Appears in the Proceedings of the 12th ECOOP, Brussels, July 1998

Framework Design for End-to-End Optimization

Aamod Sane[1], Ashish Singhai[2,⋆], and Roy H. Campbell[2]

[1] Icon Computing Inc.,
11940 Jollyville Road, Suite 300N, Austin, TX 78759 USA.
E-mail: sane@iconcomp.com, Web: http://www.iconcomp.com
[2] Department of Computer Science, University of Illinois,
1304 W. Springfield Avenue, Urbana, IL 61801 USA.
E-mail: {singhai,rhc}@uiuc.edu, Web: http://choices.cs.uiuc.edu

Abstract. Framework optimizations capitalize on object dependencies, while framework flexibility and composability demand object independence. This paper shows how to balance these conflicting needs using new design techniques. These techniques embody the observation that common optimizations can be realized by reifying and tuning object interactions. Their application is illustrated for two complex frameworks: a virtual memory framework and a framework for distributed objects. A catalog of patterns that covers common optimizations is also presented.

1 Introduction

The performance of a framework depends not only on its individual objects, but also on how they interact. For example, consider a file system with a server object S that caches disk data in its buffers. Clients normally get copies of these buffers and they may modify them without affecting the server. But if a particular client C does not modify its copies, then S can allow C to directly access its buffers. This avoids a data copy and saves memory, thereby improving performance. This improvement can not be realized by individually tuning S or C because the optimization depends upon the relationship between S and C. Such optimization, based on relationships between the components of a system rather than the components themselves, will be termed end-to-end optimization.[1]

End-to-end optimization is desirable, but it increases coupling among system components. Thus, S must be changed to support a new interface, while C must be changed to use that interface to request read-only copies. As the objects become more interdependent, the framework becomes harder to understand and reuse.

In this paper, we show how to enable end-to-end optimization in frameworks without making the objects overly interdependent. Our approach is to encapsulate object interactions and then customize the interactions for performance without changing the objects themselves.

⋆ Ashish Singhai is supported in part by the grant NSF-CDA-94-01124.
[1] The idea of whole system or end-to-end analyses originates in [24].

Appears in the Proceedings of the 12th ECOOP, Brussels, July 1998

This scheme has several benefits. First, encapsulation localizes the changes needed for optimization. So it is easier to understand the optimizations, and to estimate how adding new objects or tasks would affect efficiency. Second, we can develop domain- or application-specific optimizations and readily realize them by customizing the interactions. Third, going beyond framework objects, we can reify and optimize interactions between multiple frameworks in an application, as well as between applications (framework users) and the framework.

The idea is effective in practice because most common optimizations can be implemented via modulation of object relationships. We present such implementations as a catalog of design patterns, documenting the applicability and implementation issues for several optimizations. This gives framework builders a method for analyzing and developing whole-framework optimizations in a systematic way.

Overview. In Sect. 2 we present examples of framework optimizations selected from a wide variety of frameworks that we have built over the years in the *Choices* operating system [4] and other projects. The frameworks include a file-system framework [40], a virtual memory framework [25], an instrumentation framework [30], and a video conferencing framework [39]. Sect. 3 analyses these examples, motivating a set of objects that make object relationships first-class. We then show that well known optimization techniques from the literature can be expressed using this approach. This leads to our catalog in pattern form.

The catalog (Sect. 4) includes several classes of optimizations: Caching, Optimize Common Tasks, Lazy Evaluation, Eager Evaluation, Using Type Information, Structure Directed Optimization, Reducing Memory Usage, Concurrency Control, and Scheduling. A set of miscellaneous optimizations is also listed. For each class, we discuss how to determine when an optimization is applicable, and how to implement it. Each catalog entry also presents several examples from the literature.

Sections 5 and 6 describe two large frameworks that we have built using this approach: a virtual memory framework [25] and a distributed object framework [37]. Section 5 also includes examples of inter-framework optimizations. After presenting related work is Sect. 7, we conclude in Sect. 8 with a discussion of future research.

2 Examples

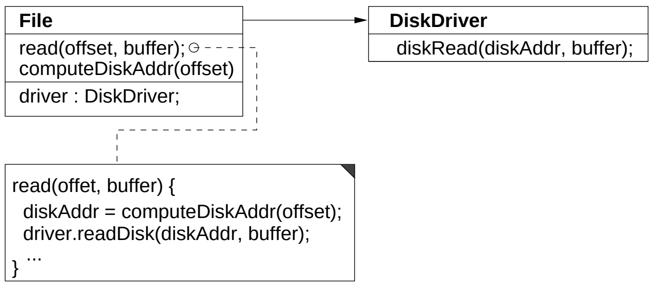
This section presents four examples of end-to-end optimizations in the frameworks we have built. Over the years, we observed that the implementations involved manipulating object interactions in similar ways. Generalizing these observations led to this paper.

2.1 Optimizing File System and Disk-device interactions

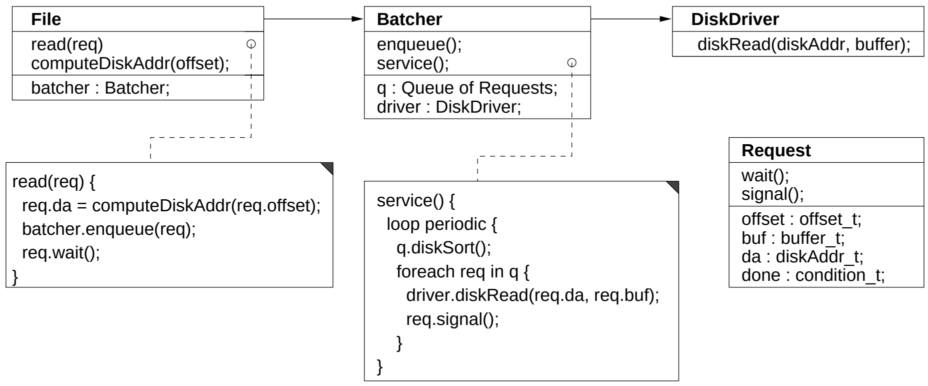
A simple implementation of *File::read(offset, buffer)* is to first compute the disk address *da* for *offset*, and then invoke *DiskDriver::readDisk(da, buffer)* to get the data. This is shown in Fig. 1(a). However, in this implementation, every access incurs disk seek latency. Therefore, fast implementations *batch* requests that refer to the same track, amortizing the latency over the batch of requests [35]. Batching is implemented by changing how *File* and *DiskDriver* interact.

For instance, first we change the arguments to *read()* and *readDisk()* into *Request*s that can be sorted, queued and compacted. Next, we modify *read()* to enqueue a *Request* and wait for it to complete. Finally, implement a *Batcher* object that groups the requests, invokes *DiskDriver* to read data, and notifies waiting parties of request completion. Same optimization also applies to *write()*. The resulting implementation is shown in Fig. 1(b).[2]

Besides other objects, this optimization introduces an object that represents the exchange of data.



(a) Simple Implementation



(b) Implementation with batching optimization

Fig. 1. File system example (Sect. 2.1) before and after the batching optimization

---

[2] To conserve space, we are presenting figures only for this example.

2.2 Changing concurrency control in virtual memory

Consider a pagefault task in a virtual memory framework [4]. The task locates the faulting virtual page using *AddressSpace*, allocates a physical page using *PhysicalPageStore*, retrieves swapped page contents from the file system, updates *VirtualMap* to make the page accessible, and updates *PhysicalMap*. A pageout task, on the other hand, locates a physical page in *PhysicalMap*, removes it from the *VirtualMap*, adds it to the kernel *VirtualMap*, and calls the file system to swap out the data.

These two tasks may execute concurrently, so they protect the objects they visit using locks. Typical implementations guidelines suggest[3] that locks be acquired and released by object methods, the lock being an instance variable. However, this hides dependencies among locks in method call sequences. Therefore it is difficult and deadlock-prone to add extensions like copy-on-write or distributed shared memory that visit and lock the same objects.

In our virtual memory framework [25] we addressed this problem by coding virtual memory tasks in objects such as *PageOutTask* and *PageFaultTask*. They encapsulate concurrency control and make the order of locking operations explicit. This simplifies adding new tasks like *PageOutToRemoteHost*, customizing tasks for performance like *UniprocessorPageFault*, and sharing and caching data across tasks.

These optimizations introduce task objects that encapsulate control flow and synchronization.

2.3 Instrumentation: Structure Directed Optimization

Instrumentation frameworks provide dynamic inspection of programs. Typically, a user associates *Instrument*s with the objects and classes to be inspected, which report their observations to an external observer, usually a GUI. But fine grained, per-method or per-instance instruments generate large volumes of data and perturb system execution. Also, it is difficult to correlate such fine grained data to answer high level queries such as "is this lock being used by that subframework?" This is because the instruments are only aware of low-level details like instance, classes, and methods; architectural abstractions like patterns and frameworks that depend on inter-object relationships are opaque to the instrumentation.

In our system [31], we use instruments that represent system structure, like *PatternInstrument* and *SubFrameworkInstrument*. These instruments integrate instance and method level data to provide high-level views. This reduces the volume of instrumentation data and minimizes system perturbation.

This optimization makes the high-level structure of interacting objects explicit. We call this *structure directed optimization*.

2.4 Video Conferencing: Control and Data Separation

In a video conferencing system that we are developing [39], *Client*s produce and consume video streams. A straightforward implementation is to let each client

[3] For example, `synchronized` methods in Java implicitly acquire and release locks.

multicast the generated streams, and receive streams generated by the others, so that they can converse with one another. But such an implementation would not scale with the number of clients. On the other hand, most clients in a conference usually listen to a single speaker, so we optimize this case.

It represents each stream as a *VideoStream* that not only abstracts video data, but also has attributes such as *isActive()* and *bitRate()*. A central *ConfControllerTask* represents the control processing for the conference. It detects active streams, inform clients about them, and throttles uninteresting streams. *Client*s need to receive and display only the stream suggested by the controller. The *ConfControllerTask* can be easily adapted to combine multiple streams (e.g., by time-sharing, or tiling) to produce a composite stream. Thus, clients would remain unchanged even if the number of active streams increased.

In this case, we arrive at an efficient implementation by encapsulating the control protocol in an object, and by exploiting the properties of video data.

3 Analyzing the Examples

In this section, we see how the optimizations from the previous section customize object interactions. We classify the interactions into those involving data, control and structure and show how to realize them using standard design patterns. A short example depicts applications of these ideas.

First, we revisit the examples from the previous section. The optimizations in these examples, summarized in Tab. 1, were implemented as follows:

Table 1. Summary of changes to examples in Sect. 2

Example	Basic Objects	Optimization	Interaction Objects
File System	File DiskDriver	Lazy Evaluation	Request Batcher
Virtual Memory	AddressSpace PhysicalMap VirtualMap PhysicalPageStore	Concurrency	PageOutTask PageFaultTask
Instrumentation	MethodInstrument InstanceInstrument ClassInstrument	Structural	PatternInstrument FrameworkInstrument
Video Conference	Client Channel	Type Based	ConfControllerTask

- In the file system example (Sect. 2.1) we reified the read and write requests into a DataObject, called *Request*, and the interaction between the file system and the disk driver into a TaskObject, called *Batcher*. This allowed us to optimize request processing by batching, i.e., rearranging the computation of requests and servicing multiple requests together.

– In the second example (Sect. 2.2), we encapsulated the concurrency control in the framework using TaskObjects like *PageFaultTask*. They localized concurrency control, and made object interactions easier to understand and customize.
– In the third example (Sect. 2.3), we made the high-level system structure explicit to reduce data that we had to store and transfer. The implementation modeled system abstractions as StructureObjects (e.g., *PatternInstrument*, *FrameworkInstrument*). This allowed us to rearrange, coalesce, and reduce computations involving large groups of framework objects.
– In the fourth example (Sect. 2.4), we decoupled data and control paths. Data communication was optimized using multicast, while control path was optimized using a centralized controller task. This resulted in a scalable video-conferencing system.

Observe that all these optimizations were implemented by specializing objects that represented different kinds of object interactions.

DataObject: DataObjects reify object interactions that exchange data.
TaskObject: TaskObjects reify control flow interactions among objects.
StructureObject: StructureObjects reify inter-object relationships such as subsystems.

Note that although we use terms like DataObject, one may use several objects to realize it. The term simply gives a common name to objects that represent data exchanged when framework objects interact.

The above analysis shows us that a framework can be made optimization-ready by introducing DataObjects, TaskObjects, and StructureObjects wherever there is a potential for exploiting object interactions. Different applications can tailor these objects in different ways, while the basic objects in the framework remain unchanged.

Implementing TaskObject, DataObject, StructureObject using patterns. Various design pattern catalogs (such as [10]) suggest patterns that can be used to implement task, data and structure objects. For instance, a TaskObject can be implemented as a COMMAND or a MEDIATOR, and may use patterns like STATE, or techniques like Object-Oriented State Machines [27]. Selection between specialized TaskObjects may be implemented as a STRATEGY. A DataObject can be implemented using PARAMETEROBJECT [26]. A StructureObject can involve a FACADE or a FACTORY augmented with interfaces for querying object structure.

Example. We recast the example of Sect. 1 in terms of TaskObject, DataObject, and StructureObject. In the example, a file framework defined *FileServer*s and *Client*s. The client C accessed a server S read-only, while the client D overwrote the data it received from S. Therefore C and S could share the same data buffers.

With our new approach, the contract between *FileServer* and *Client* would be reified by a TaskObject, *ReaderTask*, that normally copies the data from server to client. If C treats its data as read only, the designer customizes the *ReaderTask* to not copy the data for the $C-S$ interaction. Alternatively, we could encapsulate the transferred data into a DataObject, *FileData*, and customize its data access methods to avoid the copy.

The remaining problem is to configure the system to use non-copying objects for $C-S$ interactions, and copying ones for $D-S$ interactions. We solve this problem using *FileSystem*, a StructureObject, which creates and maintains associations between clients and DataObjects or TaskObjects. The *FileSystem*, implemented as a FACTORY, represents the framework itself. In our instrumentation system (Sect. 2.3), it would also serve as a data aggregation point.

In the next section, we show how most common optimizations can be described as transformations of TaskObjects, DataObjects, and StructureObjects. The catalog we present gives criteria for deciding when to apply an optimization, and shows how it transforms object interactions. We describe typical implementations as combinations of TaskObjects, DataObjects, and StructureObjects. Examples from the literature are also mentioned.

4 Catalog

Catalog descriptions. In the following catalog, we describe each optimization as follows. First we describe the circumstances under which it is applicable. Then the implementation is described in terms of TaskObject, DataObject, and StructureObject. A TaskObject reifies the sequence of method calls that performs the functions supported by the framework. A DataObject encapsulates the arguments, results, and other data manipulated by a TaskObject. A StructureObject represents object groups and object associations. (See Sect. 3 for more detailed explanations). We also give a number of familiar examples of each optimization. The optimizations and some common examples are summarized in Tab. 2.

Table 2. Types of Optimizations

Optimization	Example
Inter-object caching	File Caching
Optimize common task	Inner loops
Lazy evaluation	File Access Batching
Eager evaluation	Prefetching Memory Pages
Using type information	Copy-on-write sharing
Memory Usage Optimizations	Using buffer pools
Concurrency Optimizations	Using priority locks
Structure Directed Optimizations	Multicast Trees
Miscellaneous Optimizations	Gang Scheduling

How to use the catalog. To use this catalog, first get a clear idea of the task, data, and structure relationships in your framework. Then use the applicability criteria to if they meet the requirements for a particular optimization. The implementation sections, together with the examples from Sect. 5 and Sect. 6, explain how to implement the optimizations.

4.1 Inter-Object Caching

Caching the results of methods in individual objects is common. But we are interested in caching the results of framework tasks distributed across multiple objects.

Applicability Caching is applicable when a task that computes a result may be repeated with the same arguments. We cache the computed results the first time the task executes. For subsequent executions, we return the cached results rather than repeating the computation.

Implementation If the arguments and results of tasks are kept in DataObjects, a cache can be implemented as a collection of DataObjects. In addition, TaskObjects have to be modified to:
- remember results of computations in a cache, and
- consult the cache before repeating a computation.

Caches are located in a StructureObject, where related framework objects may access it.

Examples File block caching in file systems, document caching in Web browsers, *Memoization* in functional and logic programming, index caching in databases.

4.2 Optimize Common Task

Optimizing frequently executed tasks leads to larger gains compared to optimizing tasks that may be seldom executed. In a framework, optimizing a common code path may require breaking encapsulation and refactoring.

Applicability If a task is expected to be used frequently, optimize its code. The optimizations may slow down infrequently used tasks, or compromise encapsulation.

Implementation Optimize the TaskObject for the frequently used task by inlining calls to framework methods. This eliminates the cost of indirection, but violates encapsulation. However, the violations are confined to a single TaskObject. Other ways to optimize tasks include using faster algorithms, and rearranging code to exploit processor caches. Often optimizing common code also needs optimized data access (perhaps using caching).

The optimized TaskObject is a special version of the generic TaskObject. If the framework is changed, the optimized version may be discarded or reprogrammed.

Examples Optimizing the innermost loops in numerical programs, optimizing file systems for small files because most files are small, dynamic code specialization in Synthesis [22], pointer swizzling in object databases.

4.3 Lazy Evaluation

The premise underlying lazy evaluation is that if we delay a task until its results are actually needed, more information may become available that can help optimize the task.

Applicability Lazy evaluation is applicable when all the results from a task execution are not needed immediately and when delaying the computation allows optimizing it.

Implementation TaskObject should include two methods: *prepare()*, that performs the immediate part of the task, and remembers the arguments for the delayed part; and *commit()*, that completes the execution. DataObject should have space to remember the arguments so that it can provide *commit()* with the appropriate context when it is eventually executed.

Examples Lazy write optimization in file systems (Sect. 2.1) uses *Request* to remember arguments (contents of the block and disk address) for write. The prepare part just creates a *Request*. The commit part actually writes the blocks to disk if necessary. It can avoid the write if the blocks have been overwritten, or if the file has been deleted. Even when it does write them, it batches neighboring disk blocks together and writes them using only one disk access.

Other examples: delaying marshaling of arguments in RPC until sending them and using scatter-gather hardware to collect memory fragments (see Sect. 6 for details); direct demultiplexing for network read [1].

4.4 Eager Evaluation

Eager evaluation is the dual of lazy evaluation. It computes some results that may not be needed immediately.

Applicability Eager evaluation is applicable when the incremental cost of computing another result while one result is being computed is low compared to computing it separately, and when the other result is usually needed later.

Implementation Modify the TaskObject so that when it computes the first result, it also computes the other result and saves it in the DataObject.

If the other result is an extension of the first (e.g., fetching two disk blocks, instead of one), modify arguments (e.g., change the number of blocks to read in DataObject) to get the extended result.

Examples Computing checksum of a network packet as a side effect of copying it [6], TCP header prediction [6], prefetching in databases and file systems.

4.5 Using Type Information

Data are typed, so some of their properties are known a priori. Therefore, we can substitute faster algorithms that exploit their properties.

Applicability When data input to a task have special properties, the task can be tailored to take advantage of them. Sometimes, we can instead alter data to suit a given task.

Implementation Implement several customized TaskObjects, one for each type of data. Modify StructureObjects to associate appropriate TaskObject with a given DataObject. StructureObjects usually achieve this using language level polymorphism. Let DataObjects have interfaces for querying their properties. Useful data properties are: size, order, format, location in memory, their source and destination objects, age, and application dependent properties, such as, video, audio, images, text, and large files.

Examples File systems may prefetch file data for sequentially accessed files but not for random access files. Web browsers relax security checks if the applet *source* is the local machine. Operating systems share read-only code segments among processes. Spreadsheets can use finite precision for money but arbitrary precision for other numerical data. Numerical packages supply different algorithms for different *types* of matrices [3] such as positive definite, upper triangular, symmetric, dense, and sparse. They also precondition data for algorithms. Web servers may perform lossy compression on data depending upon the client type (on-line distillation [9]).

4.6 Memory Usage Optimizations

Memory optimizations are among the most widely studied optimizations. The amount of memory available to a framework is limited, and dynamic allocations and releases are slow. Therefore, we need to optimize memory usage for improving speed and reducing the memory footprint.

Applicability Memory optimizations are often applicable to frameworks with dynamic allocations and data movement.

Implementation Most memory optimizations are implemented by tailoring TaskObjects to streamline data movement, to exploit temporal locality, and to reuse memory. Also, DataObjects should take advantage of type information to improve spatial locality, to provide fast access, and to reduce dynamic allocations. Typical memory optimizations are described below.

- **Avoid copies** Hand off DataObject from one object to another in a TaskObject, copy on write, share read-only DataObjects.
- **Avoid dynamic allocation** Use a pre-allocated pool of DataObjects. Pool size can be estimated using arrival rate of allocations and time between allocations and corresponding releases. Using StructureObjects, allocate related objects together instead of allocating each object individually. If allocation and release happen in the same scope, use stack allocation.
- **Avoid accessing memory** Use pre-filled DataObject to avoid writing the common fields repeatedly, use scatter-gather I/O to collect memory fragments.
- **Exploit processor caching** Use eager evaluation to compute multiple results on a DataObject (e.g., computing *Packet* checksum while copying

it benefits from the fact that the *Packet* may be in the cache), special layout of DataObject for improved cache behavior (e.g., row major layout for matrices accessed in row major order).

Examples RPC frameworks avoid dynamic allocations using buffer pool for requests, operating systems use copy-on-write to share pages, networking frameworks use data structures like `mbuf` [16] and x-kernel messages [12] to add or remove protocol headers without copying. The resource exchanger pattern [28] combines many of these optimizations.

4.7 Concurrency Optimization

Concurrent operations on shared objects must be serialized in a framework. When serialization is implemented using locks, the framework must avoid cyclic dependencies. This involves analyzing all objects in a framework. Poor concurrency management may lead to poor performance due to contention hot-spots (many threads contending for access to same data), hard to detect race conditions, locking overhead, and deadlocks.

Applicability If a framework supports concurrent tasks that operate on shared objects, concurrency optimizations may be needed.

Implementation Use a TaskObject to localize all lock acquisition and release operations over a task. Thereafter, it can be specialized to implement different trade-offs. Hot-spot contention can be reduced by splitting critical sections, and locking overhead can be reduced by putting many objects under the protection of one lock [19]. On a uniprocessor, TaskObject can be changed to manipulate interrupts and scheduling to reduce locking overhead.

A StructureObject can associate different types of locks (e.g., read lock and read-write lock) with a DataObject for different access types, and group objects that are used together under one lock (see Sect. 6 for an example). StructureObject objects can explicitly represent lock hierarchies [19] and make them obvious and easy to understand.

DataObjects can be used as tokens for implementing mutual exclusion. If DataObjects are maintained in different memory pools and caches, contention for the global heap is reduced.

Continuations, which may reduce thread usage, can be implemented using TaskObjects and DataObjects (see Sect. 5.3 for examples).

Examples Databases use intention mode locking [21] to improve concurrency and avoid deadlock. Parts of operating systems disable interrupts to implement critical sections. Array locks [7] distribute accesses over a larger region of memory to reduce contention in multiprocessors.

4.8 Structure Directed Optimization

Just as algorithms are optimized for a given data structure, framework tasks can be optimized for a given organization of framework objects.

Applicability If the framework objects are interconnected in regular structures like rings or trees, then tasks can take advantage of the structure. If an object knows its role in the overall framework organization, it can often reduce computation or communication.

Implementation Reify the object interconnection structure in a StructureObject. TaskObjects may use methods of the StructureObjects. The particular optimizations depend on the application.

Examples The instrumentation example (Sect. 2.3) reifies structure (e.g., *PatternInstrument* and *FrameworkInstrument*) which helps reduce volume of data and minimizes system perturbation. Distributed mutual exclusion becomes easy if the processes have known structures, such as a ring. Multicast protocols use *MulticastTree* to optimize data communication among a group of processes. For other examples, see the uses of StructureObjects in Sect. 4.7.

4.9 Miscellaneous

Scheduling Scheduling optimizations are usually applicable to system level frameworks. Specialized schedulers encapsulate optimizations and StructureObjects associate applications with them.

Gang scheduling schedules a group of cooperating processes together. Different *Real-time scheduling* algorithms [17] implement policies for executing tasks so that they meet their deadlines. Priority inheritance scheduling [32] prevents unbounded priority inversion in real-time systems. *Message driven scheduling* schedules tasks when their message arrive.

Protocols Knowledge about the *type* of communication, transport medium, and data helps design custom protocols to optimize factors such as latency, buffer space, and throughput. Multimedia transmission protocols can be optimized for bursty loss in the Internet to avoid retransmissions [5]. Protocols for mobile computing are optimized to reduce expensive transmission from mobile hosts. Parallel distributed programs use protocols that exploit communication patterns [14].

Communication Optimizations In addition to protocols, communication across endpoints can also be optimized. Use asynchronous (non-blocking) communication for overlapping communication with computation. Avoid data conversion to a common format if both endpoints use same data representation. Since a DataObject encapsulates information exchanged between framework objects, they also represent the data to be exchanged across the network when a framework is extended to include communication. (E.g., *VMData* when we add distribution to virtual memory, see Sect. 5).

This completes the catalog. In the next section, we show how to use the catalog for optimizing a virtual memory framework and a distributed object framework.

5 Optimizing a Virtual Memory Framework

This section describes optimizations for the virtual memory framework[4] of the Choices operating system [25]. The virtual memory framework uses the networking framework to implement distributed shared memory. We show how our techniques help in optimizing interactions between these two frameworks.

5.1 Framework Description

The virtual memory framework provides memory management to user processes. A user process issues virtual addresses in its address space, and the virtual memory system translates them to physical addresses, retrieving memory contents from disk when necessary. A user process may also manipulate regions (ranges of addresses) of the address space. For example, a process may map files (or parts thereof) to regions with read, write, or execute permissions.

Basic Objects. These requirements suggest the following:

- *Domain*s that represents the address space,
- *MemoryObject*s that represent datasets like files,
- *MemoryObjectView*s that represents parts of memory objects,
- *MemoryObjectCache*s that maintain the physical pages that hold the contents of *MemoryObject*s,
- *AddressTranslation*s that manage virtual memory hardware, and
- *PageStore* that maintains unused physical pages.

These objects are tables. For example, *Domain*s map virtual addresses to *MemoryObjectView*s; *MemoryObjectView*s are ranges in *MemoryObject*s. Framework operations like pagefault manipulate these tables.

A pagefault task begins with the faulting address, visits *Domain* to get the *MemoryObject* for that address, visits *MemoryObject* to locate the *MemoryObjectCache*, and reads the contents from the *MemoryObject* into a *Page*, allocating one from *PageStore* if necessary. It then updates *AddressTranslation*, mapping the *Page* to the faulting virtual address.

Interaction Objects. Ordinarily, tasks like pagefault processing are embedded in the framework as a sequence of object method calls as described above. But in our framework, we reify them as suggested in Sect. 3.

Tasks like pagefault and pageout become *PageFaultTask* and *PageOutTask*. Their parameters, results, and the objects they visit are recorded in a *VMData* object. The virtual memory system as a whole is represented by *VMSystem*, a StructureObject implemented as a FACADE [10]. *Domain* also serves as a StructureObject, representing the virtual memory objects associated with particular processes.

[4] The idea of optimizing via reified object relationships was developed while implementing this framework; our description pretends as if this idea was applied from the beginning.

VMSystem provides methods like *pageFault()* that initiate framework tasks. For example, *VMSystem::pageFault()* creates a new *VMData* and invokes *PageFaultTask::execute(VMData)*.

5.2 Applying Optimizations

In this section, we describe a few sample optimizations from the catalog (Sect. 4).

Caching. Multi-threaded programs exhibit page-in locality, where many processes fault on a shared page in quick succession. Therefore, store the *VMData* objects associated with recent *PageFaultTask*s in a cache. *PageFaultTask*s consult the cache before processing. Other tasks like *MMapTask*, that maps files, can also use the cache.

Common Task Optimization. Typically, *PageFaultTask* operates on a single page, and only needs to update *AddressTranslation*, or refresh the hardware cache, or change memory access levels. We optimize this by creating a special *SinglePageFaultTask*. This task directly accesses internal tables and uses locks that grant priority to single-page page faults.

Lazy Evaluation. When swapping a page out, user processes must be prevented from accessing the page. So the *AddressTranslation* hardware table entries must be removed immediately. But the page contents can be written to disk at leisure. Moreover, if the page that is to be swapped is accessed again, then we can simply map the page in, eliminating disk write and read. Therefore we use lazy evaluation for pageout. *PageOutTask* has two methods, *makePageInaccessible()* and *pageFlush()*. After executing *makePageInaccessible()*, the associated *VMData* is put in a flush queue. If the page is eventually flushed, then the pageout daemon will execute *pageFlush()*; if the page is paged in, the *VMData* information will be used to make the page accessible.

Eager Evaluation. When a virtual page is accessed, the next page is also likely to be accessed. Also, paging in two pages from disk at one time is cheaper than paging them in separately. We implement this by changing *PageFaultTask*. When *PageFaultTask* encounters a missing physical page, it allocates two pages from the *PageStore* and page them in both at once, provided both are in the same *MemoryObject*.

Using Type Information. For certain memory objects, the style of access is known. For example, pages with video data may be accessed first in first out. The user can provide such type information and policies that use the information to influence pageout. The *VMSystem* associates the memory objects and policies when the objects are initialized by the virtual memory system.

Memory Usage Optimization. The virtual memory framework uses its own heaps to avoid contention. The heap is designed for *VMData* and various table entries used by the virtual memory objects.

5.3 Extensions and Intra-Framework Optimizations

The framework described above was enhanced to support copy-on-write and distributed shared memory. Implementing these enhancements to the virtual memory framework was simplified because of the first-class relationship objects.

Copy-on-write support adds StructureObjects that represent groups of copy-on-write pages. *PageInTask* is changed to distinguish between read and write accesses, and interact with the copy-on-write objects if necessary. *PageOutTask* requires no changes, except that the policy that determines suitable pages for page out is affected.

Distributed Shared Memory requires network communication of pages. Various optimizations can be used to tune the interaction between virtual memory and network frameworks.

For example, the *VMSystem* detects if a page fault is on a distributed shared memory object. It then allocates a special *VMData* that is pre-formatted as a network packet. *PageFaultTask* hands this buffer to the network framework which uses it as its network packet. This is an optimization that uses *type information* (Sect. 4.5).

Similarly, we can reduce thread usage using TaskObjects and DataObjects to implement continuations. Normally, when a *PageFaultTask* is waiting for a network message, the thread executing it is suspended. But threads are scarce resources, so it is important to use them carefully. So *PageFaultTask* provides two methods: *getPhysicalPage()* and *installPage()*. After completing *getPhysicalPage()*, the thread executing *PageFaultTask* enqueues the corresponding *VMData* in a pending queue. When the page arrives from the network, a new thread calls *PageFaultTask::installPage(VMData)* to continue fault processing. This is a *lazy evaluation* (Sect. 4.3) and *concurrency* optimization (Sect. 4.7).

Because the concurrency control was encapsulated in the TaskObjects, it was easy to ensure that adding the new objects and changing the TaskObjects for new responsibilities did not create deadlock. The *VMData* objects suggested the network optimizations and also the generalization to use continuations. Besides interaction objects, our framework uses other ideas like object-oriented state machines [27] and resource exchanger [28] to simplify the implementation.

6 Optimizing a Distributed Object Framework

In this section we consider optimizations in *Quarterware*, a framework for distributed object computing [37].

6.1 Framework Description

A distributed object framework performs two basic tasks:

1. A local method invocation on the client side is converted into a remote request. The subtasks are: marshal the arguments to the call, make a request packet, send it over the network, then wait for the reply packet (if any), and unmarshal the reply.

2. The server side receives a request from the network and converts it to a local method invocation on the indicated server object. This task must receive a request, find the server object, unmarshal the arguments, invoke the requested method on the server, and marshal and send the reply, if any.

Basic Objects. This suggests the following basic objects for this framework (similar to TAO [29] or Orbix [13]).

- *ObjectRef*s represent remote objects locally. A client invokes methods on an *ObjectRef*.
- *ActualObject*s are server objects that implement the desired functionality. Calls on *ObjectRef*s are forwarded to the corresponding *ActualObject*s.
- *Argument*s represent the arguments to a method invocation (call).
- *Result*s represent the results of a call returned by the server (*ActualObject*).
- *Channel*s represent communication media.
- *MemoryAllocator*s provide memory to hold *Argument*s and *Result*s.

Interaction Objects. The basic objects provide methods to perform individual operations. As discussed in Sect. 3, we connect them through the following interaction objects.

- *MarshalTask*s produce network representation of *Argument*s and *Result*s. They may use conventional representations such as XDR, or non-conventional marshaling mechanisms such as compression or distillation [9].
- *ClientCallTask*s implement the client side of a remote call. It involves allocating memory, invoking *MarshalTask*, sending the marshaled data on the corresponding *Channel* and doing the inverse processing for the *Result*.
- *ServiceTask*s implement the server side of a remote call. It involves receiving, unmarshaling and dispatching the request, and marshaling and returning the results.
- *CallData* encapsulates data needed for performing an invocation. It contains memory for holding *Argument*s and *Result*s, references to the *Channel*, names of *ActualObject*, method to be invoked, *MarshalTask* to use, etc.
- *CallEnvironment*s are StructureObjects that associate these objects with one another. For example, a *CallEnvironment* associates *MemoryAllocator*s, *Channel*s, *ServiceTask*s, and *ActualObject*s on the server side.

6.2 Applying Optimizations

This section illustrates a few example optimizations from the catalog (Sect. 4).

Optimizing Memory Usage. We use the *type* information for arguments, results, and *Request*s (Sect. 4.5) to optimize memory usage. Other memory usage optimizations (Sect. 4.6) [11, 8] are as follows.

- Client stubs use *stack allocation* of *CallData*s if argument sizes are known at compile time and invocation is synchronous. (*Avoid dynamic allocation, Sect. 4.6.*)

- Single threaded *ClientCallTask*s (and *ServiceTask*s) reuse the same *CallData* instance across a sequence of calls. (*Avoid dynamic allocation, Sect. 4.6.*)
- *MarshalTask* uses block copy (using `memcpy()`) for marshaling arguments that reside in contiguous memory. (*Using type information, Sect. 4.5.*)
- *ClientCallTask*s use pre-filled *CallData*s to avoid writing the header fields such as the name of actual object. (*Avoid accessing memory, Sect. 4.5.*)
- *MemoryAllocator*s provide two *types* of buffers — one for fixed size *CallData*s, and the other for variable size *CallData*s. Fixed size buffers incur *no dynamic allocation* overhead. (*Using type information, Sect. 4.5.*)
- *MemoryAllocator*s for multi-threaded servers use a *BufferPool*. A good estimate for the *size* of the pool is the expectation of *live* requests in the system, i.e., the sum of stored (enqueued) requests and the number of active threads (each thread processes one request). (*Avoid dynamic allocation, Sect. 4.6.*)

Optimizing the Common Tasks.

- *Fuse* the *MarshalTask* and *ClientCallTask* for single threaded clients, reducing function call overhead.
- We optimize for *short requests* that have either few or no arguments, using a special *ShortCallData*. It has a small inline buffer to hold the request. This reduces dynamic allocation overhead.
- In *homogeneous systems*, data representations on the sender and receiver are identical. So *MarshalTask* for such systems does not not perform data conversion.
- With *one way requests*, the sender does not expect a reply. We modify *ClientCallTask*s and *ServiceTask*s to recognize this case and not allocate resources for the *Result*.

Structure and Concurrency Optimizations. Before executing a request, the *ServiceTask* needs to increment reference counts on *OutputChannel*, *ActualObject*, and *BufferPool*. Since these objects are shared, the reference counts increments have to be protected by locks. We can reduce the number of lock operations by incrementing the reference count on *CallEnvironment* only. Since it already groups *Channel*, *Target*, and *BufferPool*, these objects are referenced automatically (see Sect. 4.7 and 4.8).

Lazy Evaluation. Usually marshaling arguments consists of linearizing the arguments in a buffer. Marshaled arguments are then sent over the *Channel*. This involves a memory-to-memory copy. If the system supports scatter-gather I/O (e.g., Unix `sendv()` call), then we can delay linearization of arguments, and combine it with *Channel::send()*. This way we can avoid the intermediate copy.

We implement this using the *lazy evaluation* pattern (Sect. 4.3) from the catalog as follows. (See Fig. 2 for implementation.)

Specializing DataObjects Customize

  - *CallData* on the client side to remember arguments rather than copying them.

– *CallData* on the server side to remember the input *Channel*.

Specializing TaskObjects Modify

– *MarshalTask::prepare()* to remember arguments.
– *MarshalTask::commit()* to linearize and write remembered arguments to the *Channel* using scatter-gather I/O.
– *ServiceTask::prepare()* to only read the request header, and remember the *InputChannel*.
– *ServiceTask::commit()*, called just before executing the request, to read the arguments from the *InputChannel* into the application memory.

```
ObjectRef::method(Argument a) {
  Result r;
  CallData cd = CDAllocator.alloc(a);
  marshaler.marshal(cd, a);
  clientCallTask.invoke(cd);
  marshaler.unmarshal(cd, r);
  return r;
}
Marshaler::marshal(CallData cd,
                   Argument a) {
  cd.buffer.copy(a.buffer);
}
ClientCallTask::invoke(
                CallData cd) {
  channel.send(cd);
  channel.receive(cd);
  return;
}
```

(a) Simple implementation

```
Marshaler::prepare(CallData cd,
                   Argument a) {
  cd.remember(a);
}
Marshaler::commit(CallData cd,
                  Channel c) {
  c.gatherSend(cd.remembered);
}
ClientCallTask::invoke(
                CallData cd) {
  marshaler.commit(cd, channel);
  channel.receive(cd);
  return;
}
```

(b) Lazy eval. on client-side

Fig. 2. Implementing lazy evaluation optimization in *Quarterware* (Sec. 6)

Optimizing Thread Usage and Scheduling. A server faces various trade-offs between response time and resource usage. For example:

– Optimize response time using more resources (threads and buffers). The extreme case is to use a new thread and a new buffer for each new request.
– Conserve resources at the cost of deteriorating response time. The extreme case is to serialize request processing using only one thread and one buffer.

– An intermediate solution is to dedicate one thread and one buffer for each client and execute requests from different clients in parallel, and those from the same clients one after the other.

We implement different versions of *ServiceTask*s to represent these trade-offs (similar to *invocation strategies* [29]). Some of the versions are:

– *STSequential* always associates the same thread with each *CallData*, i.e., task processing is sequential.
– *STMultiThreaded* creates a new thread for each task and implements thread-per-message strategy.
– *STThreadPerClient* has one thread for each client, and it uses the *inputChannel* attribute of the *CallData* to associate it with the corresponding thread.
– *STRealTime* executes real-time scheduling for requests [36]. In this case, we also have to the *Argument* format to carry additional attributes, such as, priority, and deadline.
– *STSynchronized*, like guarded commands, schedules a request subject to satisfaction of some synchronization constraints.

7 Related Work

Related work falls in two categories: work on networking optimizations and systems design, and software engineering work on reifying object relationships.

The end-to-end argument in system design was introduced by Saltzer et. al. [24]. They describe the implications of end-to-end argument for system structuring. They have argued that many system-wide properties can not be implemented without application-specific knowledge. This argument forms the basis of the insight that implementing optimizations requires changes not only to objects, but also to their interactions.

The research on Paths [20] comes closest to our work. It introduces a path as an explicit abstraction that connect the layers in an operating system and describes optimizations along a path.

Clark [6] and others describe many optimizations for network protocols. In [1] Abbot et. al. describe "Integrated Layer Processing (ILP)," their technique for optimizing protocols. ILP also relies upon fusing protocol layers to reduce copying and expediting protocol processing.

Schmidt et. al. [11] document a comprehensive set of optimizations for Object Request Brokers (ORBs). This paper is among the first works on applying networking optimizations to distributed object systems.

We distinguish our work from the above in that we show how to apply optimizations to arbitrary frameworks. Our ideas are more general than Paths [20] since we go beyond control paths: we explicitly represent data traveling along the path, as well as the structure of object interconnections. We also show how most common optimizations from the literature can be represented as transformations of control, data, and structure. In addition, we present standard techniques for describing applicability and implementations of these optimizations.

There have been many proposals to give first-class status to relationships between system components. The purpose of these proposals is to ease evolution of systems and make them easier to understand.

Reflective systems [38, 23] advocate reifying many aspects of objects. Kiczales [15] presents a design methodology, "Open Analysis and Design," which suggests incorporating reflective interfaces for customization.

Shaw [33] argues that the connections between components deserve first-class status. For example, encoding a pipe that connects two communicating programs with procedure calls obscures the data flow aspects. This paper was a starting point for our work. An architecture description language UniCon [34] and a formal description language Wright [2] explicitly model connections.

The Demeter project introduces a notation for making the structure of frameworks explicit, and express framework traversals as regular expressions [18]. This helps in framework evolution, because changes to the object structure can be expressed by changing the structure description rather than method code.

Our approach shares the advantages of reification and encapsulation with these works. However, our focus is different. We concentrate on optimizations, their representations, and their role in a framework. We use reifications as a tool to implement our optimizations.

8 Conclusion

This paper showed how to design frameworks that are easy to optimize. The design techniques allow framework users to specialize not only individual objects, but also optimize the interactions between objects. To make such customization easy, object interactions are themselves expressed as objects, and tailored using domain-specific information. Common optimizations can be expressed by modulating object relationships.

Relationship objects localize the changes required for optimization. As a result, a framework user does not have to hunt through the framework to understand the optimization. Similarly, changes to framework objects and optimizations may interact, but their interplay is made explicit by the relationship objects. This eases framework evolution and refactoring.

We also presented a catalog of optimizations that helps designers determine suitable optimizations and implement them in a systematic manner. The catalog gives designers a common vocabulary to describe optimizations and understand why and how they work. Our experience suggests that these techniques are useful in practice.

This research suggests several issues for future work. One direction is to expand the catalog of optimizations and improve the descriptions. It is remarkable that so many optimizations can be described in a uniform way by tuning object interconnections; perhaps this idea may be generalized further. It would be interesting to see whether programming languages can directly support our implementation techniques. Another possible development is to include our implementation techniques in refactoring tools.

Acknowledgments. We gratefully acknowledge help from Fabio Kon and Amitabh Dave (UIUC), John Heintz (Icon Computing), and Sharad Singhai (UMass) for their comments, and thank Monika Chandak (UIUC) for support and for loaning her machine to us.

References

1. Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.
2. Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997. CMU-CS-97-144.
3. Aart J. C. Bik and Harry A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 768 of *Lecture Notes in Computer Science*, pages 57–75. Springer Verlag, August 1993.
4. Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Experiences designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.
5. Zhigang Chen. Coding and transmission of digital video on the internet. Technical Report UIUCDCS-R-1997-2016, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.
6. D. Clark, V. Jocabson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, Jun 1989.
7. A. Dave, N. Islam, and R. H. Campbell. A low-latency scalable locking algorithm for shared memory multiprocessors. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, pages 10–17. IEEE Computer Society Press, October 1994.
8. Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997.
9. Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170. ACM, October 1996.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
11. A. Gokhale and D. C. Schmidt. Principles for optimizing corba internet inter-orb protocol performance. In *Proceedings of the 31st Hawaii International Conference onf System Sciences*, pages 375–386, January 1998. (to appear).
12. N. C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
13. IONA Technologies. *The Orbix Architecture*, November 1996. Available at `http://www.iona.com/Products/Orbix/Architecture/index.html`.
14. N. Islam, A. Dave, and R. H. Campbell. Communication compilation for unreliable networks. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 188–195, May 1996.

15. Gregor Kiczales, John Lamping, Cristina Videira Lopes, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–490, May 1997.
16. S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison Wesley, 1988.
17. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.
18. Cristina Videira Lopes and Karl Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 81–99. Springer Verlag, July 1994.
19. Paul E. McKenney. Selecting locking primitives for parallel programming. *Communications of the ACM*, 39(10):75–82, October 1996.
20. David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 153–167. USENIX, October 1996.
21. Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.
22. Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
23. *Reflection'96*, San Francisco, CA, April 1996.
24. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
25. Aamod Sane. *Techniques for Developing Correct, Fast, and Robust Implementations of Distributed Protocols.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, December 1997.
26. Aamod Sane and Roy Campbell. Composite messages: A structural pattern for communication among software components. In *OOPSLA'95 Workshop on Design Patterns in Concurrent, Distributed, and Parallel Object-Oriented Systems*, 1995.
27. Aamod Sane and Roy Campbell. Object-oriented state machines: Subclassing, composition, delegation, and genericity. In *Proceedings of the OOPSLA*, pages 17–32. ACM, October 1995.
28. Aamod Sane and Roy Campbell. Resource exchanger: A behavioral pattern for low overhead concurrent resource management. In *Pattern Languages of Program Design.* Addison Wesley, 1996.
29. Douglas C. Schmidt and Chris Cleeland. Applying patterns to develop extensible and maintainable orb middleware. Available at `http://www.cs.wustl.edu/\~schmidt/ORB-patterns.ps.gz`.
30. Mohlalefi Sefika. *Design conformance management of software systems: An architecture-oriented approach.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, August 1996. UIUCDCS-R-96-1974.
31. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-oriented visualization. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 31(10) of *ACM SIGPLAN Notices*, pages 389–405, October 1996.
32. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
33. M. Shaw. Procedure calls are the assembly language of software interconnections: Connectors deserve first class. In D. Lamb, editor, *Studies of Software Design*,

volume 1078 of *Lecture Notes in Computer Science*, pages 17–32, Baltimore, Md, May 1993. Springer Verlag.

34. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.
35. Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison Wesley, 1997.
36. Ashish Singhai, Aamod Sane, and Roy Campbell. Reflective ORBs: Support for robust, time-critical distribution. In *Proceedings of the ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*, volume (to appear) of *Lecture Notes in Computer Science*. Springer Verlag, June 1997.
37. Ashish Singhai, Aamod Sane, and Roy Campbell. Quarterware for middleware. In *Proceddings of the 18th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, May 1998.
38. Brian C. Smith. *Reflection and Semantics in A Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982. MIT-LCS-272.
39. Habanero project, 1997. Available at: `http://http://www.ncsa.uiuc.edu/SDG/Software/Habanero/`.
40. Lim Swee. *Adaptive Caching in a Distributed File System*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, November 1995.