The Design of a Simulation System for Persistent Object Storage Management

Jonathan Cook, Alexander Wolf, and Benjamin Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-647-93 March 1993

University of Colorado at Boulder

Technical Report CU-CS-647-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

The Design of a Simulation System for Persistent Object Storage Management

Jonathan Cook, Alexander Wolf, and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

{jcook,alw,zorn}@cs.colorado.edu

March 1993

Abstract

Efficient storage management of objects is an important part of any persistent object system. Storage management algorithms, such as those for clustering, caching, and garbage collection, are often complex, since they must simultaneously satisfy a number of constraints, including low CPU overhead, minimum space usage, and low latency. We have designed a simulation system that can be used to evaluate a wide variety of such algorithms. The system is innovative in that it uses trace-driven simulation as well as a loosely-coupled architecture that separates out different aspects of storage management policy, machine hardware, and simulation control into separate modules. Examples of such modules are those for storage allocation, object connectivity, physical characteristics, and cost modeling. The modules communicate using message multicast (i.e., selective broadcast). This approach supports rapid prototyping and rapid evaluation of alternative storage management algorithms.

1 Introduction

Efficient storage management of objects is an important part of any persistent object system. Storage management algorithms, such as those for clustering, caching, and garbage collection, are often complex, since they must simultaneously satisfy a number of constraints, including low CPU overhead, minimum space usage, and low latency. It is important, therefore, to gain an understanding of the performance characteristics of these algorithms before they are actually put to use in real systems.

We have developed a new approach to evaluating algorithms for persistent object storage management. The approach is based on performing trace-driven simulations within a loosely-coupled simulation system architecture.

Trace-driven simulation is a technique that has been successfully applied in the area of main-memory storage management, primarily to study garbage collection and storage allocation algorithms [15, 17, 19]. It works by processing a stream of prerecorded application events that represent activities related to object manipulation (e.g., create, read, and write). The source for those events could be either benchmarks or actual applications.

The loosely-coupled architecture separates out different aspects of storage management policy, machine hardware, and simulation control into separate modules, such as those for storage allocation, object connectivity, physical characteristics, and cost modeling. The modules communicate using message multicast (i.e., selective broadcast) mediated by a message dispatcher. The flexibility of a loosely-coupled architecture allows rapid prototyping and rapid evaluation of alternative storage management algorithms under varying circumstances by supporting the easy replacement of modules. Moreover, the set of modules involved in the simulation is not predefined and, thus, the simulation system can be conveniently extended to account for unanticipated aspects of persistent object storage management simulation.

Although the simulation system has not yet been implemented, we have performed a number of paper simulations to exercise the architectural design. (We have simulated the simulator, if you will.) This has primarily involved studies of the anticipated message traffic among sets of modules implementing various algorithms. These studies have uncovered weaknesses in our original conceptions of the roles of certain modules, but have also shown that the basic idea of using trace-driven simulation and a loosely-coupled architecture is indeed sound.

In this paper, we present the design of our trace-driven, loosely-coupled simulation system. The next section outlines the advantages of a trace-driven simulation approach and describes related work. Section 3 presents the details of the simulation system architecture. Section 4 demonstrates the utility of the system by showing how it can be used to simulate several diverse algorithms for automatic storage reclamation, or *garbage collection*, of a persistent object store. While main-memory garbage collection algorithms have been thoroughly investigated, similar algorithms for persistent object systems are only starting to be proposed and evaluated. Section 5 details the kinds of information we intend to collect with this system, including possible approaches to benchmarking. Section 6 summarizes our conclusions and plans for future work.

2 Background

There are three distinct approaches to evaluating an algorithm:

1. build and measure a prototype implementation;

2. construct and reason about an analytic model of the algorithm; or

3. perform a trace-driven simulation of the algorithm.

All of these approaches can be used to evaluate persistent object storage management algorithms, but by far the most common choice has been prototyping. In this section, we discuss these three approaches to the evaluation of persistent object storage management and consider related work in the area of main-memory heap storage management.

Prototype implementations are often used to evaluate the performance of both main-memory [5, 11, 14] and persistent object storage management [6, 8]. While this approach provides the most realistic performance estimates, it suffers in several ways. The most important drawback of prototype implementations is that they are time-consuming to implement. Many reports of prototype implementations only present the performance of a single algorithm because the cost of fully implementing and comparing multiple algorithms is prohibitive.

Prototypes have other drawbacks as well. Often they are implemented in the context of a complex hardware/software system and have dependences on the architecture of that system. Such dependences often impact prototype performance so that separation of algorithm and system effects on performance becomes difficult. Finally, measuring some aspects of the performance of prototypes, for example the cache miss rate, can be difficult because hardware monitors may be required. Furthermore, prototypes do not allow the full space of design parameters to be easily explored. For example, whereas with a prototype the disk seek time is a fixed physical constant, in a simulation, the system throughput can be evaluated as a function of seek time.

A much less costly approach to performance evaluation is the use of analytic models. Butler investigates the performance of different persistent storage management algorithms using probabilistic models of program reference and update behavior [2]. For a number of dynamic storage allocation algorithms, she shows the expected I/O costs based on complex formulas. This approach has also been taken in evaluating main-memory storage allocation [1, 7, 16]. The greatest weakness of this approach is the degree to which simplifying assumptions about program behavior must be made to keep the model tractable. Zorn and Grunwald show that even very complex analytic models of program allocation behavior often do not sufficiently model actual behavior for the purposes of accurate performance evaluation [21]. As such, analytic models are best used to estimate and compare worst-case performance of algorithms.

Trace-driven simulation fills the gap between prototyping and modeling. When driven by actual traces, the allocation behavior used to perform the evaluation reflects actual programs. Furthermore, because the algorithm is implemented as part of a simulator, many system-dependent details are avoided, greatly reducing the cost of implementation. This approach has been used very successfully in evaluating different cache and virtual memory management systems [9, 13].

More recently, trace-driven simulation has been used to evaluate main-memory dynamic storage management algorithms, including garbage collection. Zorn uses simulation to compare the cost of different generational collection algorithms, arguing that mark-and-sweep algorithms in this context often have cost that is comparable to copying algorithms [18, 19]. Ungar and Jackson use trace-driven simulation to investigate alternative tenuring policies in a Smalltalk system [15]. Zorn [20] and Wilson [17] both use simulation to investigate the cache performance of generational garbage collection algorithms.

The system described in this paper uses trace-driven simulation to investigate the performance of storage management in persistent object systems. Because this approach has proven successful in other areas of performance evaluation, we are confident that applying it in this domain also will yield significant results. Furthermore, the architecture we propose is flexible, facilitating rapid algorithm prototyping and comparison, and innovative, incorporating new ideas in designing simulation system architectures. In the next section, we describe that architecture.

3 Simulation System Architecture

The simulation architecture we propose is message based and loosely coupled. Figure 1 shows the high-level structure of the simulation system. At the top of the figure, messages corresponding to

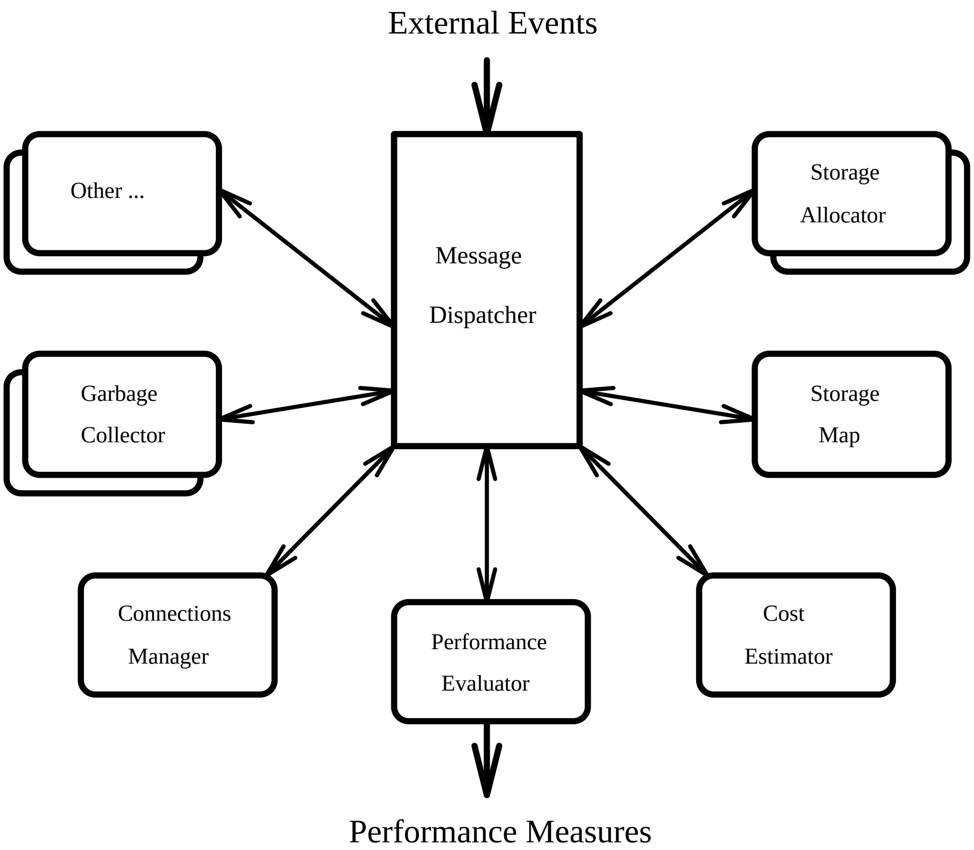


Figure 1: Simulation System Architecture.

external events enter the system and drive the simulation. These events might be generated either by an actual database or by a benchmark program. At the bottom of the figure, results in the form of performance measurements are produced. Central to the architecture is the message dispatcher, which processes all messages generated by the simulation. Modules in the system communicate only through sending messages via the message dispatcher. These messages correspond to:

- External events—persistent object operations performed by the application program.
- Internal events—object and data manipulations caused by the simulation of the persistent object management algorithm.
- Informational messages—information that is passed between modules to conduct the simulation.

The dispatcher supports *message multicast*, that is, when a message is processed by the dispatcher, some number of modules are sent the message. Each module that is interested in a particular kind of message registers a message handler for that kind with the dispatcher.

Message handling in the architecture is completely synchronous. In particular, every message handler informs the message dispatcher explicitly that it is done processing a particular message. Furthermore, if the processing of one message generates additional messages, these sub-messages must be entirely processed before the message that caused them to be generated is considered completely handled. At the highest level, the message dispatcher will read an external event and completely process it before reading the next external event. Note that these semantics do not require that message processing be fully serialized. For example, if an external event message is dispatched to several modules, they may all handle that message concurrently; they must, however, all wait for each other to complete before the next external message can be processed.[1]

3.1 System Modules

In this section we briefly describe the roles of the modules illustrated in Figure 1.

Storage Allocator: This module is responsible for knowing what regions of disk/memory are free (and conversely, allocated) and deciding how to allocate the available space when an object

[1] Of course, as with any system that supports concurrency and waiting, the potential for deadlock exists and, therefore, module implementors must be careful to avoid circular dependencies.

creation request is received. With this organization, the storage allocator needs to know nothing about how free storage is reclaimed, it must simply be informed when some region of space is determined to be free by the garbage collector.

Garbage Collector: This module's sole responsibility is to identify storage that can be safely reclaimed and then inform the storage allocator about that storage. As the figure illustrates, our simulation system allows multiple instances of either the storage allocator or garbage collector algorithms to be simulated. Because these algorithms are encapsulated as loosely-coupled modules, replacing one module with another is very easy.

Storage Mapper: This module implements the mapping from logical object identifiers to the physical location of the object and vice versa. It is responsible for translating a logical object operation, such as a read or a write, into the physical operations necessary to accomplish that operation. The storage mapper also provides information about object contents to other modules that require such information. Finally, it implements the buffer management policy and can identify what objects currently reside in memory buffers.[2]

Connections Manager: This module maintains information about the logical connectivity of all objects in the system. It is responsible for answering questions such as "what objects does this object point to?" and "what objects point to this object?". This module also maintains information about the root objects in the system. Some collection algorithms, such as generational collection, require an augmented root set, corresponding to the objects that contain pointers into a specific part of the address space. In that case, it is the responsibility of the garbage collector to maintain this information and not the connections manager.

Cost Estimator: This module augments physical operation events, such as disk reads and writes, with cost estimates of the event. For example, this module may have a model of the physical disk being used, including its seek time and latency.

Performance Evaluator: The input into this module is a stream of application and simulation event messages augmented with the costs of physical operations associated with these events.

[2] A better approach makes the buffer manager a separate module. We have eliminated this module from the figure to simplify later examples.

Based on this stream, the performance evaluator is responsible for providing information about the performance of a particular algorithm, such as the average number of disk reads and writes per object reclaimed. The main purpose of this module is to accept large quantities of raw data and reduce those data into performance metrics of interest.

Other...: Additional modules may also be necessary and can be easily accommodated in our architecture. For example, some garbage collection algorithms may require input about wall-clock time, such as an algorithm that performs a full collection once a day at 3:00am. A clock module would be responsible for generating timing events that can be used by other modules needing such information.

3.2 System Events and Event Classes

The messages flowing through the simulation system represent application program activity, internal storage management activity, and communication among different parts of the simulation system. These messages can be categorized into several classes. Table 1 identifies the message classes, presents examples of messages in each class, and shows the forms that those messages take.

Messages that correspond to actions performed by an application or by a simulated storage management system are called *events* and may result in costs recorded by the performance evaluator. For example, an object creation by an application is an event, as is a garbage collector reading an edge to carry it to the next object. Some events imply that certain information required for the simulation is needed by a module from some other module. These events are distinguished by a "yes" in the Inquiry column of Table 1.

Informational messages are used to communicate information among modules in response to inquiries. For example, the connections manager responds to an edge read event by sending an edge_data message, which contains the destination object ID of that edge; this response would allow a garbage collector to traverse the reachable objects in a collection scheme such as mark-and-sweep. Although informational messages are generally sent in response to inquiries, there can be some informational messages that are independent in nature, such as a timer message from a clock module.

The translation of events into costs is done through messages implementing a layered I/O abstraction. All events are considered *abstract* I/O messages that are translated by the storage

Message Class	Message Types	Inquiry	Form
Event	create object		create_obj(size,#edges)
	delete object		del_obj(oid)
	set root		set_root(oid)
	unset root		unset_root(oid)
	get root	yes	get_root(prev_root_oid)
	read edge	yes	rd_edge(oid,edge#)
	write edge		wr_edge(oid,edge#,to_oid)
	read data		rd_data(oid,offset,length)
	write data		wr_data(oid,offset,length)
	delete edge		del_edge(oid,edge#)
	read header	yes	rd_hdr(oid,hdr_data)
	write header		rd/wr_hdr(oid,hdr_data)
	object ID at address	yes	addr_obj(address)
Informational	root oid		root_obj(oid)
	object ID of address		obj_addr(address, oid)
	edge destination		edge_data(oid,edge#,to_oid)
	header data		hdr_data(oid,data)
Abstract I/O	all events		
Physical I/O	read address		rd_addr(addr,length)
	write address		wr_addr(addr,length)
Costed I/O	augmented read		aug_rd(addr,length,cost)
	augmented write		aug_wr(addr,length,cost)

Table 1: Message Classes and Example Messages.

map into one or more *physical* I/O messages. These are evaluated by the cost estimator, augmented with associated costs based on the cost model being used, and then translated into *costed* (or augmented) I/O messages. These latter messages are examined by the performance evaluator for analysis.

3.3 Extending the Set of Modules and Events

The set of modules and events supported by the simulation system is easily extended. For instance, a common style of persistent object system interaction is to read a group of connected objects into main memory, operate on the group for a while (possibly removing and adding objects to the group), and finally pass the group with changes back to the persistent object system to obtain persistence.

For the set of external events illustrated and used in this paper, we assume that group operations have been decomposed into operations on individual objects before the trace is generated. With this assumption, however, we lose information about how related objects are being manipulated together and, thus, limit our ability to study the effects of clustering algorithms on performance.

To accommodate such studies, we could consider adding events corresponding to operations on groups of objects (e.g., read_group and write_group). We could also add a module to process group events by decomposing them into a series of low-level events (e.g., object creations and edge writes). This would amount to an internalization of the decomposition we previously assumed was carried out before the trace is generated and essentially gives visibility of grouping to the simulation. To complete this new organization, we could add a new event, place_obj_near, that provides a hint to the storage allocator about where a particular object should be reside.

With the new module and new events in place, we can investigate the performance of clustering algorithms with only one change to the rest of the simulation system (the storage allocation module would need to respond to the place_obj_near event message). This example illustrates both the flexibility and modularity of our architecture.

4 Simulation Examples

In this section, we illustrate the behavior and structure of the simulation system using several message sequences that are part of three diverse and seminal approaches to automatic storage reclamation. The algorithms considered are a non-incremental mark-and-sweep collector [10], an incremental copying collector [1], and briefly a reference counting collector [4]. The performance of variants of these algorithms are also evaluated analytically by Butler [2].

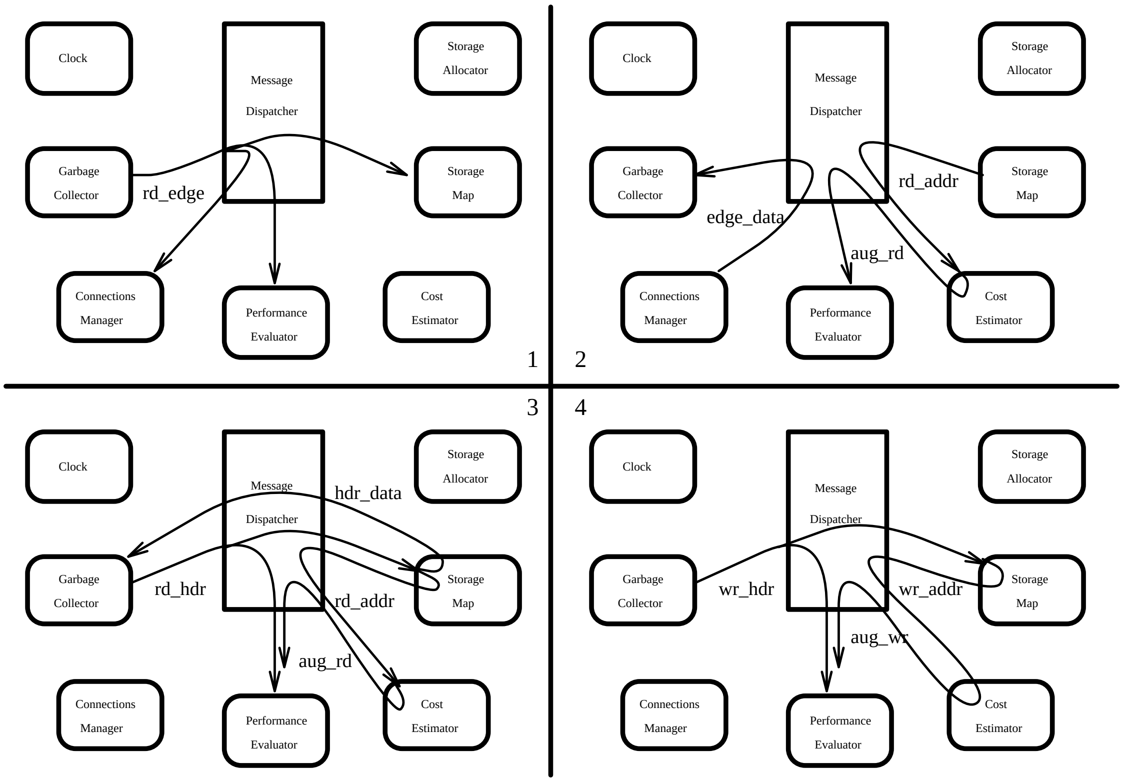
To make the sequences presentable, for each algorithm we only show a message sequence for a single, important operation in the algorithm. Before presenting the examples, we briefly describe each algorithm and the key operation that is shown. The message sequences are shown on a hypothetical instantiation of the simulation architecture, where there is just one garbage collector, one storage allocator, and which includes a clock module.

While these examples illustrate a small part of each of several storage reclamation algorithms, they are not in any way intended to prove that the simulation system we propose is fully defined and tested. We intend these examples simply to suggest the style of computation in the system. The examples both make the description of the architecture more concrete and illustrate its dynamic behavior.

4.1 Mark-and-Sweep Collection

In the classic mark-and-sweep algorithm, objects are traversed transitively starting from a root set, and marked when they are reached. After all objects are marked, the entire address space is swept and unmarked objects, known to be unreachable from the root set, are reclaimed. The non-incremental version of this algorithm performs all marking and sweeping at the same time, resulting in a potentially long pause for the application program. The key events we illustrate for this algorithm are the following. In the mark phase, we show events corresponding to a single edge traversal, mark-test, and mark-set. This operation sequence is the core of the mark phase and is repeated for each reachable edge in the address space. In the sweep phase, we illustrate the message sequence corresponding to the identification and reclamation of a single unmarked (free) object.

Figure 2 shows message sequences of this one step in the mark phase. Frame 1 shows the garbage collector sending a rd_edge message to traverse an edge to the next reachable object. In Frame 2, the connections manager responds with the ID of the object pointed to by the specified



Performance Evaluator's Trace: [(rd_edge, aug_rd),
(rd_hdr, aug_rd),
(wr_hdr, aug_wr)]

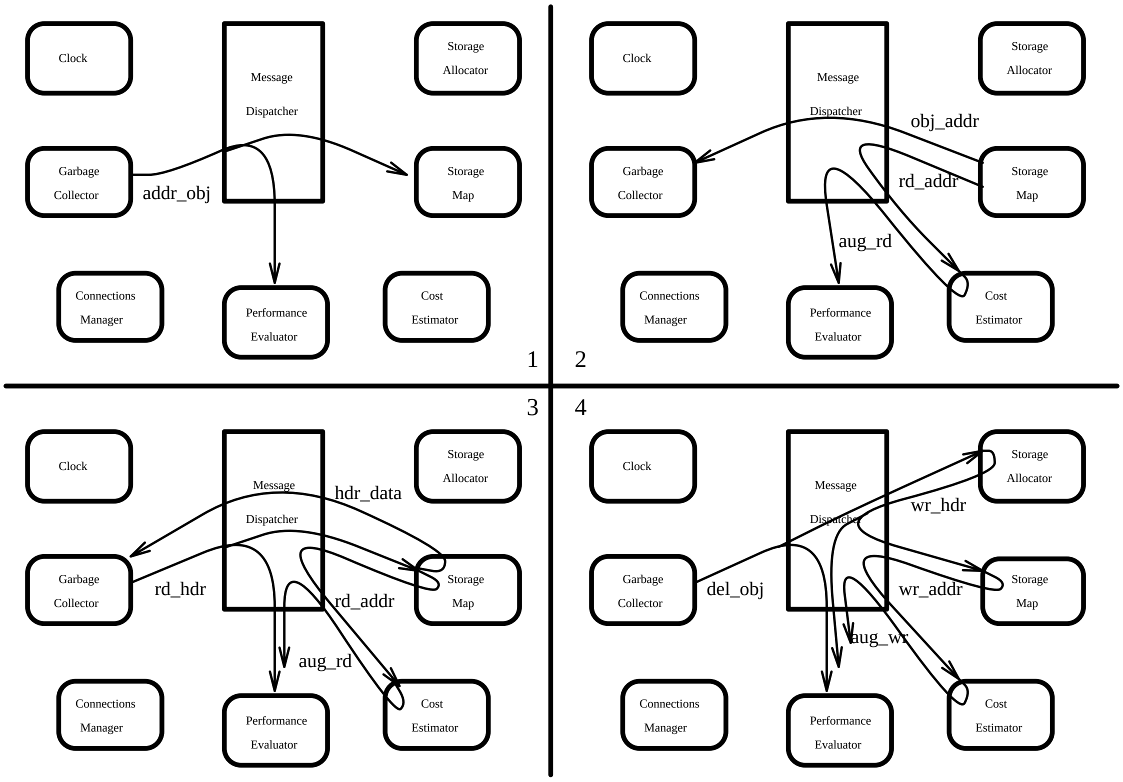
Figure 2: One Step of Marking in Mark-and-Sweep.

Message	From Module	To Module(s)
rd_edge	garbage collector	connections manager, storage manager, performance evaluator
rd_addr	storage manager	cost estimator
aug_rd	cost estimator	performance evaluator
edge_data	connections manager	garbage collector
rd_hdr	garbage collector	storage manager, performance evaluator
rd_addr	storage manager	cost estimator
aug_rd	cost estimator	performance evaluator
hdr_data	storage manager	garbage collector
wr_hdr	garbage collector	storage manager, performance evaluator
wr_addr	storage manager	cost estimator
aug_wr	cost estimator	performance evaluator

Table 2: Mark Message Sequence.

edge, while at the same time the storage map generates a rd_addr (with a corresponding augmented read from the cost estimator), which is the physical read caused by the garbage collector's rd_edge. Frame 3 shows the garbage collector reading the current mark of the object, resulting in another rd_addr. Frame 4 shows the marking event—a wr_hdr from the garbage collector resulting in an eventual augmented write from the cost estimator. The trace of messages recorded by the performance evaluator (indicated at the bottom of the figure) shows the three I/Os in terms of message pairs; the first message in the pair corresponds to the event that caused the I/O, while the second corresponds to the event augmented with cost.

For this first example we also show a complete message-trace listing in Table 2. The causal relationships are shown through the nesting (indentation) of the events—an event at a given level of nesting was caused by the most recent event at one nesting level less. While the contents of a previous informational message can have an influence on a succeeding message, we do not consider that to be the causing event. For instance, in the mark example (Figure 2) the rd_hdr event is at the same nesting level as the preceding edge_data event. Even though the rd_hdr needed the information from the edge_data event to proceed, it was the garbage collector's rd_edge event that set up the examination of the object to which that edge pointed. The same is true for the later



Performance Evaluator's Trace: [(addr_obj, aug_rd),
(rd_hdr, aug_rd),
(del_obj, wr_hdr, aug_wr)]

Figure 3: One Sweep Step in Mark-and-Sweep.

wr_hdr event, which utilized the data from the hdr_data event, but which is at the same scope as the other header processing (e.g., rd_hdr).

While this table shows a strict ordering of the events, this is not enforced by the simulation architecture, and there is some parallelism available in this trace. For example, processing of the first rd_addr and the edge_data, which are both caused by the initial rd_edge, can occur simultaneously.

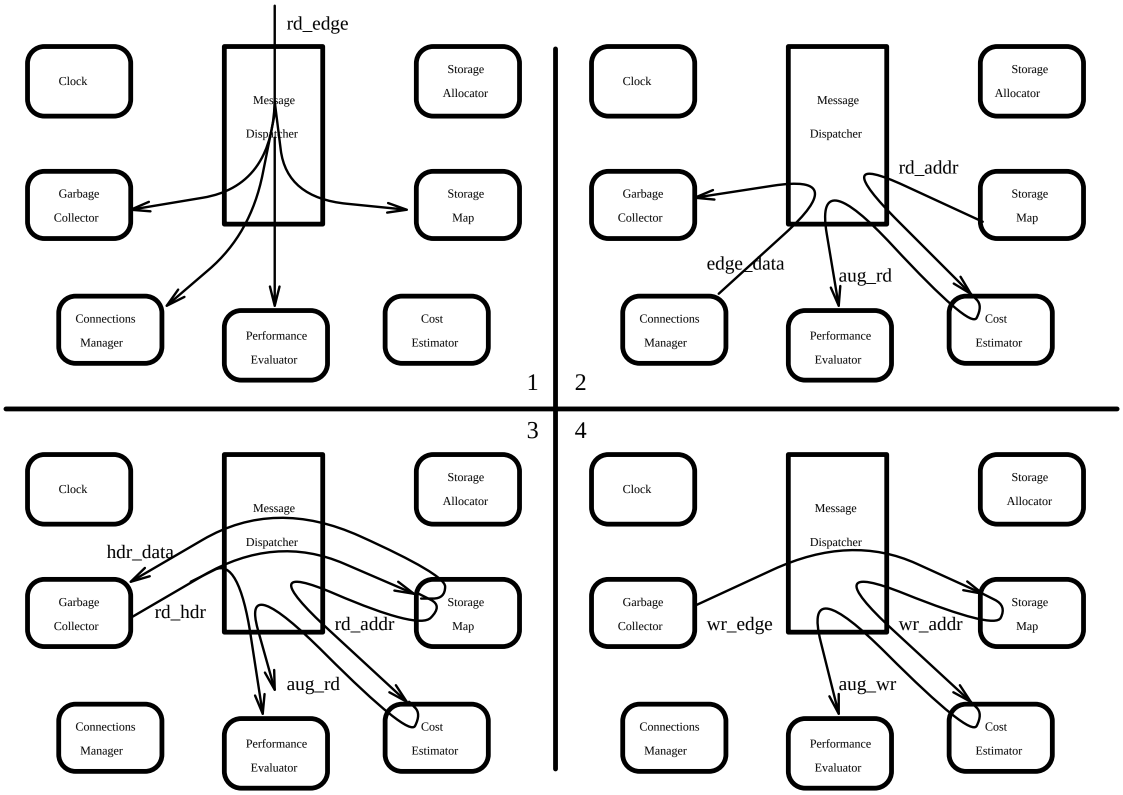
In the sweep phase of the mark-and-sweep collector, the garbage collector must scan through the objects in storage and recover them if they are not marked (thus, they were unreachable from the root(s) during the mark phase). It does this by asking the storage map which object exists at a given address, processing that object, and proceeding sequentially through the storage space. Figure 3 shows the message sequences of one step in the sweep phase, in which the object that is processed is not marked, and thus is reclaimed. Frames 1 and 2 show the address inquiry by the garbage collector, as well as the resulting physical read and object ID reply by the storage map. Frame 3 is the reading of the object header to determine the mark and the object's size (for proceeding to the next object's location).[3] Frame 4 shows the resulting deletion of the object and the write cost associated with that deletion event.

4.2 Incremental Copying Collection

The incremental copying collector is similar to the one described by Baker [1]. This algorithm implements the traditional semi-space copying collection algorithm (with spaces named FromSpace and ToSpace), but objects are copied incrementally. While the details of the algorithm are beyond the scope of this paper, we discuss one aspect of the algorithm here. The Baker collector is made incremental by maintaining an invariant that whenever pointers are read from memory, the object they point to must be located in ToSpace. This invariant is maintained by checking the value of every pointer read and relocating the object and pointer if the object is in FromSpace. If the object has already been copied, a "forwarding address" is left in the FromSpace copy of that object and the pointer being read must simply be rewritten to point to the ToSpace location of the object. The message sequence we illustrate is exactly the scenario described: following a pointer read, the collector determines if the pointer points to FromSpace, identifies a forwarding address if one exists (it does in this example), and relocates the value of the pointer just read.

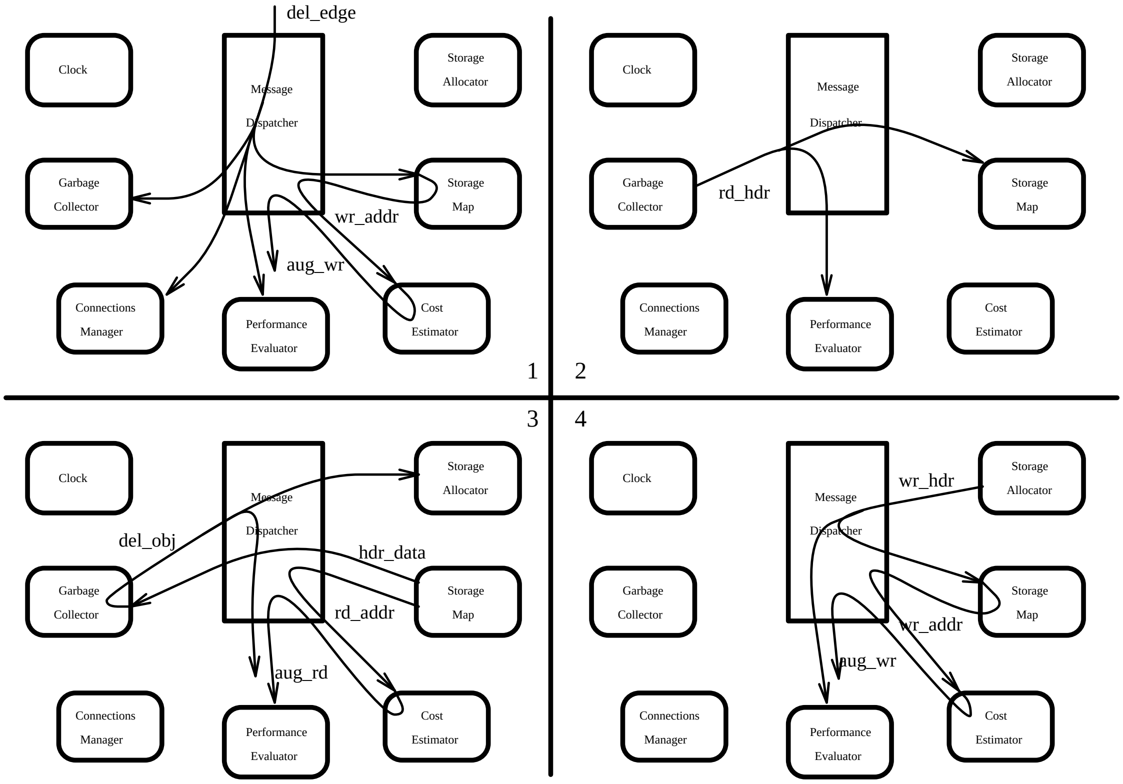
Figure 4 shows message sequences during the processing of a pointer that points to the FromSpace copy of an object. The sequence begins by the application reading the edge from one object (Frame 1), resulting in the costed read and the edge data replies in Frame 2. The garbage collector learns from the edge data that the object is in FromSpace and inquires into the status of that object, as shown in Frame 3. The reply shows that it already has a forwarding address to its copy in ToSpace, and thus the original edge that was read is updated in Frame 4.

[3] If the garbage collector had been implemented to cache this information during the mark phase, then it would have been responsible for generating the I/O message for the cost estimator to evaluate.



Performance Evaluator's Trace: [(rd_edge, aug_rd),
(rd_hdr, aug_rd),
(wr_edge, aug_wr)]

Figure 4: Forwarding a Pointer in Incremental Copying Collection.



Performance Evaluator's Trace: [(del_edge, aug_wr),
(rd_hdr, aug_rd),
(del_obj, wr_hdr, aug_wr)]

Figure 5: Collecting an Unreachable Object.

4.3 Reference Counting

The reference counting collector is another classic algorithm. Each object maintains a count of the number of objects that currently point to it and every time one of those pointers is created or deleted, the reference count is adjusted accordingly. When the reference count reaches zero, the object is reclaimed. This algorithm does not correctly reclaim circular structures, but can be used in conjunction with other collection techniques (such as mark-and-sweep) to reclaim all objects.

Figure 5 shows an application deleting an edge, resulting in the garbage collector recovering an object. To simplify this example, the object is a leaf object, so that cascading reference count changes are avoided. There are three physical I/Os that result from this trace: a write to remove the edge, a read of the object's header to get the current reference count, and a write for the storage allocator to recover the object.

5 Simulation Architecture: Input and Output

In this section, we discuss how the proposed simulation architecture will be used, including how the traces of external events will be generated and what kinds of performance evaluations are possible.

The input to the simulation architecture is an external event trace, representing a sequence of operations on application objects. This trace can either be collected from actual persistent object system executions and fed into the simulator, or the trace can be generated by a synthetic benchmark program that is intended to generate a simulated application load.

Our initial intention is to drive the simulation system with synthetic benchmarks based on the engineering database benchmark described by Cattell [3]. While this benchmark is a starting point, it is not completely appropriate for the measurement and evaluation of persistent object storage management. In particular, Catell's benchmark generates a database and then performs lookups, traversals, and inserts. This benchmark models a system in which the database is monotonically growing in size. To test the performance of storage reclamation algorithms, however, we must investigate a database in which objects are both created and deleted. Thus, we must augment the Cattell benchmark with a series of random reconnections, which will result in some database objects becoming unreachable.

After prototyping our simulation system with synthetic input data, we intend to gather external-event (i.e., application) traces from existing persistent object storage systems. Based on our experience with the synthetic data, we will have a clear understanding of what information the actual trace should contain. A collection of such traces could be used widely for detailed comparisons of implementation techniques.

The output of our simulation system depends on the information that is provided to the performance evaluation module. This module may "listen in" on most system messages, seeing every external event and the corresponding internal events annotated with a predicted cost. With these data, many metrics can be collected. We mention a few of the most important metrics to illustrate what is possible.

The throughput of a system is one significant metric. We are able to measure throughput in simple terms, as a number of I/O operations performed, or in more complex terms, if the cost module provides estimates of the cost of particular disk operations. We are also able to determine what program or algorithm events are responsible for what part of the cost. For example, we can determine what fraction of the total overhead is due to the collection module or the storage allocation module.

Another important metric is the memory/disk utilization and related fragmentation. This information is provided to the performance module by the storage map module via informational messages. When garbage collection is used in an interactive application, system response time is always a significant metric. Our system also allows us to measure the distribution of delays experienced by application users that are caused by storage management operations.

One of the benefits of using trace-driven simulation for performance evaluation is that time-dependent information about system performance is available. For example, while an analytic model, such as Butler's, can provide information about average or worst-case measures (i.e., collection overhead or interactive response), trace-driven simulation allows us to plot the time-varying behavior of these metrics. For example, it is very useful to know that the interactive response of a particular algorithm continually degrades as the program executes.

6 Summary and Future Work

We believe that trace-driven simulation will be an important tool in designing, evaluating, and comparing alternative persistent object storage management algorithms. In this paper, we present the design of an innovative simulation system for evaluating these algorithms. In particular, our system is trace-driven and its architecture is loosely coupled, allowing easy substitution of the simulation modules for rapid development and simulation of alternative algorithms using different performance metrics. Our architecture supports the model of message multicast, where modules interested in being informed about particular messages register their interest with a message dispatcher. Because it is loosely coupled, our architecture allows us to explicitly separate algorithm policy modules. For instance, in our garbage collection example, we separate the storage allocation policy from the storage reclamation policy, resulting in explicit interfaces between these modules and flexibility in the implementation and evaluation of both.

The garbage collection example illustrates how the simulation system will allow us to compare the performance of a variety of related algorithms, in this case traditional storage reclamation techniques, such as mark-and-sweep collection, copying collection, and reference counting. We will be able to perform algorithm evaluations similar to those of Butler, who used analytic models of algorithms and behavior, but to achieve results that go well beyond her worst-case studies.

Because this work is preliminary, there are many unfinished parts. Our immediate goal is to implement a prototype of the simulation architecture. Because implementations of general-purpose message-multicast systems already exist and are publically available (e.g., the Msg component of Field [12]), our hope is to use some of these components in the initial implementation. To drive the prototype, we will implement a synthetic database benchmark and gain experience both with the simulation architecture and the application interface.

Based on this prototype, we see the research proceeding in two main directions. The most important direction is to investigate the performance of different storage management algorithms. Initially, we intend to study algorithms for storage reclamation like those described in this paper. Because this system is quite flexible, we expect that other storage management algorithms, such as object clustering or caching algorithms, will be relatively easy to investigate as well.

Second, we will collect actual traces from existing persistent object systems. We expect the behaviors of actual systems to be significantly different from those of benchmark programs. Collecting and doing evaluations based on such data will lead to storage management algorithms that are effective in actual practice.

In conclusion, we believe that trace-driven simulation as a tool for algorithm evaluation has been and will continue to be very effective in many problem domains. We have outlined the design of a loosely-coupled architecture to implement trace-driven simulations in the domain of persistent object storage management. While we propose to use this architecture in a specific domain, the framework described is quite general and can be useful beyond this application.

References

[1] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[2] Margaret H. Butler. Storage reclamation in object-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 410–423, San Francisco, CA, 1987.

[3] R. G. G. Cattell. *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 6, pages 247–281. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1991.

[4] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[5] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[6] David J. DeWitt, David Mater, Philippe Futtersack, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 107–121, Brisbane, Australia, August 1990.

[7] Tim Hickey and Jacques Cohen. Performance analysis of on-the-fly garbage collection. *Communications of the ACM*, 27(11):1143–1154, November 1984.

[8] Scott E. Hudson and Roger King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[9] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[10] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[11] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[12] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, pages 57–66, July 1990.

[13] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[14] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[15] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[16] Philip L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9):491–500, September 1976.

[17] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generation garbage collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM.

[18] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.

[19] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990.

[20] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, May 1991.

[21] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. Technical Report CU-CS-603-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.