DataMesh research project, phase 1

John Wilkes
Hewlett-Packard Laboratories
wilkes@hpl.hp.com

Work performed jointly with Chia Chao, Robert English, David Jacobson,
Sai-Lai Lo, Chris Ruemmler, Bart Sears, Alex Stepanov and Rebecca Wright

1 Introduction

This position paper describes work taking place in the DataMesh research project [Wilkes89, Wilkes91] on *concurrent file systems*. A concurrent file system exploits parallelism in its construction, while presenting an external image that is compatible with existing client:server interface definitions. The work is described in a number of stages:

- the DataMesh hardware architecture is introduced;
- Jungle, the overall software architecture framework, is described;
- some existing results are described to demonstrate progress so far.

The DataMesh project is the overall umbrella activity for the research we are carrying out in the area of concurrent file systems. We believe our work is of considerable interest to future file system designers, as well as people interested in understanding and modifying existing file system designs.

The first phase of the DataMesh project is providing block-level services to its clients: the interface is in the form of read / write operations on sets of fixed-size data blocks. (You might like to think of this as the regular SCSI disk command set.) Phases 2 and 3 will provide interfaces at the file and record level respectively. Our design center is for a single DataMesh to contain perhaps 100–200 nodes.

We acquired our first DataMesh hardware preprototype in the summer of 1991 (we plan to replace it with something more closely resembling the scale and performance of our 1995 system over the next year and a half); current work emphasizes the development and evaluation of our software ideas using it. The work described here is in the development stages and doubtless subject to modification as we go along.

2 The DataMesh hardware architecture

Our goal is to develop a system storage architecture to provide high performance, high availability, scalability (in both size and component type), and standards-based connections to the open systems environment. We believe that these requirements in turn suggest particular solutions:

- *high performance*: the use of parallelism and the close coupling of processor power with storage elements;
- *high availability*: no single points of failure (i.e., there must be built-in redundancy), coupled with fault tolerant software;
- *scalability and long life*: a modular architecture, to allow a DataMesh server to expand and adapt to changing requirements over time, with smooth incremental growth;
- *open systems interconnection*: the ability for a DataMesh server to attach to the outside world through several hardware and software interconnect standards.

Our chosen hardware solution is an array of nodes of various types:

- – *port nodes* provide connectivity to the outside world through an I / O interface like SCSI or a LAN like FDDI;
- – *disk nodes* for storage;
- – *RAM nodes* (volatile or non-volatile) for caching, read-ahead, and write-behind;

– and *tertiary storage nodes* (e.g., an optical jukebox, R–DAT tape, or robot tape library).

The nodes are linked by a fast, reliable, small-area network, and programmed so that they all cooperate to appear as a single storage server to its clients.

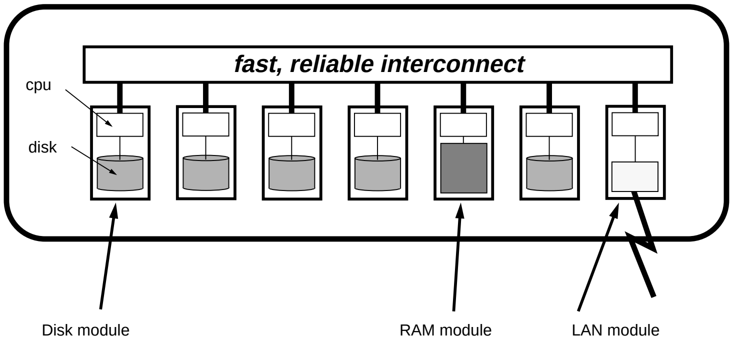

Figure 1. *System hardware model.*

By about 1995, we believe that current trends in silicon process and magnetic recording technologies will mean that each (low cost) disk node will comprise a dedicated 20 MIPS single-chip processor with about 16 MB RAM—as the disk controller—and a 2GB 3.5” disk drive mechanism. The inter-node interconnect will be capable of 10µs round-trip latency and 10MB/s bandwidth per node, both scalable up to several hundred nodes.This interconnect will be highly fault-tolerant by virtue of a great deal of built-in redundancy. And the resulting system will cost almost the same as a simple array of regular disks with similar capacity. Existence proofs for all these statements (except the cost!) can be found in research prototypes or commercial products today.

3 The Jungle software architecture

> **Jungle**, *n*. (Area of) land overgrown with underwood or tangled vegetation, especially in tropics; scene of ruthless struggle for survival; wild tangled mass. [from Hindustani *jangal* from Sanskrit *jangala* desert, forest]
>
> — *The Concise Oxford Dictionary of Current English, Oxford University Press, 6th Edition, 1976*

The overall DataMesh software architecture is called Jungle. It is designed to exploit the replicated, distributed hardware, the low latency interconnect, and the dedicated processing power in a DataMesh server. As a result, much of the Jungle architecture is concerned with things like maintaining data coherency for correctness, replication and redundancy for fault tolerance, and local caching for fast access.

Since Jungle was intended as a framework into which a number of specific algorithms and policies can be fitted, the novel ideas it contains are more to do with managing collaborating components than with particular policies. Here are a few of the things that we consider important about it:

- Encapsulation of policy decisions in *managers*. Several different policies can co-exist, with each manager specialized to serve one set of needs optimally.

  The policies will optimize for different use patterns or storage needs. For example: multiple file access methods can present the same external interface, but one will be optimized for groups of small files, another for very large ones that are always accessed sequentially. Similarly, extensible byte vectors can be implemented in many different ways: e.g., fixed-size page allocation, or a buddy-system-based extent map.
- Jungle will provide system-wide management of physical resources such as RAM caches and storage devices.

For example, cache RAM in any of the Jungle nodes is treated as part of a global pool to be allocated to claimants, whose needs are assessed (and responded to) on a system-wide basis.

- Jungle supports descriptions of performance and availability properties of storage devices and the abstractions developed on top of them, such as files. With this we are able to provide mechanisms to match user-specified needs to available storage resources (a description of our approach may be found in [Wilkes91a]). Sample applications include selective high-availability and multi-media performance guarantees.
- Upper layers will be able to extend the functionality of lower ones by downloading interpreted scripts that express the upper-layer policies, but execute in an environment that is tightly bound to the data—thereby avoiding unnecessary and costly protection and machine boundary crossings.
- Jungle provides explicit handles (called *operation groups*) on the sequencing and completion properties of related sets of function calls.

3.1 Overview of the Jungle structure

The underlying model of Jungle is of a low-level, smart *chunk store* that holds raw bags of bytes, on top of which there is a layer of *chunk vector managers* that provide an abstraction composed of sequences of chunks. Finally, there is a layer of *thing managers* that provide application- and system-specific interfaces to the stored data.

Jungle software runs on both DataMesh server and client nodes. Figure 2 shows the layer structure of the major Jungle components, without making visible the hardware boundaries. In practice, the "upper" levels will reside on the workstations, the "lower" ones in the DataMesh server itself. The objects manipulated inside Jungle are:

- *Devices*: suppliers of *slots* where data can be stored. All the slots in a single device have the same set of attributes (performance, availability, cost, etc.). A single logical device may be constructed from one or more physical storage elements (e.g., a disk array); can potentially perform replication; and can choose to hide multiple layers in the storage hierarchy (e.g., an optical jukebox front-ended by one or more disk drives).
- *Chunks*: individual pieces of storage to be stored in device slots. (Chunks are our abstractions of the "blocks" commonly found in file systems.) A chunk has a fixed size over its lifetime, determined largely by what is optimal to store on the physical medium with which it is associated; different chunks may have different sizes.
- *Chunk Vectors*: sequences of contiguously-addressed chunks. Every chunk belongs to a single chunk vector. Chunk vectors can be extended only by whole chunks: they are like a very low level UNIX file abstraction.[1] Chunk vectors may provide or be associated with performance

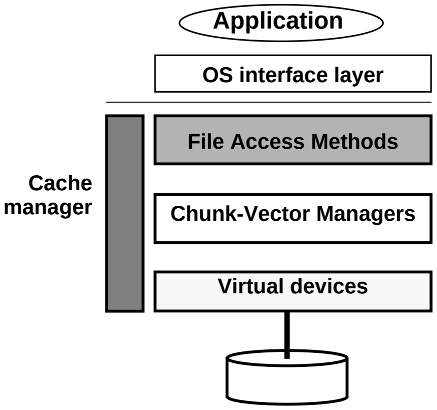

Figure 2. DataMesh *software architecture—simplified logical layering*

and availability properties, such as “always accessed in sequence”; “minimum aggregate bandwidth of 5MB/s”; “maximum 30 second downtime”.

- *Upper Level Jungle Things (ULJT)*: application-visible objects with byte- or record-level interfaces. Access to these functions are provided by *Jungle-Thing Access Managers (JTAM*s)—also called “(file) access methods”. Examples of ULTJs are UNIX files, databases, key-sequenced files, and persistent objects managed by a language runtime system.

Several implementations of each component can be active and available concurrently. For example, there will be chunk vector managers optimized for different performance properties: UNIX-file JTAMs optimized for different file sizes will coexist—one for many small files per chunk vector; one for single file per chunk vector (rather like existing UNIX file systems); and one for extremely large files that need to span multiple chunk vectors.

In addition, the Jungle architecture provides support for the following secondary objects:

- *Caches*: RAM memories that can be used to store chunks in low-latency memory. Cache managers respond to requests for physical memory from other managers, trying to satisfy those that are most “meritorious”. (We plan to explore different algorithms for expressing and calculating such merit information.)
- *Operation groups*: sets of invocations of Jungle functions that should be considered together in some fashion for performance or availability reasons. For example: “do these in this order”, “all these should be atomic”; “sequencing within this group is unimportant”; “*this* group should all happen before any of *that* group”.
- *Scripts:* small interpreted programs that the upper levels of Jungle use to augment the functionality of lower layers. Once a script has been downloaded, it acts as a subroutine that can be called. Scripts will be expressed in a simple, compact, interpreted language.

Our programming model follows that of the proxies of [Shapiro86]: any access to an object is through a local copy of the manager code. Although this may sound a little extreme at first sight, we also allow the local manager code to be but a shadow of the real thing—for example, a stub (perhaps with some local caching, perhaps not), that just forwards all requests to a remote copy of the full manager code.

4 DataMesh phase 1

DataMesh phase 1 is concerned with the Jungle *device* layer. Although this might be thought of as having little to do with file systems *per se*, we have learned that there are a great many advantages to considering the designs of both the file system and the device itself simultaneously.

The basic goal of the DataMesh phase 1 work is to develop high performance, flexible block-level servers. The interface to these servers is at the level of read and write operations on fixed-size blocks (chunks) of data. Initially, we will adhere to the existing SCSI command set so that the devices we are developing can be added directly to an existing system; in later work we plan to make minor extensions to it (such as the provision of a free command, and a trial implementation of the SCSI-3 disk-based file system proposal) to enable some of our more aggressive ideas to be tested out.

Our approach is straightforward and direct: we are constructing a prototype hardware/software testbed (the DataMesh *preprototype*); gathering and synthesizing a wide range of workloads; and using these to evaluate a range of different new disk system designs.

4.1 Phase 1 infrastructure

The DataMesh preprototype is constructed from an 8-node transputer system manufactured by Parsytec, running the Helios operating system from Perihelion. Each node has an Inmos T800 transputer, 4MB RAM and a SCSI controller chip. Seven of the nodes have an attached 5.25” SCSI disk drive; the eighth has a SCSI connection to a host workstation. The whole assemblage is attached to an HP C(an IBM PC clone) through which access to the building LAN is provided. We are able to

[1] UNIX is a registered trademark of Unix System Laboratories in the USA and other countries.

cross-compile software on our workstations, download it into the testbed, and debug it—all without leaving our offices.

On the testbed we have already developed a SCSI target-mode device driver so that we can send requests to it from the workstation as if it were a disk drive; a trial implementation of a parallel RAID 5 disk array; and a measurement and performance monitoring harness.

We have been gathering and using a number of real-system workload traces to drive our simulations. The HP-UX operating system supports an interface known as the measurement system, which can capture system events with 1µs timer resolution. Typical events include file system calls, file buffer cache misses (and hits), and disk device requests—enqueues at the device driver, and physical I/O start and completion. This is a powerful tool to use for testing the effects of device designs on file system performance. Our traces cover several months worth of activity on a single-user workstation and a local timesharing system; we are currently extending both the breadth and detail of our trace collection.

We have also put together a number of tools for synthesizing workloads given descriptions of patterns of accesses [Wright92].

4.2 Some of our results so far

Modern disks have different properties than ones studied intensively by file system designers in the 1970s. For example, it now takes only 1.3 revolutions of the disk for the head to seek from the outside edge to the inside, which suggests that existing request-scheduling algorithms (such as the SCAN algorithm used by UNIX [Coffman72]) that order requests by seek distance are no longer optimal. By taking advantage of rotation position information we have developed a family of algorithms that provide much better throughput while also avoiding starvation effects. About double the throughput can be obtained for a constant response time on real traces [Jacobson91, Seltzer90]. We are continuing this work in collaboration with Giorgio Gallo at the University of Pisa.

While the previous technique improves throughput significantly under high loads, it does little to help improve latency (other than to reduce queueing delays). A separate idea, published as [English92] improves write latencies. We called the approach *Loge*: it uses an indirection table in the disk to map logical block addresses to physical ones. In combination with around 3–5% of the disk reserved as free blocks (in much the same way that the 4.2BSD file system [McKusick84] reserves 10% of the disk to improve layout performance), write latencies for small (4KB) writes can be roughly halved, with minimal effect on read performance; Loge can also sustain roughly half the raw bandwidth of the disk for random small writes. The trick is to write to the nearest available free block: by doing so, seek and rotational latencies can almost be eliminated for small transfers. The benefits are obviously relatively greatest for the smallest transfers, especially those involved in synchronous writes, such as forcing data or the tail of a log file to disk for a commit. (Such writes can of course be avoided by file system restructuring: LFS [Rosenblum91] delays writes—thereby sacrificing availability—to achieve larger asynchronous transfers; the clustering techniques of [McVoy91] improve large sequential transfers, but don't improve random I/O.)

Although the above two techniques are best applied in the disk itself because they rely on fast access to rotation-position information, the next idea could be applied in the file system as well—possibly with even better results. (Indeed, some similar work has been done in this area on whole-file layout [Staelin91].) The presence of the Loge indirection table gives us additional benefits beyond the fast writes: we can now consider optimizing the layout of the disks on a per-block basis for better read performance, for example, by shuffling the most actively read data to the center of the disk. An investigation of techniques to do this [Ruemmler91] suggested that performance improvements of up to 20% can be achieved on an *already optimized* 4.2BSD file system layout, and should be able to do much better with the additional randomness introduced by Loge. The ability to do per-block reorginzation was crucial to the performance gains: larger units showed much less impresive gains. An important point about this approach is that repetitive sequences of random I/O can be well handled by this technique, while most other work merely improves sequential I/O. We believe this technology could usefully be combined with LFS-like techniques to improve the cleaning function and arrange for data that is accessed together to be physically co-located. We are continuing this work in collaboration with Dave Musser at Rensselaer Polytechnic Institute. One thing we learned is that—even in the presence of large file system caches—there is a surprisingly high locality in disk accesses.

Finally, we have investigated the effects of replicating data dynamically to achieve better read performance. (We did this work on whole files, unlike the rest of the results reported here.) The basic idea is to make multiple copies of frequently-accessed data—our experiments used multiple disks, but this could also be done on a single disk—and select between the copies at read time for lowest-latency accesses [Lo90]. Straightforward replication can increase update times (since multiple copies all have to be written to), so we experimented with different policies for making and discarding replicas dynamically. Our results showed that (a) dynamic replication can provide up to about 25% reduction in average file system disk access time; (b) keeping only one copy on an update is roughly twice as good as keeping them all; (c) within these parameters, all the policies that we tested worked about equally well. We believe these incremental benefits should also be obtainable from more aggressive file system implementations, too.

4.3 Current research

Current and near-term work in the DataMesh project is concentrating on phase 1, the block level server. Our pursuits cover a number of areas including:

- Unidisk virtual devices: fast-write disks (Loge), disk request scheduling, disk shuffling.
- Multidisk virtual devices: MultiLoge: Loge across multiple spindles.
- SCSI extensions: sparse addresses, multiple spigots, tagged data, load balancing across devices, and a SCSI–3 file system prototype.
- High availability: decentralized RAID 5, with no single point of failure and distributed parity calculation and error recovery; strong recovery guarantees combined with high performance (Mime [Chao92]).

We are currently bringing up the multiple-policy framework on the DataMesh 1 preprototype hardware, and are implementing the Loge and Mime devices for it, as well as simple disk and RAID disk array devices for comparison purposes.

5 Conclusions

The work described here represents a radical departure in the design of file systems: it considers the entire chain of functionality from record-level operations down to and including the device controller functionality. Our work to date has exposed several possibilities for greatly improved performance and functionality. Doubtless there will be more.

References

[Chao92] Chia Chao, Robert English, David Jacobson, Alex Stepanov and John Wilkes. *Mime: a high performance storage device with strong recovery guarantees.* CSP technical report HPL–CSP–92–9, Hewlett-Packard Laboratories, March 1992.

[Coffman72] E. G. Coffman, L.Klimko and B.Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing* **1**(3):269–79, September 1972.

[English92] Bob English and Alex Stepanov. Loge—a self-organizing disk controller. *Proceedings of Winter USENIX'92* (San Francisco, CA) Jan. 1992.

[Jacobson91] David Jacobson and John Wilkes. *Disk scheduling algorithms based on rotation position.* CSP technical report HPL–CSP–91–7,Hewlett-Packard Laboratories, Feb. 1991.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems,* **2**(3):181–97, August 1984.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review* **25**(5):1–15, 13 October 1991.

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk shuffling*. Technical report HPL–91–156, Hewlett-Packard Laboratories, Oct. 1991.

[Seltzer90] Margo Seltzer, Peter Chen and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX (*Washington, DC), pp. 22–26 January 1990.

[Shapiro86] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pp. 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[Staelin91] Carl Staelin and Hector Garcia-Molina. Smart filesystems. In *Proceedings of the Winter 1991 USENIX* (Dallas, TX), pp. 45–51, 21–25 January 1991.

[Wilkes89] John Wilkes. DataMesh — scope and objectives: a commentary. DSD technical report HPL–DSD–89–44, Hewlett-Packard Laboratories, July 1989.

[Wilkes91] John Wilkes. DataMesh — parallel storage systems for the 1990s. *Proceedings of the 11th IEEE Mass Storage Symposium* (Monterey, CA), October 1991.

[Wilkes91a] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review* **25**(1):56–9, January 1991.

[Wright92] Rebecca Wright, A library for synthetic multithread trace generation. CSP technical report HPL–CSP–92–1, Hewlett-Packard Laboratories, January 1992.