Dynamic Function Placement in Active Storage Clusters

Khalil Amiri David Petrou Gregory Ganger
Garth Gibson

June 1999

CMU–CS–99–140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Optimally partitioning application and filesystem functionality within a cluster of clients and servers is a difficult problem due to dynamic variations in application behavior, resource availability and workload mixes. This paper presents ABACUS, a run-time system that monitors and dynamically changes function placement for applications that manipulate large data sets. Several examples of data-intensive workloads are used to show the importance of proper function placement and its dependence on dynamic run-time characteristics, with performance differences frequently reaching 2–10X. We evaluate how well the ABACUS prototype adapts to run-time system behavior, including both long-term variation (e.g., filter selectivity) and short-term variation (e.g., multi-phase applications and inter-application resource contention). Our experiments with ABACUS indicate that it is possible to adapt in all of these situations and that the adaptation converges most quickly in those cases where the performance impact is most significant.

This research is sponsored by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174–96–0002. Additional support was provided by the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Hitachi, IBM, Intel, Quantum, Seagate Technology, Siemens, Storage Technology, Wind River Systems, 3Com Corporation, Data General/Clariion, and LSI Logic.

Keywords: C.2.4 [**Computer Systems Organization**]: Distributed systems, C.4 [**Computer Systems Organization**]: Performance of systems, D.2.9 [**Software**]: Management – *Software configuration managenent*, D.4.2 [**Software**]: Operating Systems – *Storage management*, D.4.3 [**Software**]: Operating Systems – *Filesystems management*, D.4.7 [**Software**]: Operating Systems – *Organization and design.*

1 Introduction

Effectively utilizing cluster resources remains a difficult problem for distributed applications. Because of the relatively high cost of remote versus local communication, the performance of a large number of these applications is sensitive to the distribution of their functions across the network. As a result, the effective use of cluster resources requires not only load balancing, but also proper partitioning of functionality among producers and consumers. While software engineering techniques (e.g., modularity and object orientation) have given us the ability to partition applications into a set of interacting functions, we do not yet have solid techniques for determining where in the cluster each of these functions should run, and deployed systems continue to rely on complex manual decisions made by programmers and system administrators.

Optimal placement of functions in a cluster is difficult because the right answer is usually, "it depends." Specifically, optimal function placement depends on a variety of cluster characteristics (e.g., communication bandwidth between nodes, relative processor speeds among nodes) and workload characteristics (e.g., bytes moved among functions, instructions executed by each function). Some are basic hardware characteristics that only change when something fails or is upgraded, and thus are relatively constant for a given system. Other characteristics can not be determined until application invocation time, because they depend on input parameters. Worst of all, many change at run-time due to phase changes in application behavior or competition between concurrent applications over shared resources. Hence, any "one system fits all" solution will cause suboptimal, and in some cases disastrous, performance.

In this paper, we focus on an important class of applications for which clusters are very appealing: data-intensive applications that selectively filter, mine, sort, or otherwise manipulate large data sets. Such applications benefit from the ability to spread their data-parallel computations across the source/sink servers, exploiting the servers' computational resources and reducing the required network bandwidth. Effective function partitioning for these data-intensive applications will become even more important as processing power becomes ubiquitous, reaching devices and network-attached appliances. This abundance of processing cycles has recently led researchers to augment storage servers with support for executing application-specific code. We refer to all such servers, which may be Jini-enhanced storage appliances [Sea99], much-evolved commodity active disks [RGF98, KPH98, AUS98], file servers allowing remote execution of applications, or advanced RAID controllers, as *programmable storage servers*, or simply *storage servers*.

In addition to their importance, we observe that these data-intensive applications have characteristics that simplify the tasks involved with dynamic function placement. Specifically, these applications all process and move significant amounts of data, enabling a monitoring system to *quickly* learn about the most important inter-object communication patterns and per-object resource requirements. This information allows the run-time system to rapidly identify functions that should be moved to reduce communication overheads or resource contention.

We call our system ABACUS, because functions associated with particular streams are moved back and forth between clients and active servers in response to dynamic system conditions. In our implementation, programmers explicitly partition the functions associated with data streams into distinct components, conforming to an intuitive object-based programming model. The ABACUS run-time system monitors the resource consumption and communication of these components, without knowing anything about their internals (black box monitoring). The measurements are used with a cost-benefit model to decide when to relocate components to more optimal locations.

In this paper, we describe the design and implementation of ABACUS and a set of experiments evaluating its ability to adapt to changing conditions. Specifically, we explore how well ABACUS adapts to variations in network topology, application cache access pattern, application data reduction (filter selectivity), contention over shared data, phases in application behavior, and dynamic competition for resources by concurrent applications. Our preliminary results are quite promising; ABACUS often improves application response time 2–10X. In all of our experiments, ABACUS selects the best placement for each function, “correcting” placement when the function is initially started on the “wrong” node. Further, ABACUS outperforms all one-time placements in situations where dynamic changes cause the proper placement to vary during an application’s execution. ABACUS is able to effectively adapt function placement based on only black box monitoring, removing from programmers the burden of considering function placement.

The remainder of this paper is organized as follows. Section 2 discusses how ABACUS relates to prior work. Section 3 overviews the design of ABACUS. In Section 4, we describe the ABACUS programming model and several example applications built upon it. Section 5 describes the run-time system, focussing on the resource monitoring and placement decision components. In Section 6, we present a variety of experiments to demonstrate the value of dynamic function placement and ABACUS’s ability to rapidly detect and adapt to dynamic conditions. Section 7 summarizes the paper’s contributions.

2 Related work

There exists a large base of excellent research and practical experiences related to code mobility and cluster computing — far too large to fully enumerate here. Our work focusses on how to adapt function placement to dynamic cluster and workload characteristics. Here, we discuss previous work on adaptive function placement and how it relates to ABACUS.

Several previous systems such as Coign and others [HS99, MvD76, HF75, FJK96] have demonstrated that function placement decisions can be automated given accurate profiles of inter-object communication and per-object resource consumption. All of these systems use long-term histories to make good installation-time or invocation-time function placement decisions. ABACUS complements these previous systems by looking at how to dynamically adapt placement decisions to run-time conditions.

River [ADAT$^+$99] is a system that dynamically adjusts per-consumer rates to match production rates, and per-producer rates to meet consumption rate variations. Such adjustments allow it to adapt to run-time non-uniformities among cluster systems performing the *same* task. ABACUS complements River by adapting function placement dynamically in the presense of *multiple* different tasks.

Equanimity is a system that, like ABACUS, dynamically rebalances service between a single client and its server [HF93]. ABACUS builds on this work by developing mechanisms for dynamic function placement in realistic cluster environments, which include such complexities as resource contention, resource heterogeneity, and workload variation.

Process migration systems such as Condor [BLL91] and Sprite [DO91] developed mechanisms for coarse-grain load-balancing among cluster systems, but did not explicitly support fine-grain function placement or adapt to inter-function communication. Mobile programming languages and environments such as Emerald [JLHB88] and Rover [JTK97] do support fine-grain mobility of application objects, but they leave object migration decisions to the application programmer. ABACUS builds on such work by providing run-time system mechanisms that automate migration decisions.

3 Overview of ABACUS

To explore the benefits of dynamic function placement, we designed and implemented the ABACUS prototype system and ported to it a diverse set of test applications. ABACUS consists of a programming model and a run-time system. Our goal was to make the programming model easy for application programmers to use. Further, we wanted it to simplify the task of the run-time system in migrating functions and in monitoring the resources they consume. As for the run-time system, our goal was to achieve low monitoring overhead and improved performance through effective placement. Moreover, it was designed to scale to large cluster sizes. Our first ABACUS prototype largely meets these goals.

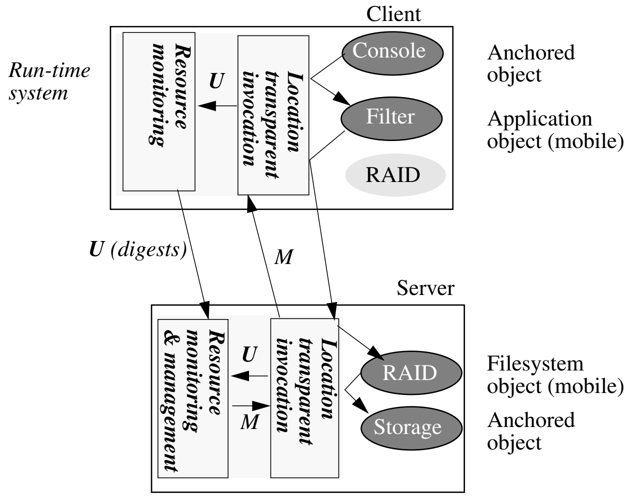

Figure 1: An illustration of an ABACUS object graph, the principal ABACUS components, and their interactions. This example shows a filter application accessing a striped file. Functionality is partitioned into objects. Inter-object method invocations are transparently redirected by the location transparent invocation component of the ABACUS run-time. This component also updates the resource monitoring component on each procedure call and return from a mobile object (arrows labeled "U"). Clients periodically send digests of the statistics to the server. Finally, resource managers at the server collect the relevant statistics and initiate migration decisions (arrows labeled "M").

The ABACUS programming model encourages the programmer to compose data-intensive applications from small, functionally independent components or objects. These *mobile* objects provide explicit methods which *checkpoint* and *restore* their state during migration.

At run-time, an application and filesystem can be represented as a graph of communicating mobile objects. This graph can be thought of as rooted at the storage servers by anchored (non-migratable) *storage objects* and at the client by an anchored *console* object. The storage objects provide persistent storage, while the console object contains the part of the application which must remain at the node where the application is started. Usually, the console part is not data intensive. Instead, it serves to interface with the user or the rest of the system at the start node and typically consists of the `main` function in a C/C++ program. This console part initiates invocations which are propagated by the ABACUS run-time to the rest of the graph.

As shown in Figure 1, the ABACUS run-time system consists of (i) a migration and location-transparent invocation component, or *binding manager* for short; and (ii) a resource monitoring and management component, *resource manger* for short. The first component is responsible for the creation of location-transparent references to mobile objects, for the redirection of method invocations in the face of object migrations, and for enacting object migrations. Finally, the first component notifies the second at each procedure call and return from a mobile object.

The resource monitoring and management component uses the notifications to collect statistics about bytes moved between objects and about the resources used by active objects (e.g., amount of memory allocated, number instructions executed per byte processed). Moreover, this component monitors the availability of resources throughout the cluster (node load, available bandwidth on network links). An analytic model is used to predict the performance benefit of moving to an alternative placement. The model also

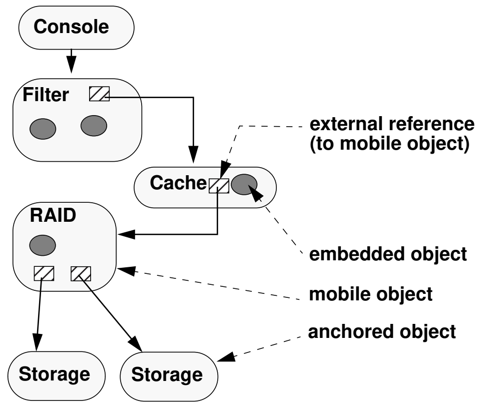

Figure 2: Mobile objects in ABACUS. Objects have private state accessible only through the exported interface. The private state contains either references to embedded objects, or external references to other mobile or anchored objects. Anchored objects do not migrate while mobile objects can. The console represents the part of the application that is not mobile. This part performs processing that is not data-intensive, usually initialization and user/system interface functions.

takes into account the cost of migration — the time wasted to acquiescing and checkpointing an object, transferring its state to the target node, and restoring it on that node. Using this analytic model, the component arrives at the placement with the best *net benefit*. If this placement is different from the current configuration, the component effects object migration(s).

The ABACUS prototype is written in C++. We leverage the language's object-oriented features to simplify writing our mobile objects. For our intra-node communication transport, we use DCE RPC. As the focus of our prototype is on function placement decisions, we leave unaddressed several important but orthogonal mobile code issues. For example, we sidestep the issues of code mobility [Tho97] and dynamic linking [BSP+95] by requiring that all migratable modules be statically linked into an ABACUS process on both clients and servers. Further, issues of inter-module protection [WLA93, NL96] and sandboxing [FHL+96] are not addressed. While these will be important issues for production systems, and we plan to address them in future generations of the prototype, they are tangential to the questions addressed in this paper.

4 Programming model

ABACUS provides two fundamental abstractions to support the efficient implementation of mobile object-based applications and filesystems: *mobile objects* and *mobile object managers.*

4.1 Mobile objects

A mobile object in ABACUS is explicitly declared by the programmer as such. It consists of state and the methods that manipulate that state. A mobile object is required to implement a few methods to enable the run-time system to create instantiations of it and migrate it. Mobile objects are usually of large granularity — they are not meant to be simple primitive types — performing a self-contained processing step that is data intensive, such as parity computation, caching, searching, aggregation, *etc*.

Mobile objects have private state that is not accessible to outside objects, except through the exported interface. The implementation of a mobile object is internal to that object and is opaque to other mobile objects and to the ABACUS run-time system. The private state consists of embedded objects and references to external objects. Figure 2 gives an example of the relationship of some objects.

The mobile object is responsible for saving its private state, including the state of all embedded objects, when its `Checkpoint()` method is called by ABACUS. It is also responsible for reinstating this state, including the creation and initialization of all embedded objects, when the run-time system invokes the `Restore()` method, after it has been migrated to a new node. The `Checkpoint()` method saves the state to either an in-memory buffer or to a file. The `Restore()` method can reinstate the state from either location.

Each storage server (a server with a data store) provides local storage objects exporting a flat file interface. Storage objects are accessible only at the server that hosts them and therefore never migrate. The migratable portion of the application lies between the storage objects on one side and the console object on the other. Applications can declare other objects to be non-migratable. For instance, an object that implements write-ahead logging can be declared by the filesystem as non-migratable, effectively anchoring it to the storage server where it is started (usually the server hosting the log).

Component objects perform their processing when one of their methods is invoked and usually make method invocations to other objects downstream. The amount of data moved in each invocation is an application-specific block of data rather than entire files at once. This is required as an artifact of our monitoring and migration system. ABACUS accumulates statistics on return from method invocations for use in making object migration decisions. If the program makes a single procedure call down a stack of objects, ABACUS will not collect this valuable information until the end of the program, at which point any migration would be useless.

4.2 Mobile object managers

Often, a service is better implemented via a single entity that controls the resources for a group of objects of the same type. For example, a filesystem may wish to control the total amount of physical memory devoted to caching, or the total number of buffers available for prefetching. To support this type of aggregation, a mobile object manager can be used. In the filesystem example, the mobile object manager would wrap the implementation of all cache objects on a given node in the cluster. Object managers provide an interface that is almost identical to that of the types they contain. They take an additional first argument to each method invocation that represents a reference to the managed object to be invoked.

4.3 Examples

To stress the ABACUS programming model and evaluate the benefit of adaptive function placement, we have implemented an object-based distributed filesystem and a few data intensive applications. We describe them in this section and report on their performance in Section 6.

4.3.1 Object-based distributed filesystem

Applications often require a variety of services from the underlying storage system. ABACUS enables filesystems to be composed of explicitly migratable objects, each providing storage services such as reliability (RAID), cacheability, and application-specific functionality. This approach was pioneered by the stackable and composable filesystem work [HP94, KS97] and by the more relevant Spring object-oriented operating system [KN93].

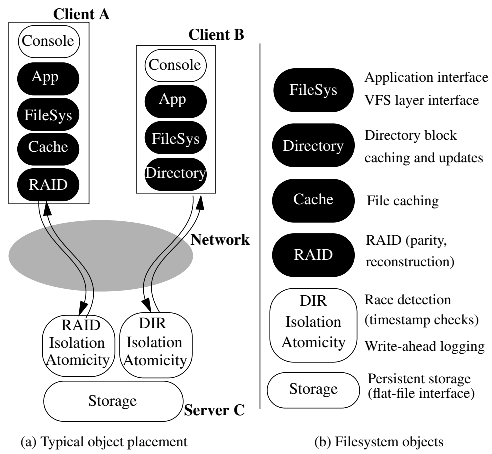

Figure 3: The architecture of the object-based distributed filesystem built atop ABACUS. The figure shows a typical file and directory object stack (a). The object placement shown is the default for high-bandwidth networks and trusted clients. Also shown are the component filesystem objects which are implemented to date and a brief description of their function (b).

Files are usually associated with more than one service. For instance, a file may be cacheable, striped, and reliable. The constructor of a cacheable object requires a backing object to read from and write back to. The backing object can be a storage object, a mirrored object, or even another cacheable object. An object may have references to more than one object from a lower layer, as would be the case if the object is mirrored.

The ABACUS filesystem provides coherent file and directory abstractions atop the flat object space exported by base storage objects. A file is associated with a stack of objects when it is created representing the services that are bound to that file. For instance, only "important" files include a RAID object in their stack. When a file is opened, the top-most object is instantiated, which in turn instantiates all the lower level objects in the object graph. Access to a file always starts at the top-most object in the stack and the run-time system propagates accesses down to lower layers as needed.

The prototype filesystem is distributed. Therefore, it must contain, in addition to the layers that are typically found in local filesystems (such as caching and RAID), services to support inter-client file and directory sharing. In particular, the filesystem allows both file data and directory data (data blocks) to be cached and manipulated at trusted clients. Because multiple clients can be concurrently sharing files, we implement AFS style callbacks for cache coherence [HKM+88]. Similarly, because multiple clients can be concurrently updating directory blocks, the filesystem includes a timestamp-ordering protocol to ensure that updates performed at the clients are consistent before they are committed at the server. This scheme is highly scalable in the abscence of contention because it does not require a lock server or any lock traffic. In Section 6, we describe how ABACUS effectively changes the concurrency control protocol during high contention to a locking scheme by simply adapting object placement.

By default, each file's graph consists of a filesystem object providing the VFS interface to applications, a cache object, an optional RAID 5 object, and one or more storage objects. The cache object keeps an index of a particular object's blocks in the shared cache kept by the ABACUS filesystem process. The RAID 5 object stripes and maintains parity for individual files across sets of storage servers. The storage objects provide flat storage and can be configured to use either the standard Linux **ext2** filesystem or our research group's Network-Attached Secure Disks (NASD) prototype [GNI+99] as backing store. Figure 3 shows a sketch of typical file and directory stacks.

Each directory's graph consists of a directory object, an isolation object, an atomicity object, and a storage object. The directory object provides POSIX-like directory calls and caches directory entries. The isolation object provides support for both cache coherence and optimistic concurrency control. The former is provided by interposing on `ReadObj()` and `WriteObj()` calls, installing callbacks on cached blocks during `ReadObj()` calls, and breaking relevant callbacks during `WriteObj()` calls. The latter is provided by timestamping cache blocks [BG80] and exporting a special `CommitAction()` method that checks specified *readSets* and *writeSets*[1] for conflicts. The atomicity object provides atomic recovery for multi-block writes by interposing on the `CommitAction()` method, and ensuring that a set of blocks (possibly from differing objects and devices) are either updated in their entirety or not updated at all. This object employs a shared log object among all of its instances.

The ABACUS filesystem can be accessed in two ways. First, applications that include ABACUS objects can directly append per-file object subgraphs onto their application object graphs for each file opened. Second, the ABACUS filesystem can be mounted as a standard filesystem, via VFS-layer redirection. Unmodified applications using the standard POSIX system calls can thus interact with the ABACUS filesystem. Although it does not allow legacy applications to be migrated, this second mechanism does allow legacy applications to benefit from the filesystem objects adaptively migrating beneath them.

The ABACUS filesystem is interesting, in and of itself, because it allows its functionality to be dynamically migrated based on run-time observations.

4.3.2 Object-based applications

Data-intensive portions of applications can be similarly decomposed into objects that perform operations such as searching, aggregation, or data mining. As data sets in large-scale businesses continue to grow, an increasingly important user application is high-performance search, or data filtering. Filtering is generally a highly selective operation, consuming a large amount of data and producing a smaller fraction. This makes an ideal candidate for execution close to the data, so long as storage resources are available and the benefit of executing this function on the storage nodes outweighs the benefit to competing workloads. The difficulty lies in not statically knowing the selectivity and resource requirements of a filter operation which can change from one invocation to the next, depending on the filter's input stream and parameters.

5 Run-time system

The ABACUS run-time system consists of per-node binding managers and resource managers. Each binding manager is responsible for the instantiation of mobile objects and the invocation of their methods in a location-transparent manner (Section 5.1), and the migration of objects between cluster nodes (Section 5.2). The resource managers are responsible for collecting statistics about resource usage and availability (Section 5.3), and making placement decisions based on the collected statistics (Section 5.4).

5.1 Object instantiation and invocation

The two kinds of nodes in an ABACUS cluster are clients and servers. Servers are nodes on which at least one base storage object resides and clients are nodes which execute applications that access storage servers.

[1] The readSet (writeSet) consists of the list of blocks read (written) by the client. A directory operation such as `MkDir()` requires reading all the directory blocks to ensure the name does not exists than updating one block to insert the new name and inode number. The readSet in this case would contain all the directory blocks and their timestamps and the writeSet would contain the block that was updated.

Since servers can also execute applications, one storage server can potentially be a client of another server.

Applications instantiate mobile objects by making a request to the ABACUS run-time system. For example, when filtering a file, the application console object will request a filter object to be created. The run-time system creates the object in memory by invoking the object manager for that type. If no object manager exists, one is created. Object managers must implement a `CreateObj()` method which takes arguments specifying any initialization information. This method returns a reference that identifies the new object within that object manager. The object manager finally creates the actual object (with the `new` operator in C++) and returns a "manager reference" to the run-time system. If the object is not managed, as is the case for most application objects, the ABACUS run-time will handle its allocation.

ABACUS also allocates and returns to the caller a network-wide unique run-time identifier, called `rid`, for the new object. [2] The caller uses the `rid` to invoke the new mobile object. The `rid` acts as a layer of indirection, allowing objects to refer to other objects without knowing their current location. The ABACUS binding manager interposes between method invocations and uses `rids` to forward them to the object's current location.

ABACUS maintains the information necessary to perform inter-object invocations in a per-node hash table that maps an `rid` to a *(node, object_manager, manager_reference)* triplet, called a location table. As mobile objects move between nodes, this table is updated to reflect the new node, new object manager, and new manager reference. The `rid` is passed as the first argument of each method invocation, allowing the system to properly redirect method calls.

At run-time, this web of objects constitutes a graph whose nodes represent objects and whose edges represent invocations between objects. For objects in the same address space, invocations are implemented via procedure calls, and data is passed without any extra copies. For objects communicating across machines or address spaces, remote procedure calls (RPCs) are employed.

5.2 Object migration

In addition to properly routing object calls, ABACUS binding managers are responsible for enacting migration. Consider migrating a given object from a *source node* to a *target node*. First, the binding manager at the source node blocks new calls *to* the migrating object, quiescing it. Then, the binding manager waits until all active invocations in the migrating object have drained (returned). Migration is cancelled if this step takes too long. Next, the object is checkpointed locally by invoking its `Checkpoint()` method. The object allocates an in-memory buffer to store its state or writes to the filesystem if the checkpoint size is large. This state is then transferred and restored on the storage node. Then, the location tables at the source and target nodes are updated to reflect the new location. Finally, invocations are unblocked and are redirected to the proper node via the updated location table. This procedure extends to migrating subgraphs of objects.

Location tables are not always accurate. Instead, they provide hints about an object's location which may become stale. Nodes other than the source and target who have cached hints about an object's location will not be updated when a migration occurs. However, stale data is detected and corrected when these nodes attempt to invoke the object at the old node. At that time, the old node notifies them that the object has migrated.

For scalability reasons, nodes are not required to maintain forwarding pointers for objects that they have hosted at some point in the past. Consequently, the old node may not be able to inform a caller the current location of an object. In this case, the old node will redirect the caller to the node at which the object originated, called the *home node*. The home node can be easily determined because it is encoded in the object's `rid`. The home node always has up-to-date information about the object's location because

[2] The `rid` consists of a network-wide identifier, which is generated by concatenating the node identifier where the object is created and a local object identifier that is unique within that node.

during each migration its location table is updated in addition to the tables at the source and target nodes. Because objects usually move between a client (an object's home node) and one of the servers, no extra messaging is usually incurred to update location tables during migration.

5.3 Resource monitoring

The resource manager on a given server seeks to perform the migrations that would result in the minimal average application response time across all the applications that are accessing it. This amounts to figuring out what subset of objects executing currently on clients can benefit from computing closer to the data. Migrating an object to the server could potentially reduce the amount of stall time on the network, but it could also extend the time the object spends computing if the server processor is loaded. Thus, while it may be beneficial to "pull" a few data-intensive objects to the server, pulling more objects may end up hurting application response time. Resource managers at the servers use an analytic model to determine which objects should be migrated from the clients to the server and which objects, if any, should be migrated back from the server to the clients.

Resource managers at the clients collect statistics about the resources consumed by active local objects. They periodically send these statistics to the servers that are being accessed by the active objects. Server-side resource managers receive and buffer these statistics and periodically inspect them to determine if a better placement exists. In particular, the required statistics include the rate of data flowing between objects, the amount of memory used by an object, the instructions executed by the object per byte of data processed, the stall time spent at each object waiting for processing at lower layers, and the load of any nodes that are candidates for object placement. We use the run-time system's intermediary role in redirecting calls to collect all the necessary statistics. By only interposing monitoring code at procedure call and return from mobile objects, we do not slow down the execution of methods within a mobile object.

Threads spend their time computing, blocked (stalled), or contending for the processor (on the ready queue). On the same node, threads can cross the boundaries of multiple mobile objects by making method invocations that propagate down the stack and then unwind back to the console object. The ABACUS resource manager must charge the time a thread spends computing or blocked to the appropriate object. Similarly, it must charge any allocated memory to the proper object. The ABACUS run-time collects the required statistics over the previous H seconds of execution, which we refer to as the length of the observation window. The statistics are collected as follows:

Data flow graph. The bytes moved between objects are monitored by inspecting the arguments on procedure call and return from a mobile object. The number of bytes transferred between two objects is then recorded in a timed data flow graph. This graph maintains moving averages of the bytes moved between every pair of communicating objects in a graph. Data flow graphs are of tractable size because most data-intensive applications do the bulk of their processing in a stream-like fashion through a small stack of objects.

Memory consumption. We monitor the amount of memory dynamically allocated by an object as follows. On each procedure call or return from a mobile object, the `pid` of the thread making the call is recorded. Thus, for at any point in time, the run-time system knows the currently processing mobile object for each active thread in that address space. Wrappers around each memory allocation routine (e.g., `malloc`, `free`) inspect the `pid` of the thread invoking the memory allocation routine, and use that `pid` to determine the current object. This object is then charged for the memory that was allocated or freed.

Instructions executed per byte. Given the number of bytes processed by an object, computing the instructions/byte amounts to monitoring the number of instructions executed by the object during the observation window. Since we know the processing rate on a node, this amounts to measuring the amount of time spent computing within an object.

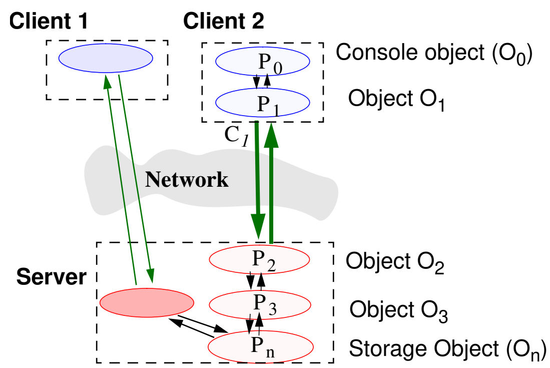

Figure 4: This figure shows how mobile objects interact in a layered system. Most of the invocations are synchronous, starting at the top-most console module and trickling down the layers. An object servicing a request may invoke one or more lower level objects.

We use two mechanisms to measure this time, interval timers and the Pentium cycle counter. Linux provides three interval timers for each thread. The `ITIMER_REAL` timer decrements in real time, `ITIMER_VIRT` decrements whenever the thread is active in user-mode, and `ITIMER_PROF` decrements whenever the thread is active in user or system-mode. We use the `ITIMER_VIRT` and `ITIMER_PROF` timers to keep track of the time spent computing in user/system mode and then charge that time to the currently executing object of a thread.

The only complication is that interval timers have a 10 ms resolution and many method invocations complete in a shorter period of time. To measure short intervals accurately, we use the Pentium cycle counter which is read by invoking the `rdtsc` instruction (using the `asm("rdtsc")` directive within a C/C++ program). Using the cycle counter to time intervals is accurate as long as no context switch has occured within the measured interval. Hence, we use the cycle counter to measure intervals of computation during which no context switches occur, otherwise, we rely on the less accurate interval timers. We detect that a context switch has occurred by seeing if the time reported by `ITIMER_PROF/ITIMER_REAL` and the cycle cycle counter for the candidate interval differ significantly.

Stall time. To estimate the amount of time a thread spends stalled in an object, we need to have more information than is currently provided by the system timers. We extend the `getitimer/setitimer` system calls to support a new type of timer, which we denote `ITIMER_BLOCKING`. This timer decrements whenever a thread is blocked and is implemented as follows. When the kernel updates the system, user, and real timers for the active thread, it also updates the blocking timers on behalf of all the threads in the queue which are marked as blocked (`TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).

Node load. We also measure the load on a given node, defined as the average number of threads in the ready queue over the observation window, H. This value is required to estimate the processing time for an object after migration to a new node given the object's instruction per byte and number of bytes processed.

Linux reports load averages for 1 min., 5 min., and 15 min. via the `/proc/loadavg` pseudo-file. We add an ABACUS specific load average which decays over the past H seconds and report this value as a fourth value in `/proc/loadavg`.

5.4 Dynamic placement

We now describe at a high-level the algorithm employed by ABACUS to optimize object placement. To begin, consider the case of a single stack of objects, as shown in Figure 4. Each invocation trickles down the stack from the client to the server and back. At each layer, some processing is performed and some time is spent transferring data between the layers. We will first derive the application response time for this simple case, then describe how it is extended to multiple concurrent stacks.

5.4.1 Single stack

Consider the stack in figure 4. Invocations start at the console object, O_0, and trickle down until object O_n. Let's denote the processing time expended by an object by P_j, and the "blocking" or "stall" time between objects O_i and O_{i+1} by C_i. Thus, we can represent the response time of an application's invocation as:

$$T_{app} = \sum_{j=0}^{n} P_j + \sum_{j=0}^{n} C_j \tag{1}$$

or the total processing time plus the total stall time.

Let's further denote the total number of bytes processed by an object during the history window H by b_j, and the inherent processing ratio, expressed in instructions/byte for an object, by h_j. Given the average number of tasks in the ready queue over the history window, L_{node}, and the raw processing rate, R_{node}, a particular node's effective processing rate as seen by an object is $r_{node} = R_{node}/L_{node}$. We now break down the time spent in an observation window in terms of inherent application and system parameters as follows:

$$T_{app} = \sum_{j=0}^{n} \frac{h_j b_j}{r_{node}} + \sum_{j=0}^{n} C_j \tag{2}$$

The numerator $h_j b_j$ represents the number of instructions executed by the object during the observation window while the denominator represents the effective processor rate as observed by the object.

In our model, local communication within one machine is instantaneous, i.e., an object does not block when communicating with another object colocated on the same node. This assumption is safe because inter-node communication is so much less expensive than network communication. Assume that under the original placement objects O_1 through O_k execute on the client and objects O_{k+1} through O_n execute on the server. With these refinements, we rewrite T_{app} as follows:

$$T_{app}(k) = \sum_{j=0}^{k} \frac{h_j b_j}{r_{client}} + C_k + \sum_{j=k+1}^{n} \frac{h_j b_j}{r_{server}} \tag{3}$$

This equation expresses application response time as the sum of the time spent computing at the client, the time spent transferring data over the network, and the time spent processing at the server. T_{app} is expressed as a function of k, the point at which the application's stack is split. Note that migrations potentially change the load on participating nodes and the stall time on the network between them.

Recall that in the above equation, r_{server} can be expressed in terms of raw processor rate, R_{server}, and node load, L_{server}, as R_{server}/L_{server}. Substituting R_{server}/L_{server} for r_{server} and R_{client}/L_{client} for r_{client} in Equation 3, we get:

$$T_{app}(k) = L_{client}(k) \sum_{j=0}^{k} \frac{h_j b_j}{R_{client}} + C_k + L_{server}(k) \sum_{j=k+1}^{n} \frac{h_j b_j}{R_{server}} \tag{4}$$

Equation 4 can be used to estimate application response time under a hypothetical placement. It requires inherent parameters, such as instructions per byte and raw processor rates, which can be derived directly from static characteristics or from collected statistics. It also requires calcluating node load, $L_{server}(k)$ and $L_{client}(k)$, whose values under the hypothetical placement must be estimated.

Given k', a hypothetical stack split point, and assuming a single active application stack, the load on the server, $L_{server}(k')$, can be estimated from the current measured load, $L_{server}(k)$ as:

$$L_{server}(k') = L_{server}(k) + \frac{\sum_{j=k'}^{k} h_j b_j}{R_{server} T_{app}(k')} \tag{5}$$

The equation above assumes $k' < k$. A similar equation can be easily derived for the other case. Equation 5 states that the difference in load due to the change in placement is equal to the *fraction of time the objects that are hypothetically migrated to the server will spend processing at the server*. The remainder of the time the objects are passive, waiting for client-side processing or network transfers.

The new value for application response time can be expressed by rewriting Equation (4), substituting k' for k, as:

$$T_{app}(k') = L_{client}(k') \sum_{j=0}^{k'} \frac{h_j b_j}{R_{client}} + C_{k'} + L_{server}(k') \sum_{j=k'+1}^{n} \frac{h_j b_j}{R_{server}} \tag{6}$$

Equations 5 and 6 can be used together to calculate the new value for T_{app} under stack split k'. The only remaining complication is that estimating the new value for T_{app} using Equation 5 requires the new value for server load, L_{server}. But we arrived at an estimate of the new server load that requires the new value for T_{app}. To resolve this circular dependency, an iterative algorithm can be used. The current values are used to initialize T_{app} and L_{server}, then Equation 6 is invoked to compute new T_{app}. The new value for T_{app} can then be used to estimated the new load, and the algorithm repeats until it converges.

5.4.2 Concurrent stacks

In practice, multiple applications can be active at the same time. The server-side resource managers must decide at what point to split *each* stack such that the average T_{app} across all applications is minimal. A migration in one application's stack usually impacts the server load and can result in slow-downs or speed-ups to the remainder of server-resident applications objects.

We formalize the notion of a placement as a vector of stack splits across all active applications. Thus a placement, P, can be denoted as $(k_1, k_2, ..., k_N)$, where N is the number of application stacks actively accessing the server. The response time for application T_{app} with stack split k_1 can be written using equation (4) as:

$$T_{app}(P) = L_{client}(P) \sum_{j=0}^{P} \frac{h_j b_j}{R_{client}} + C_{k_1} + L_{server}(P) \sum_{j=k_1+1}^{n} \frac{h_j b_j}{R_{server}} \tag{7}$$

Equation 7 expresses application response time as a function of the placement P. Let's first consider the implications of Equation 7. Naturally, a placement P that migrates *more* objects to the server results in increasing the node load, $L_{server}(P)$, magnifying the third term in Equation 7 above for each application that performs any processing at the server. If the server were infinitely fast, changes in the load, $L_{server}(P)$, will make little difference to the the third term of Equation 7 relative the second term, or network stall time. Thus, in the case of an infinitely fast server CPU, the placement algorithm should select k (the point at which to split the stack) such that stall time on the network, C_k, is minimized without worrying about changes in the third term. In practice, however, the server's CPU is not infinitely fast and increases in $L_{server}(P)$ can cause server-side processing to be the bottleneck under high load. Under such conditions, migrating more objects to the server becomes counter-productive because the savings in stall time are offset by slower server-side processing.

In the general case, the server-side resource managers have to estimate values for client and server load under a hypothetical change in placement. In particular, client load, L_{client}, is assumed to remain constant. Since the server-side resource managers have no control over the allocation of the client CPU or over what applications can be started, this is a reasonable assumption in practice. The new server load, L_{server}, after a hypothetical change in placement is estimated using Equation 6 above.

5.4.3 Placement algorithm

Now that we can analytically predict the change in average T_{app} given a hypothetical change in object placement, we need to decide how to use this to enact effective placement decisions. In particular, if the server has foreknowledge of all the application that will start, then it can determine the best "future schedule" for allocating its resources to application objects so that average response time is minimized by calculating the new value for T_{app} under each placement and selecting the best placement at each point in time in the future.

In the absence of this knowledge, however, the server has to operate only based on statistics about the past behavior of applications that have already started. ABACUS allocates the server's resources greedily to the currently known applications, and then reconsiders the decision when other applications start. ABACUS makes changes placement if it would result in a reduction in average response time, considering the cost of the migrations needed to move to the next placement. The cost is modeled as the time wasted to acquiescing, checkpointing, transferring state, and restarting a migrated objects. We keep a history of the time needed to quiesce activity in an object graph and we estimate the cost of the transfer from the size of the checkpoint and the measured bandwidth between the source and target node. A relocation is performed only if the net benefit — benefit minus cost — is greater than a threshold value, B_{thresh}. In our prototype, both this threshold and the history window size are configurable. In our experiments, we used a value of 1 second for H and 30% for B_{thresh}.

6 Performance benefits of dynamic placement

In this section, we show how performance depends on the appropriate placement of function. The several sections that follow give increasingly difficult cases where ABACUS can adapt function placement even when the correct location is hard or impossible to anticipate at design-time. We begin with scenarios in which the objects' correct location is based on hardware characteristics, application run-time parameters, application data access patterns, and varying levels of data contention among multiple applications. We conclude with the following sections stressing adaptation under dynamic conditions: phases of application behavior, and contention by multiple applications.

6.1 Evaluation environment

Our evaluation environment consists of eight clients and four storage servers. All twelve nodes are standard PCs running RedHat Linux 5.2 and are equipped with 300 MHz Pentium II processors and 128 MB of main memory. None of our experiments exhibited significant paging activity. Each server contains a single Maxtor 84320D4 IDE disk drive (4 GB, 10 ms average seek, 5200 RPM, up to 14 MB/s media transfer rate). Our environment consists of two networks, a switched 100 Mbps Ethernet, which we refer to as the *SAN* (server-area network) and a shared 10 Mbps segment, which we refer to as the *LAN* (local-area network). All four storage servers are directly connected to the SAN, whereas four of the eight clients, are connected to the SAN (called SAN clients), and the other four clients reside on the LAN (the LAN clients). The LAN is bridged to the SAN via a 10 Mbps link. While these networks are of low performance

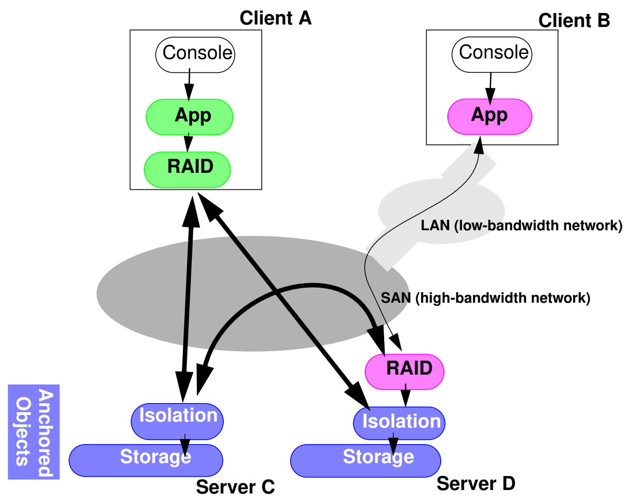

Figure 5: This figure shows a file bound to an application accessing a RAID object which maintains per-file parity code and accesses storage objects running on the storage devices. We show one binding of the stack (Client A) where the RAID object runs at the client, and another binding (Client B) where the RAID object runs at one of the storage devices. Thicker lines represent more data being moved. The appropriate configuration is dependent on the bandwidth available between the client and storage devices. If the client LAN is slow, Client B's partitioning would lead to lower access latencies.

by today's standards, their relative speeds are similar to those seen in high-performance SAN and LAN environments (Gbps in the SAN and 100 Mbps in the LAN.

The bar graphs in the following sections adhere to a common format. Each graph shows the elapsed time of several configurations of an experiment with a migrating object. For each configuration, we report three numbers: the object (1) statically located at the client, (2) beginning at the client, but with ABACUS dynamically monitoring the system and potentially migrating the object, and (3) statically at the storage server. Graphs with confidence intervals report averages over five runs with 90% confidence. We have intentionally chosen smaller benchmarks to underscore ABACUS's ability to adapt quickly. We note that the absolute benefit achieved by dynamic function placement is often a function of the duration of a particular benchmark, and that longer benchmarks operating on larger files would amortize adaptation delays more thoroughly. Throughout the experiments in this section, the observation window, H, was set to 1 second, and the threshold benefit, B_{Thresh}, was set to 30% of the observation window.

6.2 Adapting to network topology/speed

Issue. Network topology/speed dictate the relative importance of cross-network communication relative to server load. Here we evaluate the ability of ABACUS to adapt to different network topologies. We default to executing function at clients to offload contended servers. However, ABACUS moves function to a server if a client would benefit and the server has the requisite available cycles. Our goal was to see whether ABACUS can decide when the benefit of server-side execution due to the reduction in network stall time exceeds the possible slowdown due to slower server-side processing.

Experiment. Software RAID is an example of a function which moves a significant amount of data and performs quite a bit of computation (XOR). Files in the ABACUS filesystem can be bound to a RAID object

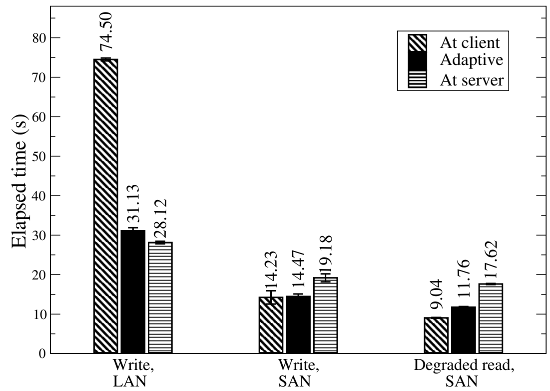

Figure 6: his figure shows the results of our RAID benchmark. Contention on the server's CPU resources make client-based RAID more appropriate, except in the LAN case, where the network is the bottleneck.

providing storage striping and fault-tolerance. The RAID object maintains parity on a per-file basis, stripes data across multiple storage servers, and is distributed to allow concurrent accesses to shared stripes by clients by using a timestamp based concurrency control protocol [AGG00]. The RAID object can execute at either the client nodes or the storage servers. The object graph used by files for this experiment is shown in Figure 5.

The proper placement of the RAID object largely depends on the performance of the network connecting the client to the storage servers. Recall that a RAID small write invokes four I/Os, two to pre-read the old data and parity, and two to write the new data and parity. Similarly, when a disk failure occurs, a block read requires reading all the blocks in a stripe and XORing them together to reconstruct the failed data. This can result in substantial network traffic between the RAID object and the storage servers.

We construct two workloads to evaluate RAID performance on ABACUS. The first consists of two clients writing two separate 4 MB files sequentially. The stripe size is 5 (4 data + parity) and the stripe unit is 32 KB. The second workload consists of the two clients reading the files back in degraded mode (with one disk marked failed).

Results. As shown in Figure 6, executing the RAID object at the server improves RAID small write performance in the LAN case by a factor of 2.6 over executing the object at the host. The performance of the experiment when ABACUS adaptively places the object is within 10% of the optimal. Conversely, in the SAN case, executing the RAID object locally at the client is 1.3X faster because the client is less loaded and able to perform the RAID functionality more quickly. Here, ABACUS arrives within 1% of this value. The advantage of client-based RAID is slightly more pronounced in the more CPU-intensive degraded read case, in which the optimal location is almost twice as fast as at the server. Here, ABACUS arrives within 30% of the optimal. In every instance, ABACUS automatically selects the best location for the RAID object.

6.3 Adapting to run-time parameters

Issue. Applications can exhibit drastically different behavior based on run-time parameters. In this section, we show that the data being accessed by a filter (which is set by an argument) determines the appropriate location for the filter to run. For example, there's a drastic difference between `grep kernel Bible.txt` and `grep kernel LinuxBible.txt`.

Experiment. As data sets in large-scale businesses continue to grow, an increasingly important user

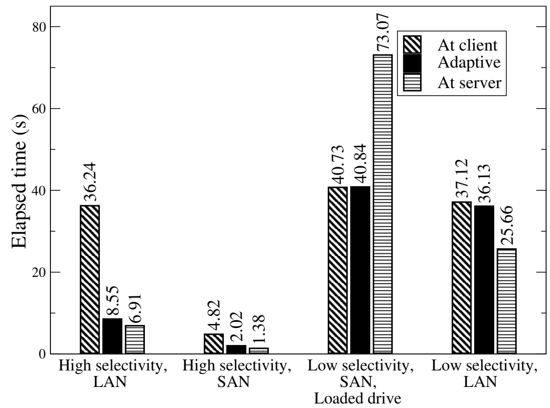

Figure 7: The performance of our filter benchmark is shown in this figure. Executing the filter at the storage server is advantageous in all but the third configuration, in which the filter is computationally expensive and runs faster on the more resourceful client.

application is high-performance search, or data filtering. Filtering is generally a highly selective operation, consuming a large amount of data and producing a smaller fraction. We constructed a synthetic filter object that returns a configurable percentage of the input data to the object above it. Highly selective filters represent ideal candidate for execution close to the data, so long as storage resources are available.

In this experiment, we varied both the filter's selectivity and CPU consumption from low to high. We define selectivity as: selectivity $= (1 - \text{output}/\text{input})$. A filter labeled low selectivity outputs 80% of the data that it reads, while a filter with high selectivity outputs only 20% of its input data. A filter with low CPU consumption does the minimal amount of work to achieve this function, while a filter with high CPU consumption simulates traversing large data structures (e.g., the finite state machines of a text search program like `grep`).

Results. Figure 7 shows the elapsed time to read and filter a 16 MB file in a number of configurations. In the first set of numbers, ABACUS migrates the filter from the client to the storage server, coming within 25% of the ideal case, which is over 5X better than filtering at the client. Similarly, ABACUS migrates the filter in the second set. While achieving better performance than statically locating the filter at the client, ABACUS reaches only within 50% of the optimal because the time required for ABACUS to migrate the object is a bigger fraction of total runtime. In the third set, we run a computationally expensive filter. We simulate a loaded or slower storage server by making the filter twice as expensive to run on the storage server. Here, the filter executes 1.8X faster on the client. ABACUS correctly detects this case and keeps the filter on the client. Finally, in the fourth set of numbers, value of moving is too low for ABACUS to deem it worthy of migration. Recall that the migration threshold is 30%.

6.4 Adapting to data access patterns

Issue. Client-side caches in distributed file and database systems often yield dramatic reduction in storage access latencies because they avoid slow client networks, increase the total amount of memory available for caching, and reduce the load on the server. However, enabling client-side caching can yield the opposite effect under certain access patterns. In this section, we show that ABACUS appropriately migrates the per-file cache object in response to data access patterns via black-box monitoring.

Experiment. Caching in ABACUS is provided by a mobile cache object. Consider an application that

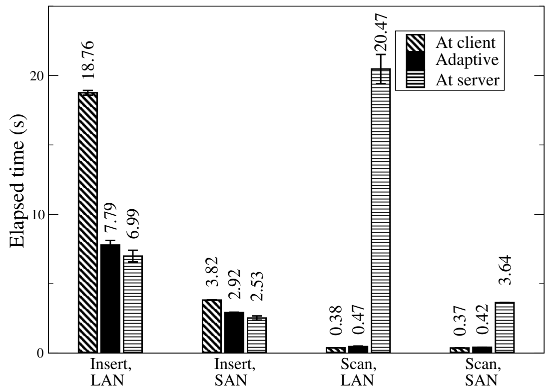

Figure 8: The figure shows that client-side caching is essential for workloads exhibiting reuse (Scan), but causes pathological performance when inserting small records (Insert). ABACUS automatically enables and disables the client caching by placing the cache object at the client or at the server.

inserts small records into files stored on a storage server. These inserts require a read of the block from the server (an *installation read*) and then a write back of entire block. Even when the original block is cached, writing a small record in a block requires transferring the entire contents of each block to the server. Now, consider an application reading cached data. Here, we desire the cache to reside on the client.

We carried out the following experiments to evaluate the impact of and ABACUS's response to application access patterns. In the first benchmark, *table insert*, the application inserts 1,500 128 byte records into a 192 KB file. An insert writes a record to a random location in the file. In the second benchmark, *table scan*, the application reads the 1,500 records back, again in random order. The cache, which uses a block size of 8 KB, is large enough for the working set of the application. Before recording numbers, the experiment was run to warm the cache.

Results. As shown in Figure 8, locating the cache at the server for the insert benchmark is 2.7X faster than at a client on the LAN, and 1.5X faster than at a client on the SAN. ABACUS comes within 10% of optimal for the LAN case, and within 15% for the SAN case. The difference is due to the relative length of the experiments, causing the cache to migrate relatively late in the SAN case (which runs for only a few multiples of the observation window). The table scan benchmark highlights the benefit of client-side caching when the application workload exhibits reuse. In this case, ABACUS leaves the cache at the client, cutting execution time over caching at the server by over 40X and 8X for the LAN and SAN tests respectively.

6.5 Adapting to contention over shared data

Issue. Filesystem functionality such as caching or namespace updates/lookups is often distributed to improve scalability [HKM+88], as are functions of other large applications. When contention for the shared objects between clients is low, executing objects at the client(s) accessing them yields higher scalability and better cache locality. When contention increases over a shared object, however, server-based execution is more preferrable. In this case, client invocations are serialized locally on the server, avoiding the overhead of retries over the network.

This kind of adaptation also solves performance cliffs caused by false sharing in distributed file caches. When several clients are writing to ranges in a file that happen to share common blocks, the invalidation traffic can degrade performance so that write-through to the server would be preferable.

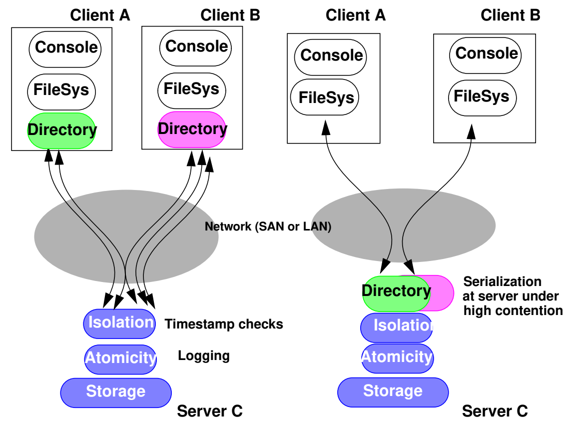

Figure 9: This figure shows directory updates from multiple contending clients. While distributing directory management to clients is beneficial under low contention, under high contention, it results in a flurry of retries per directory operation. When the object is moved to the storage device, multiple client requests are serviced by (multiple threads in) the same object, serializing them locally without the cost of multiple cross-network retries. Adaptation in this case automatically shifts the concurrency control algorithm from a distributed optimistic scheme to a local locking scheme.

	LAN	SAN
At client	125.50	86.10
Adaptive	6.66	27.33
At server	0.00	0.00

Table 1: Shown are the number of retries (averaged over 5 runs) incurred by the optimistic concurrency control scheme when inserting entries into a highly contended directory. We show the directory object statically placed at the client, dynamically located by ABACUS, and statically placed at the storage server.

Experiment. We chose a workload that performs directory inserts in a shared namespace as our contention benchmark. This benchmark is even more complicated than the distributed file caching case and therefore more challenging to ABACUS. Directories in ABACUS present a hierarchical namespace like all UNIX filesystems and are implemented using the object graph shown in Figure 9. When clients access disjoint parts of the directory namespace (*i.e.,* there are no concurrent conflicting accesses), the optimistic scheme in which concurrency control checks are performed by the isolation object (recall Section 4.3.1) works well. Each directory object at a client maintains a cache of the directories accessed frequently by that client, making directory reads fast. Moreover, directory updates are minimally cheap because no metadata pre-reads are required, and no lock messaging is performed. Further, offloading the server for the bulk of the work results in better scalability and frees storage devices to execute demanding workloads from competing clients. When contention is high, however, the number of retries and cache invalidations seen by the directory object increases, potentially causing several round-trip latencies per operation. When contention increases, we desire the directory object to migrate to the storage device. This would serialize client updates through one object, thereby eliminating retries.

We constructed two benchmarks to evaluate how ABACUS responds to different levels of directory contention. The first is a high contention workload, where four clients insert 200 files each in a shared

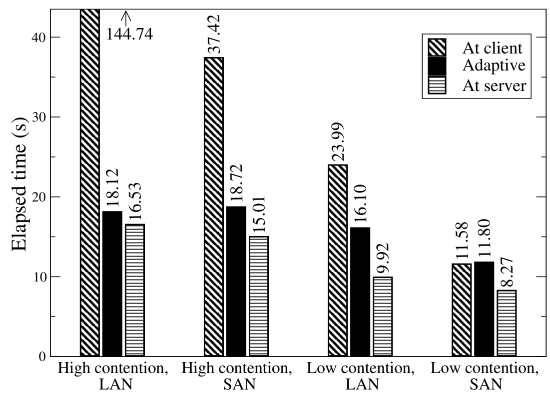

Figure 10: This figure shows the time to execute our directory insert benchmark under different levels of directory contention. ABACUS migrates the directory object in all but the fourth case.

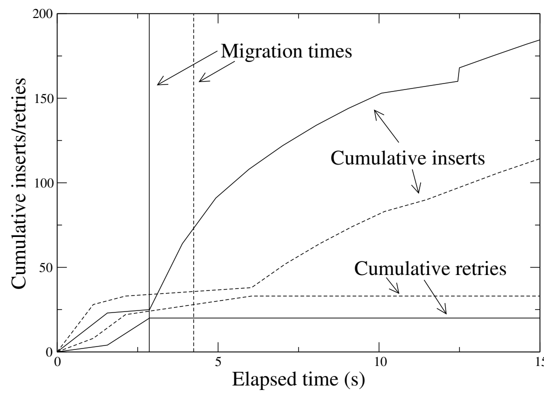

Figure 11: This figure shows the cumulative inserts and retries of two clients operating on a highly contended directory over the SAN. Client 1's curves are solid, while client 2's are dotted.

directory. The second is a low contention workload where four clients insert 200 files each in private (unique) directories.

Results. As shown in Figure 10, ABACUS cuts execution time for the high contention workload by migrating the directory object to the server. In the LAN case, we are within 10% of the optimal. The optimal is 8X better than locating the directory object at the host. We come within 25% of the optimal for the high contention, SAN case (which is 2.5X better than the worst case). We look closely at the retry behavior in Table 1. There are lower retries under ABACUS for the high contention, LAN case than for the high contention, SAN configuration. In both cases, ABACUS observed relatively high traffic between the directory object and storage. ABACUS estimates that moving it closer to the isolation object would make retries cheaper (local to the storage server). It adapts more quickly in the LAN case because the estimated benefit is greater. ABACUS had to observe far more retries and revalidation traffic on the SAN case before deciding to migrate the object.

Under low contention, ABACUS makes different decisions in the LAN and SAN cases, migrating the directory object to the server in the former and not migrating it in the latter. We started the benchmark from

	Insert	Scan	Insert	Scan	Total
At client	26.03	0.41	28.33	0.39	55.16
Adaptive	11.69	7.22	12.15	3.46	34.52
At server	7.76	29.20	7.74	26.03	70.73
Optimal	7.76	0.41	7.74	0.39	16.30

Table 2: Inter-application phases. This table shows the performance of a multiphasic application in the static cases and under ABACUS. The application goes through an insert phase, followed by a scan phase, back to inserting, and concluding with another scan. The table shows the completion time in seconds of each phase under each scheme.

a cold cache, causing many installation reads. Hence, in the low contention, LAN case, ABACUS estimates migrating the directory object to the storage server, avoiding the network, is worth it. However, in the SAN case, the network is fast enough that ABACUS cost-benefit model estimates the installation read network cost to be limited. Indeed, the results show that the static client and storage server configurations for the SAN case differ by less than 30%, our migration threshold.

Note also that the directory objects from different clients need not all migrate to the server at the same time. The server can decide to migrate them independently, based on its estimates of the migration benefit for each client. Correctness is ensured even if only *some* objects migrate, because all operations are verified to have occurred in timestamp order by the underlying isolation/atomicity object. Figure 11 shows a timeline of two clients from the high contention, SAN benchmark. The graph shows the cumulative number of inserted files and the cumulative number of retries for two clients. One client experiences a sharp increase in retries and its object is moved to the server first. The second happens to suffer from a relatively low, but steady retry rate, which triggers its adaptation a little later. The first client experiences a sharp increase in the rate of progress soon after it migrates. The second experiences a substantial, but *lower*, increase in its rate of progress after it migrates, which is expected as storage server load increases.

6.6 Adapting to application phases

Issue. Having established that optimal placement depends on several system and workload characteristics, we further note that these characteristics change with time on most systems. In this section, we are concerned with characteristics that vary with algorithmic changes within the lifetime of the application. The next section addresses adaptation in light of competition from concurrent applications.

Applications rarely exhibit the same behavior or consume resources at the same rate throughout their lifetimes. Instead, an application may change phases at a number of points during its execution in response to input from a user or a file or as a result of algorithmic properties. Multiphasic applications make a particularly compelling case for the function relocation that ABACUS provides.

Experiment. We now revisit our file caching example with the difference that it is now multiphasic. We run a benchmark which does an insert phase, followed by scanning, followed by inserting, and concluding with another scan phase. The goal is to determine whether the benefit estimates at the server will eject an application that changed its behavior after being moved to the server. Further, we wish to see whether ABACUS recovers from bad history quickly enough to achieve adaptation that is useful to an application that exhibits multiple contrasting phases.

Results. Table 2 shows that ABACUS migrates the cache to the appropriate location based on the behavior of the application over time. First, ABACUS migrates the cache to the server for the insert phase. Then, ABACUS ejects the cache object from the server server when the server detects that the cache is being reused by the client. Both static choices lead to bad performance in alternating phases. Consequently,

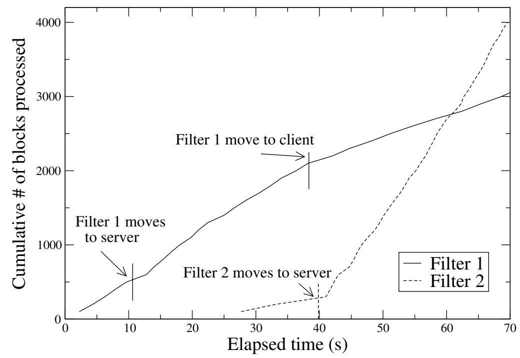

Figure 12: This figure plots the cumulative number of blocks searched by two filters versus elapsed time. ABACUS's competition resolving algorithm successfully chooses the more selective Filter 2 over the Filter 1 for execution at the storage server.

ABACUS outperforms both static cases — by 1.6X compared to the client case, and by 2X compared to the server case. The optimal row refers to the minimum execution time picked alternatively from the client and server cases. We see that ABACUS is approximately twice as slow as the optimal. This is to be expected, as this extreme scenario changes phases fairly rapidly.

6.7 Adapting to competition over resources

Issue. Shared storage server resources are rarely dedicated to serving one workload. An additional complexity addressed by ABACUS is provisioning storage server resources to competing clients. Toward reducing global application execution time, ABACUS resolves competition among objects that would execute more quickly at the server by favoring those objects that would derive a greater benefit from doing so.

Experiment. In this experiment, we run two filter objects on a 32 MB file on our LAN. The filters have different selectivities, and hence derive different benefits from executing at the storage server. In detail, Filter 1 produces 60% of the data that it consumes, while Filter 2, being the more selective filter, outputs only 30% of the data it consumes. The storage server's memory resources are restricted so that it can only support one filter at a time.

Results. Figure 12 shows the cumulative progress of the filters over their execution, and the migration decisions made by ABACUS. The less selective Filter 1 is started first. ABACUS shortly migrates it to the storage server. Soon after, we start the more selective Filter 2. Shortly after we start the second filter, ABACUS migrates the highly selective Filter 2 to the server, kicking back the other to its original node. The slopes of the curves show that the filter currently on the server runs faster than when not, but that Filter 2 derives more benefit since it is more selective. Filters are migrated to the server after a noticeable delay because in our current implementation, clients do not periodically update the server with resource statistics.

7 Conclusions

In this paper, we demonstrate that optimal function placement depends on system and workload characteristics that are impossible to predict at application design or installation time. We propose a dynamic approach to function placement in clusters where a distributed runtime system continuously adapts placement decisions based on resource usage and availability. Measurements demonstrate that placement can be decided based on black-box monitoring of application objects, in which the system is oblivious to the function being implemented. Preliminary evaluation shows that ABACUS, our prototype system, can improve application response time by 2–10X. In all of our experiments, ABACUS selects the best placement for each function, "correcting" placement when function was initially started on the "wrong" node. Further, ABACUS outperforms all one-time placements in situations where dynamic changes cause the proper placement to vary during an application's execution. In summary, our results to date indicate a promising future for this approach. We believe that the ability to continually and automatically reconfigure storage clusters will eventually help reduce their exorbitant management costs.

References

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999.

[AGG00] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *Proceedings of the 20th International Conference on Distributed Computing Systems (to appear)*, April 2000.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, CA, October 1998.

[BG80] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th Conference on Very Large Databases*, October 1980.

[BLL91] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, October 1991.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[DO91] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice & Experience*, 21(8):757–785, August 1991.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels meet recursive virtual machines. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 137–151, October 1996.

[FJK96] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 149–160, New York, June 4–6 1996.

[GNI+99] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. In *Proceedings of the USENIX '99 Extreme Linux Workshop*, June 1999.

[HF75] Griffith Hamlin, Jr. and James D. Foley. Configurable applications for graphics employing satellites. In *Computer Graphics (SIGGRAPH '75 Proceedings)*, pages 9–19, June 1975.

[HF93] Eric H. Herrin, II and Raphael A. Finkel. Service rebalancing. Technical Report CS-235-93, Department of Computer Science, University of Kentucky, May 18 1993.

[HKM+88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[HP94] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[HS99] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, January 1988.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, March 1997. http://www.pdos.lcs.mit.edu/rover/.

[KN93] Yousef Khalidi and Michael Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[KPH98] K. Keeton, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(3), 1998.

[KS97] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, August 1997.

[MvD76] Janet Michel and Andries van Dam. Experience with distributed processing on a host/satellite graphics system. In *Computer Graphics (SIGGRAPH '76 Proceedings)*, pages 190–195, July 1976.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., October 1996.

[RGF98] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*. Very Large Data Base Endowment, August 1998.

[Sea99] Seagate. Jini: A Pathway for Intelligent Network Storage, 1999. Press release, http://www.seagate.com/-corp/vpr/literature/papers/jini.shtml.

[Tho97] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[WLA93] Robert Wahbe, Steven Lucco, and Thomas Anderson. Efficent software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.