To appear in the USENIX Annual Technical Conference, San Diego, CA, June 2000.

Dynamic Function Placement for Data-intensive Cluster Computing

Khalil Amiri, David Petrou, Gregory R. Ganger, Garth A. Gibson
Carnegie Mellon University
{amiri+,dpetrou,ganger,garth}@cs.cmu.edu
http://www.pdl.cs.cmu.edu/

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.*

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 cannot be determined until application invocation time, because they depend on input parameters. Worst of all, many change at run-time due to an application changing phases 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-

This research is supported 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 members of the PDC, including EMC, Hewlett-Packard, Hitachi, Intel, LSI Logic, Novell, Quantum, Seagate Technology, and 3COM.

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 [25], much-evolved commodity active disks [1, 19, 24] or file servers allowing remote execution of applications, 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 move and process 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. In our prototype system, called ABACUS, functions associated with particular data streams are moved back and forth between clients and servers in response to dynamic 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 can outperform 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. Section 4 describes the ABACUS programming model and several example applications built upon it. Section 5 describes the run-time system. Section 6 presents a variety of experiments to demonstrate the value of dynamic function placement and ABACUS's ability to effectively 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. This section discusses the most relevant previous work on adaptive function placement and how it relates to ABACUS.

Several previous systems such as Coign and others [16, 22] 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 [4] 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 presence of multiple *different* tasks.

Equanimity is a system that, like ABACUS, dynamically balances service between a single client and its server [14]. 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.

Hybrid shipping [10] is a technique proposed to dynamically distribute query processing load between clients and servers of a database management system. This technique uses *a priori* knowledge of the algorithms implemented by the query operators to estimate the best partitioning of work between clients and servers. Instead, ABACUS applies to a wider class of applications by relying only on black-box monitoring to make placement decisions, without knowledge of the semantics or algorithms implemented by the application components.

Process migration systems such as Condor [7] and Sprite [8] 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 systems such as Emerald [18] and Rover [17] do support fine-grain mobility of application objects, but they leave migration decisions to the application programmer. Similarly, mobile agent systems, such as Mole [26] and Agent Tcl [12], enable agent migration but do not provide algorithms or mechanisms to decide where agents should be placed. ABACUS builds on such work by providing run-time 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 several test applications to it. 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.

The ABACUS programming model encourages the programmer to compose data-intensive applications from explicitly-migratable, functionally independent components or objects. These *mobile* objects provide explicit methods that *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 that 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 that 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, or *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. Also, each machine's binding manager notifies the local resource manager of each procedure call to and return from a mobile object.

The resource manager uses the notifications to collect statistics about bytes moved between objects and about the resources used by the objects (*e.g.*, amount of memory allocated, number of instructions executed per byte processed). A resource manager also monitors the load on its local processor and the experienced stall time on network transfers to and from the storage servers that are actively

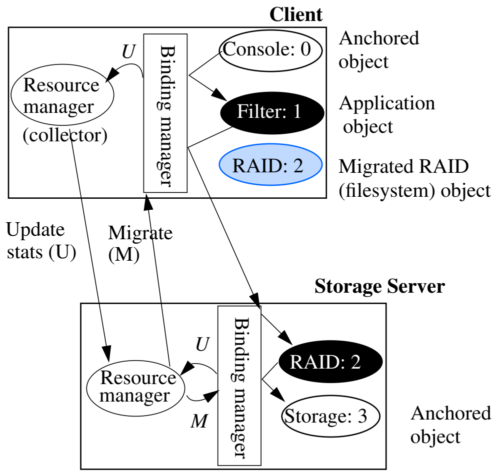

Figure 1: An illustration of an ABACUS object graph, the principal ABACUS components, and their interactions. This example shows a filter application accessing a striped file. Functionality is partitioned into objects. Dark ovals depict mobile objects, while clear ovals mark anchored objects. Inter-object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

accessed by local mobile objects. Server-side resource managers collect statistics from client-side resource managers and employ an analytic model to predict the performance benefit of moving to an alternative placement. The model also takes into account the cost of migrations, including the time wasted waiting until the object is quiescent and the time wasted for checkpointing the object, transferring its state, and restoring it on the target node. Using this analytic model, the server-side resource manager arrives at the placement with the best *net benefit*. If this placement is different from the current configuration, the necessary object migrations are effected.

The ABACUS prototype is written in C++. We leverage the language's object-oriented features to simplify writing our mobile objects. For our inter-node communication transport, we use DCE RPC. Because the focus of our prototype is on function placement decisions, we do not address several important but orthogonal mobile code issues. For example, we sidestep the issues of code mobility [27] and dynamic linking [6] by requiring that all migratable modules be statically linked into an ABACUS process on both clients and servers. Further, issues of inter-module protection [28, 23, 9] are not addressed. While these will be important issues for production systems, they are tangential to the questions addressed in this paper.

4 Programming model

The ABACUS programming model has two principal aspects: *mobile objects*, which represent the unit of migration and placement, and an *iterative processing model*, which defines how mobile objects are composed into entire data processing applications.

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 instances of it and migrate it. Mobile objects are usually of medium 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, or aggregation, for example.

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. A 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. Both methods are invoked when there is no external invocation active within the mobile object.

Each storage server (i.e., 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).

4.2 Iterative processing model

Synchronous invocations start at the top-level console object and propagate down the object graph. Each invocation returns back to the console object with a result after a specified number of application records have been processed. Once an invocation returns to the console, objects in the graph are usually no longer active. Objects are activated again by the next *iteration*, which starts with a new invocation initiated by the console. Sometimes, objects may become "spontaneously" active, initiating invocations before any request is received from top-level objects. This occurs when objects perform background work (such as write-behind in a cache object), although that is not assumed to be the common mode of operation.

The amount of data moved in each invocation is an application-specific number of records, and not the entire file or data set at once. This iterative property is required by 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.

4.3 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.

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 (*e.g.*, RAID), caching, and application-specific functionality. This approach was pioneered by the stackable and composable filesystem work [13, 21] and by the Spring object-oriented operating system [20].

The ABACUS filesystem provides coherent file and directory abstractions atop the flat file 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 [15]. 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 absence of contention because it does not require a lock server or any lock traffic. In Section 6.5, we describe how ABACUS automatically 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

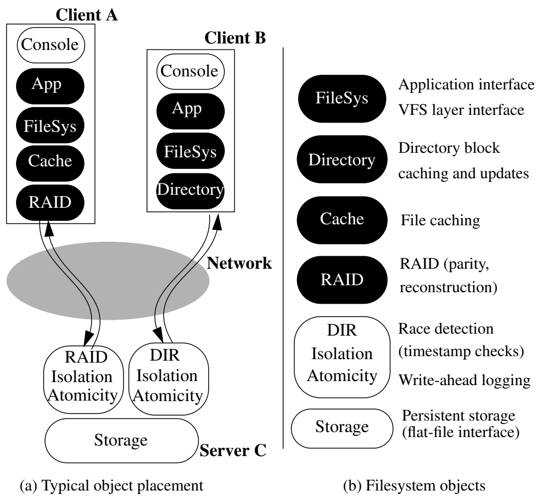

Figure 2: The architecture of the object-based distributed filesystem built atop ABACUS. The figure shows a typical file and directory object stack (a). The object placement shown is the default for high-bandwidth networks and trusted clients. Also shown are the component filesystem objects which are implemented to date and a brief description of their function (b).

applications, a cache object, an optional RAID 5 object, and one or more storage objects. To ensure that parity is not corrupted by races involving concurrent writes to the same stripe, a RAID isolation/atomicity object is anchored to each storage server. This object intercepts all reads and writes to the base storage object and verifies the consistency of updates before committing them. The protocols used by this isolation object are highly scalable and are described elsewhere [2]. 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 CMU's Network-Attached Secure Disks (NASD) prototype [11] as backing store. Figure 2 shows a sketch of typical file and directory stacks.

Each directory's graph consists of a directory object, an isolation/atomicity object and a storage object. The directory object provides POSIX-like directory calls and caches directory entries. The isolation/atomicity object provides support for both cache coherence and optimistic concurrency control, and also ensures the atomicity of multi-block writes to a directory. For performance reasons, the implementation of the isolation/atomicity object is specialized to directory semantics and is therefore different from the RAID 5 isolation/atomicity object described above. This object ensures cache coherence by interposing on read and write calls, installing callbacks on cached blocks during read calls and breaking relevant callbacks during writes. It ensures proper concurrency control among simultaneous updates by timestamping cache blocks [5] and exporting a special `CommitAction()` method that checks specified *readSets* and *writeSets* for conflicts.[1] Finally, the atomicity of multi-block writes is provided by ensuring that a set of blocks (possibly from differing objects and devices) are either updated in their entirety or not updated at all, by using a write-ahead log that is shared among all of the instances of the isolation/atomicity object.

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.

Object-based applications. Data-intensive applications can be similarly decomposed into objects that perform operations such as search, aggregation, or data mining. Porting a data-intensive application, such as search, to ABACUS is straightforward. Most search applications already iterate over input data by invoking successive `read` calls to the filesystem and operating on a buffer at a time. Porting this kind of application simply requires encapsulating the filtering component of the search into a

[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 exist then 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.

C++ object and writing `checkpoint/restore` methods for it. These methods are also relatively straightforward, since the state often consists of just the positions in the input and output files, and the contents of the current buffer.

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 for the migration of objects between cluster nodes (Section 5.2). The resource managers collect statistics about resource usage and availability (Section 5.3) and use these measurements to adapt placement decisions to improve total application response time (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 that execute applications that access storage servers. 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 the `new` operator of the C++ run-time.

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 `rid`s 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_reference_within_node)* pair. As mobile objects move between nodes, this table is updated to reflect the new node and the new object reference at that node. 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. 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.

[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.

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 of 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 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, extra messaging is not usually required to update location tables during migration.

5.3 Resource monitoring

The run-time system uses its intermediary role in redirecting calls to collect all the necessary statistics. By only interposing monitoring code at procedure call and return from mobile objects, ABACUS does not slow down the execution of methods within a mobile object. This section explains how the needed statistics are collected.

On the same node, threads can cross the boundaries of multiple mobile objects by making method invocations that propagate down the stack. The 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 observation window. We describe how some of these statistics are collected:

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. These 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. ABACUS monitors 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. Given the processing rate on a node, this amounts to measuring the time spent computing within an object. We use a combination of the Linux interval timers and the Pentium cycle counter to keep track of the time spent processing within a mobile object.

Stall time. To estimate the amount of time a thread spends stalled in an object, one needs more information than is currently provided by the POSIX 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 of any threads in the queue that are marked as blocked (`TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).

5.4 Dynamic Placement

The resource manager on a given server seeks to perform the migrations that will 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 most 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's processor is overloaded.

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. The analytic model considers alternative placement configurations and selects the one with the best *net benefit*, which is the difference between the benefit of moving to that placement and the cost of migrating to it. This net benefit represents the estimated reduction in execution time over the next H seconds.

A migration is actually enacted only if the server-side resource manager finds a new placement whose associated net benefit exceeds a configurable threshold, B_{Thresh}. This threshold value is used to avoid migrations that chase small improvements, and it can be set to reflect the confidence in the measurements and the models used by the run-time system. Server-side resource managers do not communicate with one another to figure out the globally optimal placement. A server-side resource manager decides on the best alternative placement considering only the application streams that access it. This design decision was taken for robustness and scalability reasons.

The details of the computation required to estimate the net benefit are discussed in an associated technical report [3]. Here, we outline the intuition behind the computation. The server-side resource manager receives the per-object measurements described above. It also receives statistics about the client processor speed and current load and collects similar measurements about the local system and locally executing objects. Given the data flow graph between objects, the measured stall time of client-side objects, and the latency of the client-server link, the model estimates the change in stall time if an object changes location. Given the instructions per byte and the relative load and speed of the client/server processors, it estimates the change in execution time if the object changes placement. In addition to the change in execution time for the migrated object, the model also estimates the change in execution time for the other objects executing at the target node (as a result of the increased load on the node's processor). When considering different object placements, we treat the memory available at the server as a fixed constraint. Together, the changes in stall time and execution time amount to the benefit of the new placement. In computing this benefit, our analytic model assumes that history will repeat itself over the next window of observation (the next H seconds). The cost associated with a placement is estimated as the sum of a fixed cost (the time taken to wait until the object is quiescent) plus the time to transfer the object's state between source and destination nodes. This latter value is estimated from the size of the checkpoint buffer and the bandwidth between the nodes.

6 Performance evaluation

In this section, we show how performance depends on the appropriate placement of function. The subsections 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. This includes scenarios in which the objects' correct location is based on hardware characteristics, application run-time parameters, application data access patterns, and inter-application contention over shared data. This also includes scenarios that stress adaptation under dynamic conditions: phases of application behavior and contention by multiple applications. We could not perform a fair comparison of applications running on ABACUS to those running on a network filesystem, such as NFS, because our filesystem implementation differs from that of Linux's NFS. The differences give ABACUS applications advantages that have little to do with adaptive function placement.

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 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. The goal is 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 that moves a significant amount of data and often touches every byte (computes the bitwise XOR of the contents of multiple blocks). Files in the ABACUS filesystem can be bound to a RAID object that provides 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 [2]. 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 3.

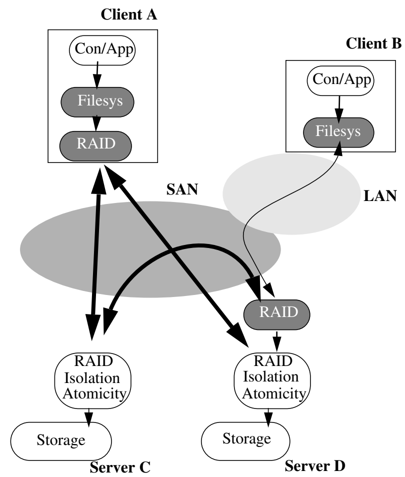

Figure 3: This figure shows a file bound to an application accessing a RAID object which maintains per-file parity code and accesses storage objects running on the storage devices. We show one binding of the stack (Client A) where the RAID object runs at the client, and another binding (Client B) where 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.

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 involves 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

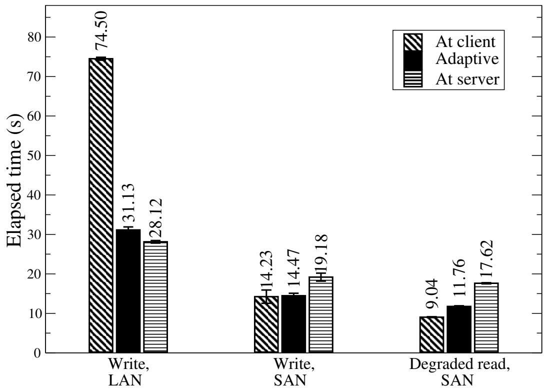

Figure 4: This figure shows the results of our RAID benchmark. Contention on the server's CPU resources make client-based RAID more appropriate, except in the LAN case, where the network is the bottleneck.

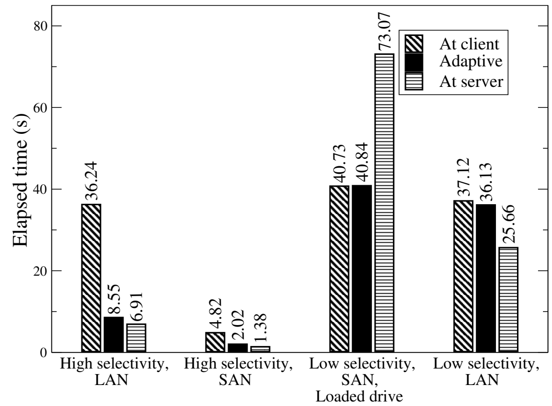

Figure 5: The performance of our filter benchmark is shown in this figure. Executing the filter at the storage server is advantageous in all but the third configuration, in which the filter is computationally expensive and runs faster on the client, has more CPU resources available.

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 4, 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 optimal. Conversely, in the SAN case, executing the RAID object locally at the client is 1.3X faster because the client has a lower load and is 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 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 application is high-performance search, or data filtering. Filtering is often 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 $(1 - \text{output}/\text{input})$. 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 5 shows the elapsed time

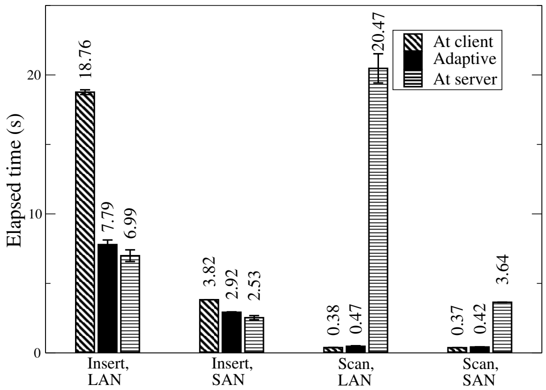

Figure 6: The figure shows that client-side caching is essential for workloads exhibiting reuse (Scan), but causes pathological performance when inserting small records (Insert). ABACUS automatically enables and disables the client caching by placing the cache object at the client or at the server.

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 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, the value of moving is too low for ABACUS to deem it worthy of migration. Recall that the migration threshold is 30%, and note that this applies to the *estimated* benefit computed by ABACUS and not the real benefit.

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 subsection, 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 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 the 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 6, 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 [15]. 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 over a shared object increases, a server-based execution becomes more efficient. 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.

Experiment. We chose a workload that performs directory inserts in a shared namespace as our contention benchmark. Directories in ABACUS present a hierarchical namespace like all UNIX filesystems and are implemented using the object graph shown in Figure 7.

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) 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 bulk of the work from the server 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 directory. The second is a low contention workload where four clients insert 200 files each in private (unique)

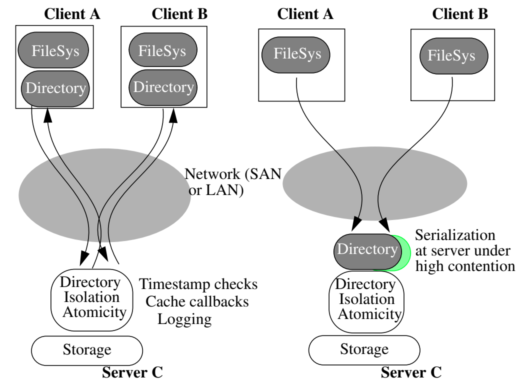

Figure 7: This figure shows directory updates from multiple contending clients. While distributing directory management to clients is beneficial under low contention, under high contention it results in a flurry of retries per directory operation. When the object is moved to the storage device, multiple client requests are serviced by (multiple threads in) the same object, serializing them locally without the cost of multiple cross-network retries.

directories.

Results. As shown in Figure 8, ABACUS reduces execution time for the high contention workload by migrating the directory object to the server. In the LAN case, ABACUS is within 10% of the optimal. The optimal is 8X better than locating the directory object at the host. ABACUS comes within 25% of optimal for the high contention, SAN case (which is 2.5X better than the worst case). ABACUS estimates that moving it closer to the isolation object would make retries cheaper. 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 a cold cache, causing many installation reads. Hence, in the case where there is little contention and the application is running over the LAN, ABACUS estimates that migrating the directory object to the storage server is worthwhile, because it avoids the latency of the low-speed LAN. However, in the SAN case, the network is fast enough that ABACUS cost-benefit model

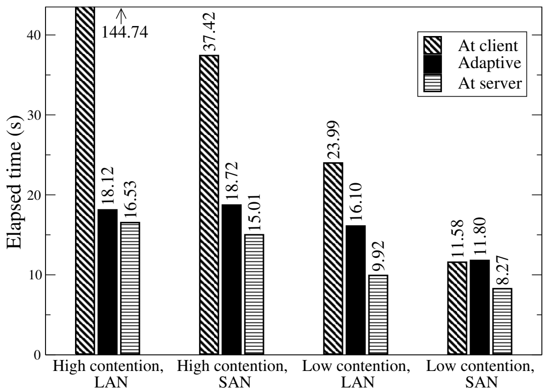

Figure 8: This figure shows the time to execute our directory insert benchmark under different levels of directory contention. ABACUS migrates the directory object in all but the fourth case.

	Insert	Scan	Insert	Scan	Tot.
Client	26.0	0.4	28.3	0.4	55.2
Adapt.	11.7	7.2	12.1	3.5	34.5
Server	7.8	29.2	7.7	26.0	70.7
Opt.	7.8	0.4	7.7	0.4	16.3

Table 1: 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.

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. This benchmark does not exhibit a case where client-side placement is better because our client population is of limited size (only four nodes). 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.

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 subsection, we are concerned with characteristics that vary with algorithmic changes within the lifetime of the application. 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 dynamic function relocation that ABACUS provides.

Experiment. To explore multiphasic behavior, we revisit our file caching example. Specifically, we run a benchmark that 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 1 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 when it detects that the cache is being reused by the client. Both static choices lead to bad performance in alternating phases. Consequently, 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

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 9 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 thereafter, 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 the estimated benefit was close to the configured threshold. Longer history windows will amortize the migration cost over a longer window of benefit, resulting in migration occurring sooner. In general, the history window should be at least long enough to capture many iterations up and down the object stack, so that the statistics collected by ABACUS are representative of application behavior.

ABACUS does place a run-time overhead compared to traditional implementations. A filter implemented and running on ABACUS runs up to 25% slower than one implemented directly atop of the Unix Filesystem, in the case where no migrations occur. [3] Furthermore, ABACUS can, under pathological conditions, result in worse performance than either static placement. Consider the case of an application that completely changes behavior right after ABACUS decides to migrate it, inducing a migration back to the original node, only to change its behavior again and start another cycle. In this case, the application will always be placed on the "wrong" node and will incur additional migration costs that are linear with the migration frequency, which is about once every history window. This worst case behavior is currently bounded by noticing objects that rapidly ping-pong back and forth between locations and anchoring them in one default placement until the application terminates.

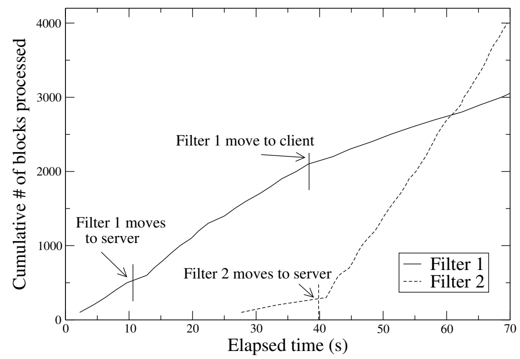

Figure 9: This figure plots the cumulative number of blocks searched by two filters versus elapsed time. ABACUS's competition resolving algorithm successfully chooses the more selective Filter 2 over the Filter 1 for execution at the storage server.

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 where function placement is continuously adapted by a run-time system 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. 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

[3] We believe that part of this overehead can be eliminated with a more optimized implementation.

future for this approach.

Acknowledgements. We would like to thank John Wilkes, Richard Golding, David Nagle, our shepherd Chris Small, and our anonymous reviewers for their valuable feedback.

References

[1] A. Acharya, M. Uysal, and J. 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, Oct. 1998.

[2] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, Apr. 2000.

[3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement in active storage clusters. Technical Report CMU–CS–99–140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1999.

[4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. 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.

[5] 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*, Oct. 1980.

[6] 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, Dec. 1995.

[7] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, Oct. 1991.

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

[9] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. 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, Oct. 1996.

[10] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 149–160, New York, June 4–6 1996.

[11] G. A. Gibson, D. F. Nagle, W. Courtright, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD scalable storage systems. In *Proceedings of the USENIX '99 Extreme Linux Workshop*, June 1999.

[12] R. S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.

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

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

[15] 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), Feb. 1988.

[16] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[17] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Mar. 1997. http://www.pdos.lcs.mit.edu/rover/.

[18] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Jan. 1988.

[19] 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.

[20] Y. Khalidi and M. Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, Dec. 1993.

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

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

[23] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., Oct. 1996.

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

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

[26] M. Straer, J. Baumann, and F. Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996.

[27] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept. 1997.

[28] R. Wahbe, S. Lucco, and T. Anderson. Efficent software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Dec. 1993.