Quest machine

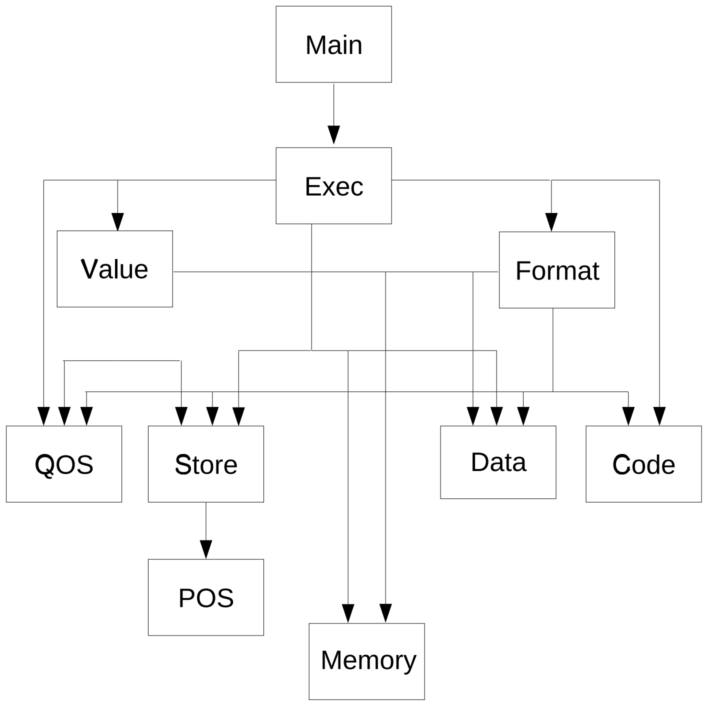

Figure 4: The module tree of the Modula-3 P-Quest machine

(see Appendix A) and makes them available to the Store module. The module Store provides higher levels of the architecture with functions

- to start and end a session with a P-Quest store
- to create new objects
- to read/write data from/to objects
- to determine the virtual memory address of an object
- to activate a garbage collection
- to stabilise the store
- etc.

The function *setup* of this module starts a session with a Napier Stable Heap. There are serveral function parameters which must be passed to the Stable Heap layer. The important ones are *save*, *restore* and *stabilise*.

The *stabilise* function is called by the Stable Heap layer if there is not enough storage to create a shadow copy of an already persistent object that is to be modified. There are two possible solutions for the implementation. First, an implicit stabilization of the Stable Heap can be activated, which will implicitly free the shadow space therefore allowing higher levels to continue execution. Second, the session with the Stable Heap can be canceled. Currently, the second alternative is implemented because only the P-Quest programmer can decide which states of the Stable Heap are consistent and therefore should be stabilized. Another reason is that otherwise a programmer cannot reset the Stable Heap to the last user-defined consistent, stable state (undo a set of changes). Clearly, it is not feasible to have a fixed size shadow space, because this delegates some memory management tasks to the client of the persistent storage system.

The functions *save* and *restore* are called by the Stable Heap layer prior to respectivly after a garbage collection or stabilization of the heap. As mentioned in section 2.3, a Napier client has the ablility to cache objects outside the Stable Heap. Before the heap is stabilized, all cached objects must be written back to the Stable Heap to form a consistent state. This is the task of the *save* function. The *restore* function fulfills the opposite task, to read back the cached objects. This is necessary because object identifiers and virtual memory addresses of objects might be changed by the Stable Heap layer and all components of a machine state have to be marked as reachable.

The modules Main, Exec and Value make use of the ability to cache objects. Each of them contains a local save and restore procedure to maintain the encapsulation of the local buffers. The save procedures are activated bottom up whereas the restore procedures are activated top down. Figure 5 shows the activation chain of the save procedures. The technique used to implement the hierarchical call-chain is termed callback procedures. The dynamic binding required for the save call-chain is achieved as follows:

```
VAR saveproc : PROCEDURE()

PROCEDURE Save() =
BEGIN saveproc();   END Save;

PROCEDURE SetSaveProcedure(proc: PROCEDURE())  =
BEGIN saveproc := proc;  END SetSaveProcedure;
```

The module Store contains two procedures. The procedure *Save* is passed to the Stable Heap layer when a session with the storage system is started. The second one, *SetSaveProcedure*, allows higher levels to dynamically define new save procedures. If a P-Quest user starts a session with the persistent abstract machine, the local variable *saveproc* of the module Store is bound to the save procedure *MainSave* of the module Main by calling the procedure *SetSaveProcedure(MainSave)*. If subsequently the Stable Heap layer calls the *Save* function of the Store module, the *MainSave* function is executed first. The same concept is used for the implementation of the restore procedures.

The stabilise function and the garbage collector are available to a P-Quest programmer through a P-Quest standard library module (see appendix B). Whereas the first is never called by the PQM, the garbage collector is activated if a new object cannot be created, because there is not enough memory available. If after a garbage collection there is still not enough persistent storage, the Napier Object Store is exhausted. In this case the objects stored in the Object Store can only be read and changed. This elucidates the disadvantage of a fixed size object store.

Another disadvantage stems from the fact that the shadow space is also of fixed size. On every call to an updating function such as *SetInt, SetPointer*, the Stable Heap function *CanModify* must be called, because there is no way to determine if the entire object must be copied before it can be altered. Hence, if an object A is to be modified **n** times, the *CanModify*

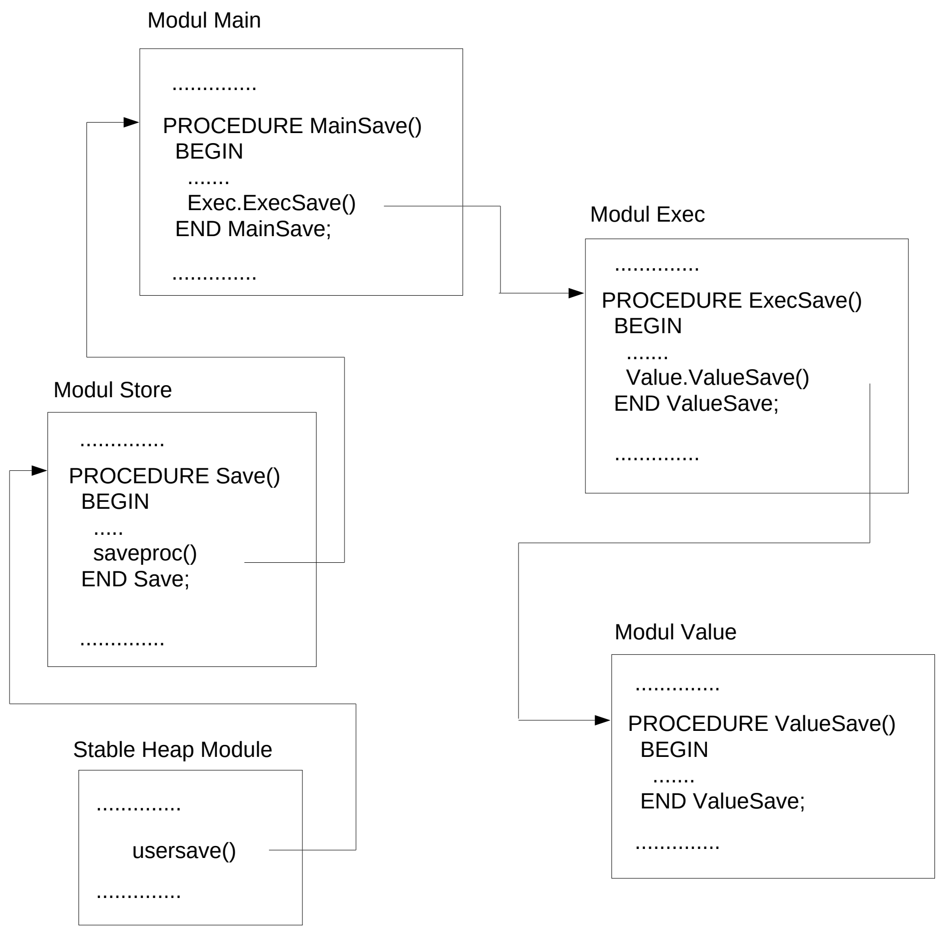

Figure 5: The save procedure activation chain

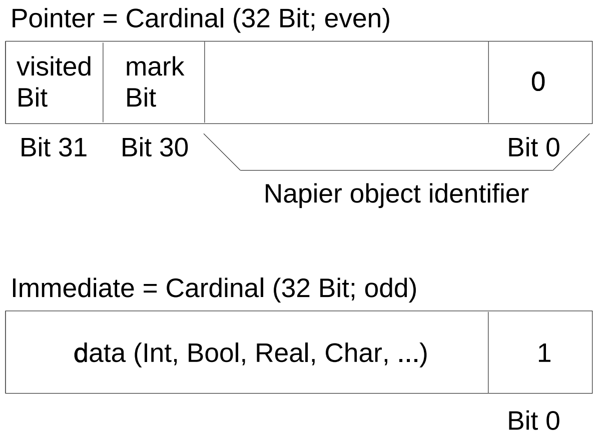
Pointer = Cardinal (32 Bit; even)
visited Bit
mark Bit
0
Bit 31
Bit 30
Bit 0
Napier object identifier
Immediate = Cardinal (32 Bit; odd)
data (Int, Bool, Real, Char, ...)
1
Bit 0

Figure 6: Layout of persistent atomic values

function also must be called **n** times, with obvious negative influence on the performance of the P-Quest system. As already mentioned above, clients of a persistent object storage system should not be involved in solving storage system tasks.

The last module of this group is the Memory module. The Memory module provides functions to modify objects via virtual memory addresses. This kind of object addressing is used to access data form frequently used objects such as machine stacks, byte code objects and to initialize newly allocated objects. The possibility to access objects via virtual memory addresses is the main reason for the good performance of the P-Quest system.

4.2.2 Data Modules

The modules Data, Format, Code, Value and QOS belong to this group of modules. The main task of these modules is to implement the various kinds of atomic and structured data representations used by other modules and to provide operations on this data. Hence, other modules can fully abstract from implemtation details.

The module Data contains the definitons of the atomic values used by the PQM such as Card, Int, Real, etc. and their persistent counterparts Immediate, Polymorph, Pointer used to store values persistently within Napier objects. The main reason for this is that the garbage collector must be able to distinguish pointer (object identifiers) from non-pointer values. The type Polymorph is a non materialized data type which is only used for typchecking. The layout of the persistent representation of the Pointer and Immediate data types are shown in figure 6. As can be seen from the figure, pointers can be distinguished from immediates by the value of Bit zero. The other tag fields of a pointer value are used to control the portable data export (extern).

The module Value implements the structured data types of the PQM and provides operations on them. Examples of structured types are:

- tuple, string, date, ...
- closure, machineState, stack, moduleBadge ...

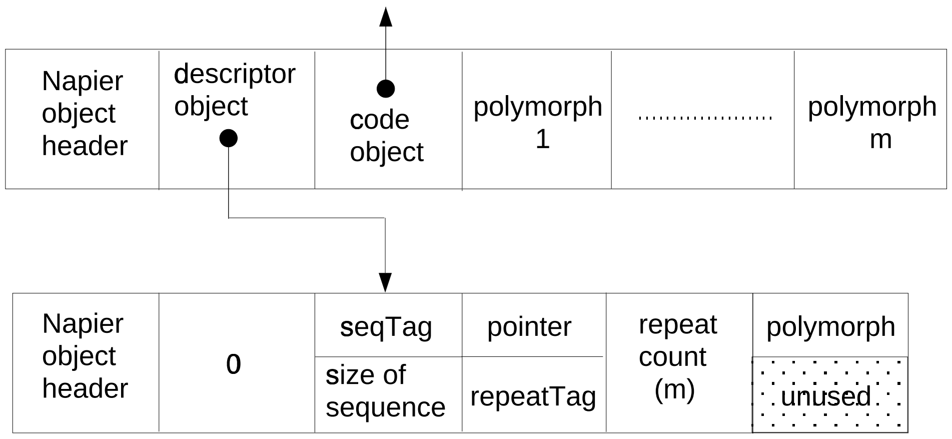

Figure 7: Representation of a closure object

Examples of operations are:

- newTuple, getTuple, setTuple, stringCat
- newClosure, getClosure, ...

The universal layout of a closure object and its associated format descriptor object is given in figure 7. Every object starts with an object header (two words) which is implemented by the Napier system, followed by a format field that contains a pointer to the format descriptor object of the object. The rest of the closure object is used to store the data of the closure object. It contains the pointer to the code object of the closure and data representing the global environment of the closure (a P-Quest function). A given descriptor object is shared among all objects of the same type (closure, tuple, etc.) and size. To distinguish descriptor objects from data objects, an important requirenment of the extern algorithm, the format field of the descriptor object is set to zero. The following part of the descriptor object describes the commmon structure of a closure object of size n. The *seqTag* states that the closure consists of a sequence of length 2; a pointer field and a repetition (*repeat Tag*) of n (repeat count) polymorphic values (pointers or immediates). The desriptor objects thus control the traversal of objects while writing them to a file.

The main components of the Format module are procedures that implement the extern and intern algorithms, which are used to write object graphs (tuples, lists, ..., modules, programs, etc.) in a *machine independent* format to respectivly read them from operating system files. This feature enables a P-Quest programmer to take subcollections of objects (data, functions, modules, ...) from one object store and transfer it to another object store and is also used to bootstrap the P-Quest system. The external format of a P-Quest file is compatible to the external format of non-persistent Quest files. Therefore, a P-Quest and Quest programmer can freely exchange data between these systems.
The Code module contains the definition of the PQM machine code instructions and operand types.

OpCodeClass = {*OpApplyCase, OpClosureCase, OpStoreCase, ..., OpStringCase, ...*}

OpClass = {*OpAddrImmediateCase, OpLiteralCase, ..., OpAddrGlobalCase, ...*}

The operations of the *OpCodeClass* Store are completly rewritten and are available to the P-Quest programmer through the P-Quest module Store. This module provides for example the functions *garbageCollect, stabilise* and *halt.* The first two functions were already discussed. The function *halt* allows a programmer to suspend a program at arbitrary points in time and to return to the operating system shell. The command PQuestRecover restarts the program with its original state (the statement following the *halt* function).

The Store module also exports functions that allow a programmer to access the values of an object in an unsound way. These functions are normally used by the P-Quest language processor to generate the P-Quest byte code, but they can also be used to access any object data, given the object identifier of the object and a legal byte-offset. These features cannot be used in type-safe programs since the module is tagged as "unsound".

The last module belonging to this group is the module QOS. This module provides functions to access operating system files and some other operating system-dependent services.

4.2.3 Execution Modules

The *Main* module contains the main program of the PQM. After executing the Setup procedures of all the other modules that bind the global variables of these modules to persistent objects like descriptor objects and error message objects, the command line parameters passed by the P-Quest user are read. The value of the last parameter determines the action to be taken. If a name of a P-Quest file which contains a pre-compiled P-Quest program is passed, the program is imported into the object store and an initial machine state is generated. If no value is passed, a machine state for the closure stored in the first field of the P-Quest root object is generated. If an exclamation mark is passed the machine state stored in the second field of the P-Quest root object, which represents a suspended program, is retrieved.

The resulting machine state is then passed to the procedure *Exec* of the module Exec, listed below. The Exec module contains the implementation of the P-Quest interpreter.

```
PROCEDURE Exec(machineState: Data.Pointer) =
BEGIN
  MachineStateGet(machineState, stack, TL, TP, FP,
           CP, PC, EX, ErrorFlag, ShadowFP);
           AbsoluteState(stack, TL, TP, FP, CP, PC ShadowFP);
           Execute();
           RelativeState(stack, TL, TP, FP, CP, PC ShadowFP);
           .........
END Exec;
```

In a first step the global variables representing the state of the interpreter are extracted from the machine state passed by the Main module and an absolute machine state is generated. A machine state, as stored within the object store, never contains addresses because they might be invalidated by a call of the save function or following a system fault. The main reason for addressing machine state objects via virtual memory addresses is to boost performance. The Execute call then starts the interpreter.

Some precautions have to be made to ensure that the PQM can continue execution following a stabilization or a garbage collection. First, because the Napier system only allows a fixed amount of persistent memory to be updated between two stabiliztions, any object that must be updated by the save procedures must be modifiable at any point in time. Otherwise, for example if a P-Quest programmer calls the function stabilise and the P-Quest root object cannot be updated, the system will fail. Therefore, all these objects are shadow copied in the setup phase, following a stablisization and a garbage collection.

Second, any value stored within a machine stack must be tagged, and addresses (ProgramPointer, TrapPointer, ...) must be converted to relative offsets before they are stored. Otherwise, the garbage collector may fail or the program cannot be continued because of an invalidated object address.

5 Summary and Conclusions

In this paper three object store systems are compared based on their suitability as servers for a simple single-user persistent higher-order language, P-Quest. Moreover, details of the P-Quest implementation are presented that shed some light on the general problems that arise in the process of implementing a highly polymorphic DBPL on a pre-existing object store.

To show how persistence influences the performance of the P-Quest system, the performance of some Quest functions executed within the Quest and P-Quest system is compared in Fig. 8.

The first two functions contain WHILE statements with an empty statement list. In the first case, the loop control variable is stored on the stack cached in main memory, whereas in the second case it is taken from a tuple in the persistent store and therefore accessed indirectly. As can be seen from the figure, the execution of the first loop in the P-Quest system takes *less* time than in the Quest system. The main reason for this is the fact that the PQM accesses 32 Bit values in a single Modula-3 instruction whereas the QM needs two instructions (first half word, second half word). The crucial difference for the second function stems from the fact that deferencing persistent objects using the Stable Heap access functions is more costly because of the increased number of procedures to be called (Napier Stable Heap, Napier Stable Store) and because of page faults to implement recovery. A static progam analysis which determines the cases when it is useful to address objects via virtual memory addresses could decrease the time used by the P-Quest system drastically.

The comparison of the performance of a simple function (calls involving primitive operations that can be evaluated on the stack) lead to simliar results. The execution of a recursive function (e.g. factorial) involves dereferencing a closure object several times, therefore, the execution within P-Quest takes more time.

The results of creating a list of 1000 Person tuples with five attributes, traversing the list and traversing the list with an update on one tuple attribute leads to the same conclusion: dereferencing persistent objects is expensive.

The compilation of a (P-)Quest program composed of 446 lines of source code (56 functions) takes twice the time in the P-Quest system. This difference in time is compensated if the progam is executed in subsequent sessions with the P-Quest system because the executable representation of the program can be stored permanently whereas within the non-persistent system the compiled program must be imported and decoded in every session it is to be used in.

To summarize, the performance penalty to be paid for the significant additional oper-

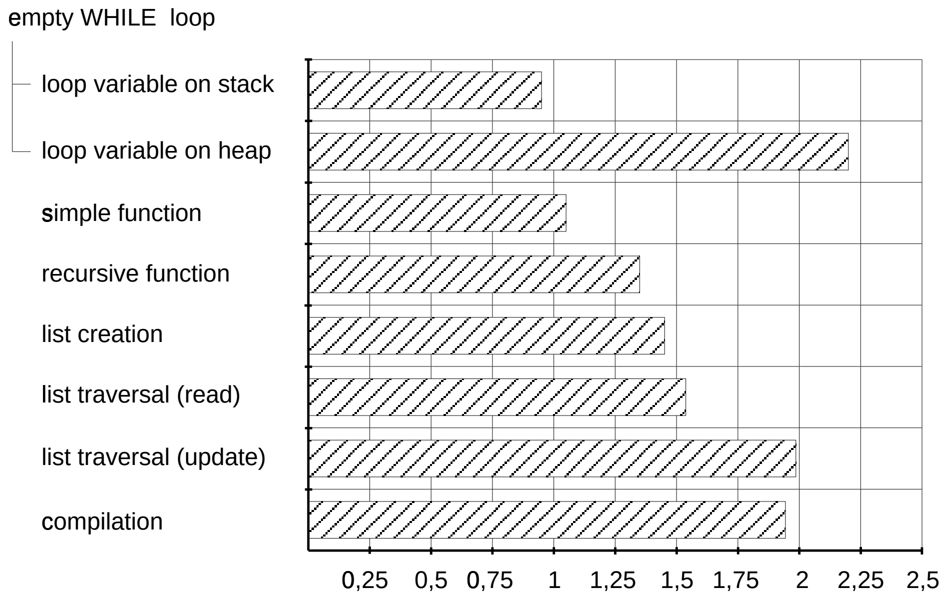

Figure 8: Comparison of the P-Quest / Quest runtime ratio for several programs

ational support provided by an object store (very large address space, recovery, garbage collection) in typical applications is surprisingly low (i.e. below a constant factor of two).

The primary goal of the P-Quest project was to gain experience in implementing simple persistent programming environments and to identify the functionality of object stores required by language clients. The concepts of three persistent object stores, the O_2 Object Manager, the Mneme Store and the Napier Object Store were examined to select the most suitable for the integration with the abstract machine of Quest.

The persistence definition of the three systems is based on reachability and differs only in the number of root objects that have a unique external name. This difference had no influence on the selection, because the Quest abstract machine (QM) does not require multiple root concepts. As was shown in section 4.1, the reachabilty concept offered by all three stores is adequate for the representation of all dependencies between the components of a complete Quest system.

A more important issue concerns the data model supported by an object server. A central requirement of the QM is to be able to uniquely identifiy parts of the persistent memory without restrictions on the kind of data that can be stored within each portion of a persistent object. Both, the Mneme system and the Napier system support this kind of data model whereas the O_2 systems notion of "typed" objects is too biased to a specific data model (object types, sets, lists). This was the main reason to exclude O_2 from the selection list. Generally, typical language clients like Quest do not require set-oriented abstractions as provided by systems like O_2.

Since the performace of a persistent programming environment is a significant criterion for its acceptance, the object storage system should provide fast access methods. Mneme as well as Napier both support direct and reliable access to persistent objects.

Additional features of the Napier system are automatic management of cached objects, garbage collection, stabililty (transactional updates) and recovery from system crashes. Nearly all of these concepts and additional ones (pools, variable object management policies, partitioned object spaces,...) can be found in the design of the Mneme system but are currently not implemented. The lack of important mechanisms like garbage collection and a stablity mechanism were the main reason why the Napier system was selected for the integration with the Quest abstract machine.

In contrast to O_2, the Napier as well as the Mneme object store are single-user systems. Thus it is not possible to share the functionality stored within an object store. Cleary, both systems have to address this issue in future versions.

A The Modula-3 Interface to the Napier Object Store

```
INTERFACE Store;

IMPORT Data;

CONST Headersize = 8;   (* pointees allocated for the object header *)

PROCEDURE SetSaveProcedure(proc: PROCEDURE());

PROCEDURE SetRestoreProced(proc: PROCEDURE());
(* define the save/restore procedures passed to the Napier POS *)

PROCEDURE Setup():BOOLEAN;
(* start a session with a Napier POS and *)

PROCEDURE CloseStore();
(* stop a session with a Napier POS. Modifications made since the
   last call to Stabilise are not saved *)

PROCEDURE Statistics();

PROCEDURE RootObject(): Data.Pointer;
(* Return the object identifier of the root object of a POS *)

PROCEDURE GarbageCollect();
(* Do a full garbage collection of the store*)

PROCEDURE CanModify(key: Data.Pointer): BOOLEAN;
(* Create a shadow copy of a persistent object before altering the
   object. May be called redundantly for the same object. *)

PROCEDURE Stabilise();
(* Write all modifications since the last call of this function or the
   start of the session to the POS *)

PROCEDURE KeyToAddress(key: Data.Pointer): Data.Pointer;
(* Determine the virtual memory address of an object, which is used by
   functions of the Memory module *)

PROCEDURE CreateObject(size: Data.Card; numberOfPolys: Data.Card) :Data.Pointer;
(* The size will be rounded up to multiple of PointeesPerPolymorph *)

PROCEDURE DestroyObject(obj: Data.Pointer);
(* – unused *)

PROCEDURE GetSize(obj: Data.Pointer): Data.Int;
(* Return the size of an object *)

PROCEDURE GetPointee(obj: Data.Pointer; off: Data.Card): Data.Pointee;
(* Return the byte stored at byte offset off. *)

PROCEDURE SetPointee(obj: Data.Pointer; off: Data.Card; pointee: Data.Pointee);
```

```
(* Write pointee to byte offset off within the specified object *)

PROCEDURE GetPolymorph(obj: Data.Pointer; off: Data.Card): Data.Polymorph;
(* Return the Polymorph stored at byte offset off. *)

PROCEDURE SetPolymorph(obj: Data.Pointer; off: Data.Card; polymorph: Data.Polymorph);
(* Write polymorph  to byte offset off within the specified object *)

PROCEDURE GetPointer(obj: Data.Pointer; off: Data.Card): Data.Pointer;
(* Return the object identifier stored at byte offset off. *)

PROCEDURE SetPointer(obj: Data.Pointer; off: Data.Card; pointer: Data.Pointer);
(* Write pointer to byte offset off within the specified object *)

PROCEDURE GetImmediate(obj: Data.Pointer; off: Data.Card): Data.Immediate;
(* Return the immediate stored at byte offset off. *)

PROCEDURE SetImmediate(obj: Data.Pointer; off: Data.Card; immediate: Data.Immediate);
(* Write immediate to byte offset off within the specified object *)

PROCEDURE GetSmallCard(obj: Data.Pointer; off: Data.Card): Data.SmallCard;
(* Return the small cardinal stored at byte offset off. *)

PROCEDURE SetSmallCard(obj: Data.Pointer; off: Data.Card; smallCard: Data.SmallCard);
(* Write smallCard to byte offset off within the specified object *)

PROCEDURE GetSmallInt(obj: Data.Pointer; off: Data.Card): Data.SmallInt;
(* Return the small integer stored at byte offset off. *)

PROCEDURE SetSmallInt(obj: Data.Pointer; off: Data.Card; smallInt: Data.SmallInt);
(* Write smallInt to byte offset off within the specified object *)

PROCEDURE GetInt(obj: Data.Pointer; off: Data.Card): Data.Int;
(* Return the integer stored at byte offset off. *)

PROCEDURE SetInt(obj: Data.Pointer; off: Data.Card; int: Data.Int);
(* Write int to byte offset off within the specified object *)

PROCEDURE GetFloat(obj: Data.Pointer; off: Data.Card): Data.Float;
(* Return the single precision float stored at byte offset off. *)

PROCEDURE SetFloat(obj: Data.Pointer; off: Data.Card; float: Data.Float);
(* Write float to byte offset off within the specified object *)

END Store.
```

B The P-Quest Interface to the Napier Object Store

```
unsound interface Store
export
  (* Note: All offsets given in pointees *)

  headersize: Int
  (* Number of pointees allocated for object header. *)

  byteSize, shortSize, longSize, realSize: Int
  (* Number of pointees allocated for base types. *)

  garbageCollect(): Ok
  (* Do a full garbage collection of the (persistent) store. *)

  stabilise(): Ok
  (* Checkpoint the persistent store. *)

  halt(): Ok
  (* Checkpoint the persistent store and quit the program.
     The program can be restarted at the current instruction. *)

  keyToAddress(A ::TYPE key :A): Int
  (* Return the physical address of an object of type String, Tuple,
     Array, Option, Fun, Interface. The returned address is invalidated
     by (automatic) garbageCollect, stabilise and halt. *)

  getSize(A ::TYPE key :A): Int
    (* Return a multiple of PointeesPerPolymorph. Only valid for
       String, Tuple, Array, Option, Fun and Interface objects. *)

  getByte(A ::TYPE key :A off :Int): Int
  setByte(A ::TYPE key :A off :Int val :Int): Ok

  getShort(A ::TYPE key :A off :Int): Int
  setShort(A ::TYPE key :A off :Int val :Int): Ok

  getLong(A ::TYPE key :A off :Int): Int
  setLong(A ::TYPE key :A off :Int val :Int): Ok

  getReal(A ::TYPE key :A off :Int): Real
  setReal(A ::TYPE key :A off :Int val :Real): Ok

end;
```

References

[AGO89] A. Albano, G. Ghelli, and R. Orsini. Types for Databases: The Galileo Experience. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[BDK91] F. Bancilhorn, C. Delobal, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers (to appear), 1991.

[BHM90] C. Bliss, A.L. Hosking, and J.E.B. Moss. Design of an Object Faulting Persistent Smalltalk. COINS Technical Report 90-45, University of Massachusetts at Amherst, May 1990.

[BM91] A.L Brown and R. Morrison. A Generic Persistent Object Store. PPRR 2-91, Universities of Glasgow and St Andrews, 1991.

[BMM+92] A.L. Brown, G. Manietto, F. Matthes, R. Müller, and D.J. McNally. An Open System Architecture for a Persistent Object Store. In *25th Hawaii International Conference on System Sciences*, volume 2, pages 766–776, January 1992.

[Bro89] A.L Brown. Persistent Object Stores. PPRR 71-89, Universities of Glasgow and St Andrews, March 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital systems research center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[CDKK85] H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug. Design and Implementation of the Wisconsin Storage System. *Software-Practice and Experience*, 15(10), October 1985.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, June 1989.

[DD90] A.J.T. Davie and McNally D.J. Statically Typed Applicative Persistent Language Environments (STAPLE) User's Manual. CSRR CS/90/14, University of St.Andrews, Scotland, 1990.

[DFMV90] D.J. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. Rapport Technique 42-90, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, October 1990.

[HM91] A.L. Hosking and J.E.B. Moss. Towards Compile-Time Optimisations for Persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[KMP+83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R Report, Lilith Version. Technical report, Departement Informatik, ETH Zürich, Switzerland, February 1983.

[LR89] C. Lécluse and P. Richard. The O_2 Database Programming Language. Rapport Technique 26-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, January 1989.

[MBCD89] R. Morrison, A.L. Brown, R. Connor, and A. Dearle. The Napier88 Reference Manual. PPRR 77-89, Universities of Glasgow and St Andrews, 1989.

[Mos89a] J.E.B. Moss. Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, pages 358–374, June 1989.

[Mos89b] J.E.B. Moss. Managing Persistent Data with Mneme: User's Guide to the Client Interface. *Object Oriented Systems Laboratory:* Departement of Computer and Information Science University of Massachusetts Amherst, March 1989.

[Mos89c] J.E.B. Moss. The Mneme Persistent Object Store. COINS Technical Report 89-107, University of Massachusetts at Amherst, October 1989.

[Mos90] J.E.B. Moss. Garbage Collecting Persistent Object Stores. Presented at the OOPSLA/ECOOP '90 Workshop on Garbage Collection, October 1990.

[MOS91] F. Matthes, A. Ohori, and J.W. Schmidt. Typing Schemes for Objects with Locality. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991. (also appeared as TR FIDE/91/12).

[MS88] J.E.B. Moss and S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science. Springer-Verlag, September 1988.

[Ric89] J.E. Richardson. E: A Persistent Systems Implementation Language. Technical Report 868, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SM80] J.W. Schmidt and M. Mall. Pascal/R Report. Bericht 66, Fachbereich Informatik, Universität Hamburg, West Germany, January 1980.

[SM91] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager : an Overview. Rapport Technique 27-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, February 1989.