The Effect of Mobile Code on File Service

Tammo Spalink and John H. Hartman
Department of Computer Science
The University of Arizona
{tammo,jhh}@cs.arizona.edu

Garth Gibson
School of Computer Science
Carnegie Mellon University
garth+@cs.cmu.edu

Abstract

Mobile code promises to improve the functionality and performance of applications, but may have a detrimental effect on overall system performance. In this paper we consider the effect of moving an application from a client to a file server, both on the application and the server. Under what circumstances does application performance improve, and does it come at the expense of other (non-mobile) background applications using the same server? We use a trace-driven simulation to measure the effect of mobile code, allowing system parameters such as the size of the server memory and server speed relative to client speed to be varied. We found that several factors influence the benefit of mobile code. Server memory does not appear to be a significant problem; relatively small server caches have a high hit rate even when shared with mobile code. The relative CPU performance of the client and server has a bigger effect on system performance: mobile code should not be run on the server if its CPU is a bottleneck.

1 Introduction

Mobile code is location-independent: it moves dynamically from one machine to another, taking advantage of the different resources available on different machines. Mobile code allows an interactive application to move to the user, improving interactive response. A data processing application can move to its data, avoiding the overhead of transferring the data over the network. The goal is to improve application performance; mobile code does this by allowing the application to move its functionality dynamically between machines. Java [Microsystems95] is a popular example of mobile code; its bytecode is architecture-independent, and its API is operating system-independent, allowing a Java application to run on any platform that supports Java.

The assumption underlying mobile code is that the grass is greener on the other side of the fence: namely, that performance will improve if the application moves to a new location. This isn't always the case, however. Circumstances may conspire against a mobile application: resources may be inadequate at the new location, due to performance or scarcity; the cost of moving the application may exceed the benefit, etc. Furthermore, how does moving an application to the "other side of the fence" affect those applications that choose not to (or can't) move? Does the mobile application benefit at the expense of the non-mobile applications that share the system's resources? The "benefits" of mobility might depend on your point of view.

This is a study of the effects of mobility on both a mobile application and the other applications in the same system. We use a trace-driven simulation to measure the performance benefits of moving a data-intensive search application (`agrep` [Wu91]) from a client to the file server. By varying the server memory size and the server speed relative to the client, we can identify those system configurations in which `agrep`'s performance improves when it is moved to the server, and those in which it slows down. We also measure the server's response time in servicing file access requests from other clients, to study the effect of the mobile code on other applications that compete for the server's resources.

Our results show that predicting the benefits of mobility is difficult. Under extreme circumstances mobility has obvious benefits, e.g. it is clearly a win to move a data-intensive application from a slow client to a fast, lightly-loaded server to avoid transferring data over a slow network. Often it isn't that easy to predict the benefits. Dealing with this complexity is a major challenge to people designing applications that use mobile code and the systems that support it. Mobility isn't always a win, and predicting when it is and when it isn't is a complex function of the application behavior, the performance and capacity of the system resources, and the behavior of the other applications that share those resources. Server memory does not appear to be a significant problem; even with relatively small server caches, the hit rate does not change dramatically when the cache is shared with mobile code. Moving `agrep` to the server does reduce the total amount of work done by the system, by eliminated network protocol overhead, but increases the amount of work done by the server itself. The server CPU saturates if it is only half the speed of the client CPUs, leading to an extremely high `agrep` running time and extremely high access latencies. This sensitivity to server CPU speed is particularly worrisome because the file server CPU has long been known to be a bottleneck in network file systems [Lazowska86].

2 Benefits of Mobile Code

Mobile code offers application developers new latitude in using the system's resources. No longer must remote resources be accessed remotely; instead, the application can move to the resources and use them locally. This can reduce both network traffic and network protocol overhead on both the client and the server. Under the right circumstances mobility reduces the total amount of work done by the system, improving the performance of the entire system. Under the wrong ones, the entire system slows down.

The expectation that a mobile application's performance will improve when it is moved to the server is based on several assumptions. First, the overhead of transferring data between the server and the client via the network is a significant fraction of the application's run-time. Second, sufficient resources exist on the server to run the mobile application efficiently. The server must have enough free memory to run the mobile application without excessive paging. The server CPU must have enough idle cycles to satisfy the mobile code. Third, the performance of the server resources must be adequate relative to the client. It won't do the mobile application much good to move it to a server with a slower CPU and memory system.

It is easy to overlook the mobile application's effect on the other applications in the system. The service provided to other applications may degrade because the mobile application consumes the server's resources. At a minimum, the mobile application requires memory and CPU cycles, reducing the amount of both available to service other applications. Server memory consumption is particularly worrisome, because reducing the amount of memory available for the server cache increases the number of server disk accesses, which can have a big effect on the server performance. In general, running an application on the server requires more server memory than running it on the client, due to memory pages needed to hold the program text and data. Moving the application to the server also presumably reduces its run-time, increasing the rate at which it issues file access requests and having a greater effect on other applications that access the server. This can increase the variance of the server access time. On the other hand, moving the application to the server reduces the amount of network protocol processing the server must do, perhaps freeing those resources for use in servicing other applications. It also reduces network traffic, decreasing network congestion and increasing file access performance.

Our hypothesis when we began this study was that the amount of memory available on the server, and the ratio of the server CPU speed to the client would have a big effect on the benefits of mobility. Under some circumstances, it would actually slow the system down as a whole to move an application from the client to the server. We tested this hypothesis using a trace-driven simulation. This allowed us to vary these system parameters, and measure the effect on the mobile application and the background jobs.

Although there may not always be a benefit for a client or the system in remotely executing code on a single shared server, benefits may be had from exploiting parallelism when there are multiple servers. Riedel et. al. show a speedup of 2x for running application-level database code across 10 Active Disks versus a single-processor specialized database server [Riedel98]. Active Disks contain relatively low-powered CPUs and act as storage servers. To maximize the performance and scalability of such a system, it is important to understand how resource utilization at a specific server influences the rest of the system. Our study focuses on one server.

3 Experiment Setup

For our experiments we performed a trace-driven simulation of a mobile application, `agrep` [Wu91], running on the client of an NFS file server versus on the server itself. The server also serviced background requests from non-mobile applications, represented by a trace of NFS activity [Dahlin]. The server's performance in handling these background requests reflects `agrep`'s effect on the system as a whole.

3.1 Agrep Trace

agrep is a popular tool for efficient full text searching, and seems a likely candidate to benefit from mobile code. It searches entire files for strings that match a given expression, requiring it to perform a relatively small amount of computation relative to the amount of data it processes. Of the many commonly used applications that may benefit from being executed remotely, agrep is data-intensive with only minimal processing needs. Thus, results using agrep should be representative of what can be gained using mobile code in storage systems. As input to our simulator we used a trace of agrep's behavior collected by Tomkins et. al. [Tomkins97]. To create the trace, agrep ran on an instrumented Digital UNIX kernel, and searched a kernel source tree of 1400 files comprising 23 MB.

Each record in the trace contains an absolute time, opcode, file identifier, offset, and size. The opcode distinguishes opens, closes, reads and writes. The data associated with the I/O operations are not available in the trace, nor is disk layout information. This prevents the simulator from accurately simulating the file system and disk.

The records were created by a run of agrep, and presumably are dependent: an I/O is initiated only after the previous I/O completes and the data processed by agrep. The time between the completion of one request and the start of the next is assumed to be agrep computation. This interval is scaled in the simulation by the CPU speed so that the I/O rate is higher on a faster CPU, but I/Os are never overlapped.

Simulating agrep properly requires knowing agrep's instantaneous memory requirements during its entire run. This information allows the simulator to allocate the minimum amount of memory to agrep at all times. Unfortunately, this information is not available in the trace, so we make the very conservative assumption that agrep's memory footprint is at all times equal to its maximum footprint. During a run of agrep on inputs of similar size, we determined that its memory image grew to a maximum size of 800KB; we therefore assume that agrep requires 800KB during its entire run.

3.2 Background Trace

We simulate background requests to the server using a trace of NFS requests to an Auspex NFS server [Dahlin]. These traces contain all NFS traffic to an Auspex file server at UC Berkeley over the period of seven days. The traces were obtained by snooping on the network, and therefore only contain NFS requests to the server; application requests to the client caches are not present. The Auspex traces therefore reflect NFS client behavior, and not application file-access behavior.

Each request in the trace contains an absolute time, NFS opcode, file identifier, offset, size, and client identifier. The NFS opcode distinguishes attribute requests from block requests, and reads from writes. Like the agrep trace, the Auspex trace does not contain the actual data accessed, nor does it contain disk layout information. This prevents accurate simulation of file system and disk overhead.

The trace also does not contain information about related events issued by a client. The only context known about a particular event is the time at which it occurred, and the client that issued it. It is likely that the events form a partial ordering: some I/O requests undoubtedly depend on the completion of previous I/O requests. Unfortunately, the trace does not capture these relationships. It seems unlikely, however, that all events related to a single client are dependent, as in the agrep case, so that the next event should not be issued until the previous event has completed and the client's think time expired. We therefore make the simplifying assumption that client requests are independent. Each request from the client to the server is issued at the absolute time recorded in the trace (scaled by the client processing speed), regardless of state of previous requests. It is important to note that this assumption will affect our results since some of the requests probably *are* dependent.

3.3 Simulator

The performance of a network file server is measured by its response time to client requests. The simulator we have implemented models a network file server that can host mobile code in addition to servicing traditional file service requests.

The operation of the simulator is straightforward: client requests are read from the traced workloads, interleaved based on the traced timing information, and then sent through various simulator modules which calculate the request response times. The response time is determined by the speed of the server CPU, the server memory, the server disk, and the network, as well as any queuing delay at those resources. A request can also become queued waiting for the cache if no free blocks are available.

3.3.1 Cache Simulation

All caches are fully associative with an LRU replacement policy. Blocks are locked during I/O operations, such as reading data from the disk into the cache. A request that tries to access a locked block is queued until the block is unlocked. Under overload situations this means that server accesses may find the entire cache locked, causing all requests to be queued and increasing the server access latency.

All caches are write-through. Although write-back caches would perhaps be more realistic, they complicate the performance measurements. Dirty blocks in the cache represent potential work (i.e. work that has been deferred but must eventually be done). This potential work must be accounted for in the simulation; a server that defers all disk writes until after the measurement interval will look very fast compared to one that doesn't. Also, the write-back of dirty blocks is typically started when the number of dirty blocks exceeds a threshold. An accurate comparison is possible only if the number of dirty blocks is the same at the start of each measurement, otherwise it will affect the number of pages written back during the measurement interval. For these reasons we chose to use a write-through cache in our current experiments.

The cache simulator also supports infinite caches, those in which capacity misses do not occur. This results in the highest possible hit rate, because the only cache misses are the capacity misses that occur when a block is accessed for the first time. The simulator implements infinite caches by increasing the cache size on each cache miss.

The cache simulator reduces the size of the server block cache by the number of pages needed by the `agrep` program for its text, data, and stack. This amounts to 800KB, or 100 blocks.

When the `agrep` is run on the client its file accesses are filtered through a client cache, in the same manner that the Auspex trace records server accesses downstream of the client caches. As a result, when `agrep` runs on a client its accesses should "blend in" to the server access stream, and the characteristics of its requests should be similar to the background traffic. We make the assumption that there is no file sharing between the `agrep` and the background traffic, so that the `agrep` only hits in the server cache for blocks that it previously accessed.

Metadata requests comprise 87% of the requests made by the background trace and 42% of the requests made by `agrep`, averaged across all six intervals. The server metadata cache is infinite, resulting in a hit rate greater than 90%. Although the traces contain some requests for less than a block of data, the sizes are rounded up by the simulator.

3.3.2 Disk Simulation

The server has one disk, with a peak transfer bandwidth of 10MBs and an average seek time of 8ms. The disk handles one request at a time, and the simulator assumes that each request requires an average seek to the proper track. The lack of data and disk layout information in the traces makes a more accurate simulation infeasible, and therefore a more accurate disk model unnecessary. The disk latency for a request is the amount of time that request spends in the disk queue plus the actual access and transfer time.

$$\text{disk latency} = \text{queueing} + 8\text{ ms seek} + \frac{\text{request size}}{\text{10MBs disk}}$$

3.3.3 CPU Simulation

The server CPU is used both to perform protocol and file system processing on traditional file access requests and to perform computation for mobile code running at the server.

	Read Cycles (thousands)	Write Cycles (thousands)
Metadata	33	64
Block Data	100	199

Figure 1: Average cost of I/O operations

To reduce the effect mobile code running on the server has on background requests as much as possible, the simulator gives priority to file access requests. If the `agrep` is running when a request is received, the server immediately blocks the `agrep` and processes the request. The `agrep` receives only idle server CPU cycles.

Our model of CPU requirements is very simple (Figure 1), and assigns a constant number of cycles to each type of server access. Our numbers are derived from those published by Gibson et. al. [Gibson97], who measured the CPU cost of different types of NFS requests. These costs were calculated for a 133 MHz Digital AlphaStation. The number of cycles required for a given request is scaled by the speed of the server, to account for the different server CPU speeds used in different experiments. Our baseline speed is 133 to match the table.

$$\text{CPU latency} = \text{queueing} + \frac{\text{table value}}{\text{server speed}}$$

3.3.4 Network Simulation

The network model is based on a 100Mbps switched Ethernet. Network latency is incurred by both client requests and server replies. The latency of a network data transfer is modeled as the sum of a hypothetical link latency and the data transfer time of the network:

$$\text{network latency} = 20\ \mu\text{s link} + \frac{\text{data size}}{\text{100Mbps netw.}}$$

This simple model does not capture network contention; we assume that the network possesses sufficient aggregate bandwidth to avoid this problem.

3.3.5 Simulator Validation

To ensure that the simulator was behaving correctly we performed several tests that singled out the effect of important variables. The problem with simply running the simulator on many input samples is that the number of interdependent variables can hide problems. By eliminating the network, and using an infinitely fast disk, we were able to single out the CPU for analysis. Removing the cache and using an infinitely fast CPU singled out the disk. Once confidence can be placed in the individual components, remaining problems can be chased by performing a number of more realistic tests, for which the results are predictable.

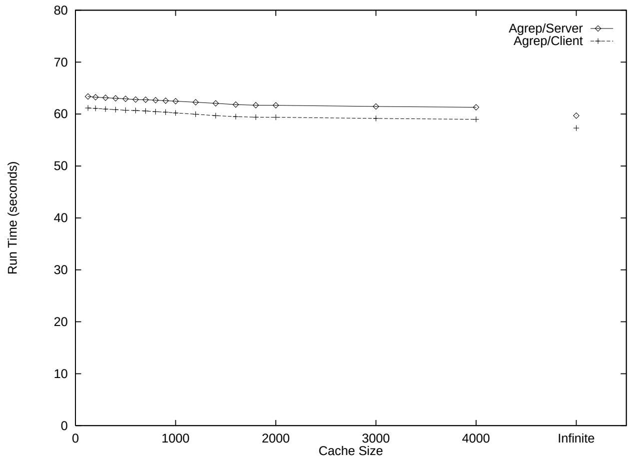

Figure 2: `agrep` run-time as a function of the server cache size. `agrep`'s file accesses have poor locality, so larger server caches provide little benefit. The syntax *which*/*where* indicates which traffic is measured (`agrep` or Background) and where `agrep` was running (Client or Server).

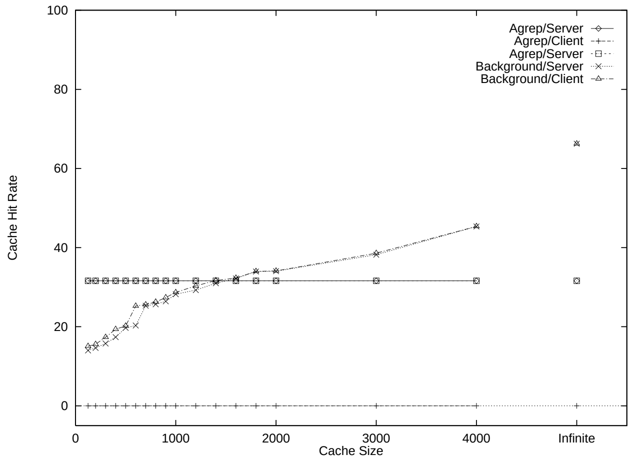

Figure 3: Percentage of block requests that hit in the server cache, as a function of the cache size. Background traffic benefits from a larger server cache, `agrep` does not.

4 Experiment Results

4.1 Measurement Intervals

The entire Auspex trace comprises 168 hours. `agrep` typically completes within two minutes on our simulated hardware. Running the `agrep` against the entire Auspex trace would obviously have a minimal effect on the background performance. We are interested in its effect on those accesses that occur while the `agrep` is running, or after it runs but while it still has lingering effects on the system. To do this, we extracted the six three-minute intervals with the most server accesses from the trace. For each interval the simulator was run on the preceding three hours of the background trace, to warm the caches. Then the `agrep` was run at the start of the interval, and the traces processed until the end of the interval. Because we use the busiest intervals, the blocks accessed by `agrep` are flushed out of the server cache relatively quickly, eliminating any lingering effects caused by running `agrep` and limiting the number of background accesses we must consider in our measurements.

To summarize, the results of our tests are taken from the short periods of time during which `agrep` affects the system. Prior to each `agrep` run, a long trace of background traffic is played to warm caches. These long traces were taken from the busiest periods of the whole Auspex trace to make the conditions most challenging for `agrep`.

4.2 Block Cache Size

Our hypothesis was that the size of the server cache would have a significant effect on the `agrep` run-time. Surprisingly, we found that this was not the case (Figure 2). `agrep` makes a single sequential pass through its input files, resulting in low hit rate on the server cache, independent of its size.

Increasing the server cache size does not improve `agrep`'s running time, and therefore we would expect it doesn't improve `agrep`'s hit rate on the server cache either. Our simulation shows this to be true (Figure 3). When `agrep` is run on a client any locality in its accesses is filtered out by the client cache, resulting in a server cache hit rate of close to zero.

The background traffic exhibits more locality than `agrep`, and as a result it benefits from a larger server cache. When `agrep` is run on the server it consumes 100 blocks from the server cache, reducing the background hit rate. The effect is pronounced with a small server cache. This is hardly surprising: when server memory is valuable it is costly to use it to run mobile code.

The hit rates on the server cache directly translate into access latencies (Figure 4). As the server cache grows, the access latencies for the background traffic drops because more accesses hit in the (fast) cache. `agrep`, on the other hand, has poor locality, so its access latency is close to the disk latency. The difference between the access latency of `agrep` running on the client vs. on the server is caused by the network latency and the overhead of network protocol processing. The background latency is never close to the disk latency, even for small caches, because of its high fraction of metadata requests.

A breakdown of access latency (Figure 5) reveals that faster response times with larger block caches are the result of higher system resource availability, and therefore reduced queuing delays. With very small caches the disk queuing delays are especially evident.

The CPU and network metrics are flat because all requests are of fixed size, and without queueing delay, the

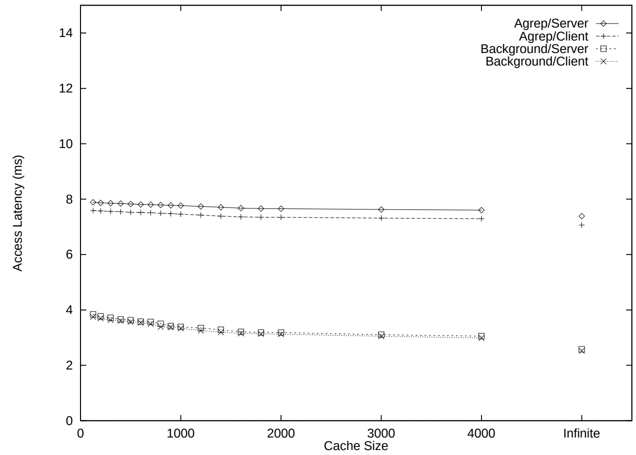

Figure 4: Average request response time observed by the client, for all requests (including metadata) as a function of server cache size.

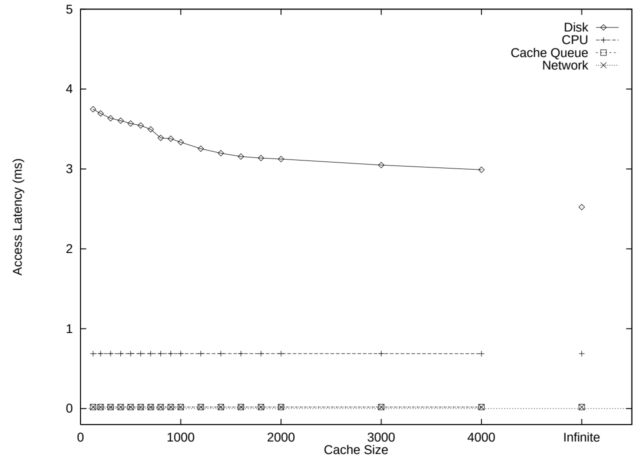

Figure 5: Breakdown of the average request response time for the background load during the runs of `agrep` on a client machine for varying cache sizes.

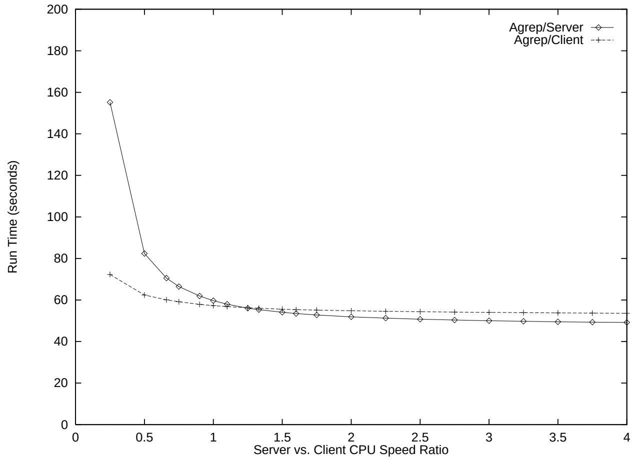

Figure 6: Run-time for `agrep` vs. server CPU speed

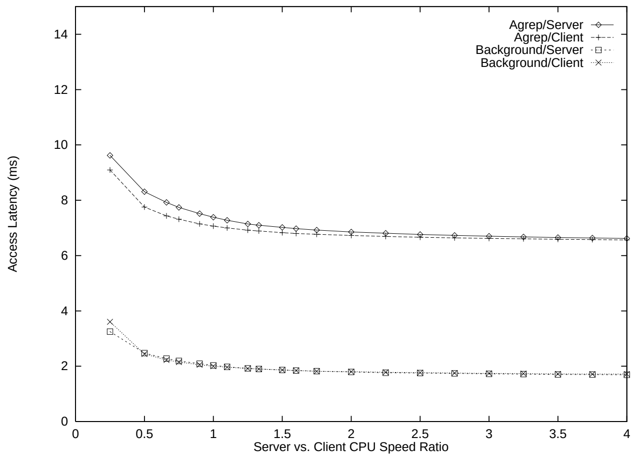

Figure 7: Access latency for all requests (including metadata) vs.server CPU speed.

result of the simple modeling algorithms always predicts the same value for these requests.

4.2.1 Summary

`agrep` is largely unaffected by the server block cache size, while the background load performs noticeably better with larger caches. This is exacerbated by `agrep` effectively shrinking the cache when it runs on the server. However, this effect is prevented from becoming severe by the large percentage of metadata requests in the background load which are not affected by the size of the block cache.

4.3 Server/Client Speed Ratio

Another factor that affects the benefits of mobile code is the speed of the server relative to the speed of the client. Mobile code benefits from a faster server, but can suffer from a slower one. As one might expect when the server's CPU is faster than the clients', the `agrep` benefits from running on the server (Figure 6). This performance improvement is bounded however, because of the relatively small amount of processing `agrep` performs. Most of the run-time is spent in the disk subsystem. A slower server hurts `agrep`, because it would have completed faster if it had stayed home on the (faster) client. `agrep`'s performance degrades dramatically as the server slows, to the point of starvation, because background requests receive priority and they take relatively longer to process on a slow server.

The server access latencies exhibit similar behavior (Figure 7). As the server CPU slows, the access latency for both `agrep¡` and the background traffic increases because the server CPU begins to saturate.

5 Related Work

Acharya et. al. [Acharya98] have modeled a system of Active Disks, similar to that of Riedel. They also propose a "stream-based programming" model of mobile code, and estimate performance improvement for a number of applications using their system. This work complements ours by suggesting a specific platform for mobile code in storage systems.

Ruemmler and Wilkes describe the implications of accurate disk modeling [Ruemmler94]. Wilkes also describes file system simulation with Thekkath and Lazowska [Thekkath92]. This work explains it is dangerous to rely too heavily on abstract models. The level of rigor which they describe as desirable would have been preferable for our simulation had the necessary data been available in the traces we used. However, we believe our assumptions are sufficiently conservative to predict the coarse trends described in our conclusions.

Cache analysis and simulation of caching behavior been looked at by Shirriff [Shirriff91], Dahlin [Dahlin], and Welch [Welch91]. Our work extends the principles of cache analysis to the domain of mobile code.

Mobile code for systems level applications has been explored by Bershad [Pardyak96] and Douglis [Douglis91]. This work has been primarily concerned with the mechanics and practical implementation concerns of mobility. Our work is different in that we take implementation for granted, and focus on the performance of its application to the specialized area of file systems.

6 Conclusions

In this paper we consider the potential benefits of using mobile code on file servers. We implemented a trace-driven simulator to discover under which circumstances mobile code improves performance, and to discover the cost to other applications.

The expectation that mobile code allows for significant performance improvement was based on several assumptions, some of which proved to be incorrect. Initially, we expected that resource availability on the server, specifically cache size and a fast CPU, would be critical both to mobile code performance and to minimizing the cost for background requests.

The simulations show that server cache size is not very important to predicting the effects of mobility. The access locality of our mobile application is low, and the cache hit rate of background requests is not significantly affected by the mobile code's presence. The server CPU speed has a more pronounced role. If the server CPU is highly utilized and few cycles are available for mobile code use, the performance of mobile code will be poor. Despite these discoveries, we still expected that eliminating data transfer between the server and the client would make mobile code attractive. Unfortunately, we overestimated the overhead associated with current low latency, high bandwidth networks. The time lost to the transfer is not significant.

Our experiments demonstrate that predicting the benefits of mobility is difficult. Although extreme circumstances such as very high latency congested networks, can still result in obvious performance gain, the benefit of mobile in current research environments is not clear cut.

References

[Acharya98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms, and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, 1998.

[Dahlin] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis scalability for network file systems. Technical Report CSD-94-798, University of California, Berkeley.

[Douglis91] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice & Experience*, to appear. An earlier version is available as Computer Science Division (EECS), University of California, Berkeley, Technical Report UCB/CSD 89/540, November 1989, 1991.

[Gibson97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25,1 of *Performance Evaluation Review*, pages 272–284, New York, June15–18 1997. ACM Press.

[Lazowska86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *TOCS*, 4(3):238–268, AUG 1986.

[Microsystems95] Sun Microsystems. The java language overview. *White Paper*, 1995.

[Pardyak96] Przemyslaw Pardyak and Brian Bershad. Dynamic binding for an extensible system. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 201–212, Berkeley, CA, USA, October 1996. USENIX.

[Riedel98] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*, New York, USA, 1998.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[Shirriff91] Ken W. Shirriff and John K. Ousterhout. A trace-driven analysis of name and attribute caching in a distributed system. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 315–332, Berkeley, CA, USA, January 1991. Usenix Association.

[Thekkath92] Thekkath, Wilkes, and Lazowska. Techniques for file system simulation. Technical Report TR 92-09-08, University of Washington, 09 1992.

[Tomkins97] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25,1 of *Performance Evaluation Review*, pages 100–114, New York, June15–18 1997. ACM Press.

[Welch91] Brent B. Welch. Measured performance of caching in the Sprite network file system. *Computing Systems*, 4(3):315–342, Summer 1991.

[Wu91] Sun Wu and Udi Manber. agrep - A fast approximate pattern-matching tool. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 153–162, Berkeley, CA, USA, January 1991. Usenix Association.