The Message Driven File System: a Network Accessible File System for Fine-Grain Message Passing Multicomputers [1]

Yair Zadik

yair@scp.caltech.edu

Scalable Concurrent Programming Laboratory

California Institute of Technology

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

1 March 1996

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986.

Acknowledgments

This work owes a great debt to other members of the research groups at both California Institute of Technology and the Massachusetts Institute of Technology. First among these is my thesis advisor, Steve Taylor, who provided valuable guidance. In addition Daniel Maskit, who in addition to implementing the MDC compiler and runtime without which this work could never have been done, also provided much advice and support. Kenneth Schneider wrote the J language implementation the block I/O routines upon which this work was based. Finally, Andrew Chang and other members of the Concurrent VLSI Architectures project at MIT have provided constant assistance with hardware and low level software.

Contents

Chapter 1

Introduction

This thesis describes an experimental message-driven file system and its implementation on a 512-computer J-machine with thirty-two, 402Mb SCSI disks. The J-machine [5] is an architectural experiment which focuses on the evaluation of hardware support for concurrent execution and active messages. The hardware is designed to scale to many thousands of fine-grain nodes, each of which has a relatively small local memory (1Mb) but fast communications. Active messages are supported directly through the hardware process model and integrated low latency messaging model.

The system described here is an experiment to exploit these hardware features to implement a file system optimized for the needs of scientific computing. For the target applications, it is important that files be easily transferred to and from other computers over the local network. In this way, valuable disk space on the J-machine can be reserved for running applications. Also, computations on the data which do not need the power of a massively parallel machine can be carried out on a desktop workstation instead. Ideally, the user could manipulate files and transfer them to and from the J-machine using familiar commands implemented using industry standard protocols. To meet these goals, MDFS implements a Network File System (NFS) [18] compatible interface.

MDFS is written so as to be optimized for another common characteristic of scientific codes: checkpointing. Typical scientific code will write intermediate results to disk from time to time so that long jobs can be stopped and later restarted. Typically, each node in the system reads one file of data at startup. It then alternates between computing data and writing results to disk. Each node only writes its own data, but most or all nodes will want to write at approximately the same time. Thus, checkpointing is characterized by few reads and many writes, bursts of disk operations with long periods of inactivity between bursts, and a single reader or

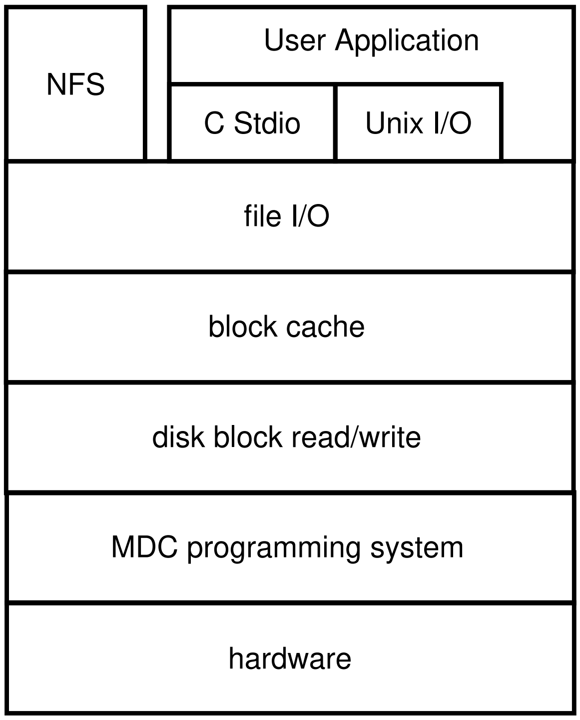


Figure 1.1: File system layers

writer per file. While optimized for checkpointing, MDFS supports other file access patterns for increased compatibility with older codes.

Within this scientific computing context, our efforts to exploit concurrent execution and active messages lead naturally to a few simple design principles: fine-grain layering and low overhead in upper layers. MDFS is composed of a collection of software layers. Each layer is a collection of servers which provide services to upper layers by using the services provided by lower layers. This produces a structured software design in which each component can be independently written and tested. It also provides the flexibility needed for experimentation as the internal design of a layer may be changed drastically without affecting other layers, and new layers may be added as decisions on functionality or global design change. Layers are *fine-grain* in that they perform small, well defined task in the overall file system. In addition to the obvious benefit that an implementation of small, well defined functions is easier than that of larger, more complex functions, this design eases partitioning into fine-grain tasks. Each request to a layer is already a fine-grain task and may be partitioned further by issuing multiple requests to lower level layers. Thus fine-grain layering provides many opportunities for concurrency.

The MDFS design attempts to minimize overhead associated with requests at the "client" nodes, i.e. those implementing higher layers. This serves two purposes: first, it maximizes resources on the client, and second, it decouples clients from servers. It is important, especially in the upper layers, that the interface minimize its usage of CPU and memory to maximize resources available for implementing the client's tasks. Additionally, minimal state should be kept on the client side in order to maximize the flexibility inherent in the interface and to maximize the opportunities for parallel execution.

Figure 1.1 illustrates the relationship of the layers in MDFS. Except for the hardware and MDC layers which may interact with all layers, each layer only communicates with the layers immediately above and below it. Within a computer, procedure calls are used for requests to the layer below and return values communicate information to the layer above. The remote procedure call mechanism in MDC can be used to execute multiple requests concurrently across nodes.

Services have a well defined interface written, whenever possible, in terms of "stateless" requests. That is, all necessary state is either encoded within the arguments of a request or derivable from them. The server needs to maintain no implicit client state to fulfill a request. For efficiency, servers may cache information to optimize common requests, but such behavior is hidden from the client. This approach is similar to the one employed in NFS[18]. While NFS uses this approach to increase reliability, MDFS uses it to minimize interdependent requests: requests can be issued in parallel without concern for their affects on hidden state.

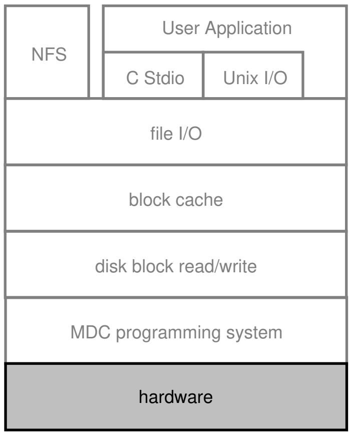
The remainder of this paper describes in detail the various layers of MDFS and the experiments performed using it.

Chapter 2

Design and Implementation

2.1 The Hardware Layer

2.1.1 Basic Hardware Mechanisms



The J-machine consists of a number of Message Driven Processor (MDP) nodes arrange in a three dimensional mesh. Physically, a J-machine consists of an stack of processor boards, each containing an 8 by 8 grid of MDPs and associated memory. The boards are stacked vertically with special connectors sandwiched between them to provide for the Z-direction connections. The X- and Y-direction connections are provided by traces on the processor boards. Additional X and Y connections are possible via connectors located around the edges of each board. The message passing network employs *deterministic worm-hole routing* for efficient communication of small messages. Each MDP computes independently of all others, having its own local memory, registers, and instruction stream. Message passing is used for all communications and synchronization between processor nodes.

The MDP provides direct hardware support for active messages accessible via machine language instructions. The active message paradigm treats messages as a means of program control as well as data movement. Thus each message is explicitly associated with code, and message reception is implicit. To facilitate this, a program running on one MDP may send messages asynchronously to any other MDP in the network (including itself) by issuing an appropriate set of assembly language `send`

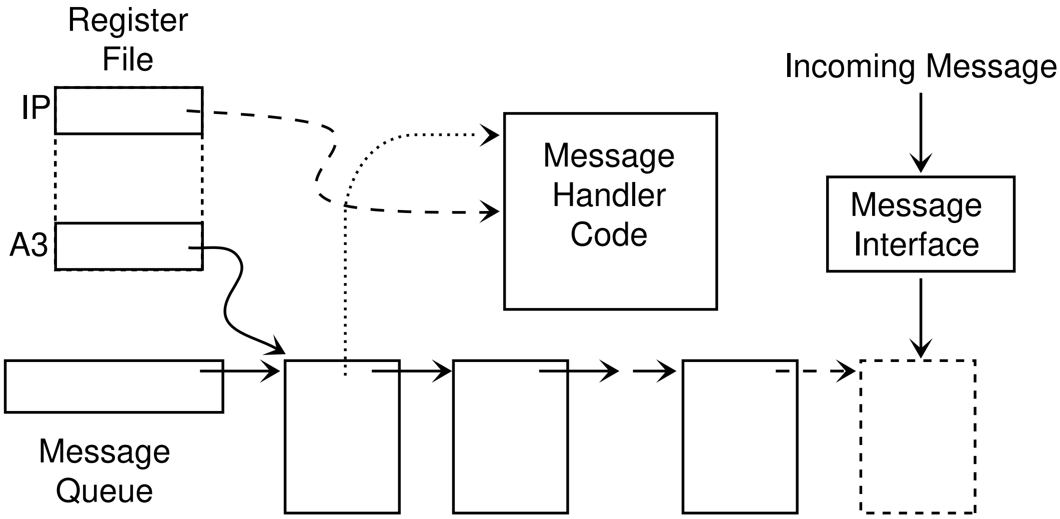


Figure 2.1: Abstract view of MDP active messages

instructions. The destination MDP needs to take no special actions for this message to be received. Upon reception, messages are place in a hardware managed queue (see figure 2.1). As part of its header, each message must contain the location of a *message handler*, a routine which will interpret the message and act on it appropriately. When a message reaches the head of its queue, the hardware places a pointer to it in a special register and dispatches the message handler automatically. Thus, a message automatically starts a thread of execution (including access to associated data) on the destination MDP. Message handlers execute until completion, at which point the associated message is removed from the queue and the next message handler is dispatched. Message reception is handle by the hardware in parallel with execution of message handlers. A handler may be dispatched prior to the arrival of the end of its message: the MDP automatically blocks when it needs data still in transit. To aid in system programming, there are two message priorities, each with its own hardware queue. Message handlers at the lower priority are preempted by message handlers at the higher priority. Additionally, there is a background context which executes when all message queues are empty. These hardware mechanisms provide the framework for a low overhead implementation of active messages.

The MDP also has hardware support for efficient synchronization. Each memory location in the MDP consists of a 32 bit value and four *tag bits* which (among other things) can be used for hardware synchronization. The tag bits can be set to indicate the location is a *future* variable, a location whose value is as yet undefined and which will be defined sometime in the future. A read of a future variable causes a fault, allowing the operating system to suspend the process until the appropriate data arrives.

2.1.2 File System and Networking Support Hardware

In addition to the mesh of MDPs, a fully functional J-machine includes a host (currently a Sun Microsystems IPX workstation), a host interface, a set of disk interfaces, and a set of Small Computer System Interface (SCSI) disks. The host boots the machine, performs diagnostics on it, and downloads code to it. These tasks are accomplished through the host interface card which connects to the host via an SBus slot, and to the J-machine using an edge connector.

The host, acting through the host interface, can inject messages directly into the message passing network and receives messages from the network. After the initial boot sequence, this is the primary method for the outside world to interact with J-machine nodes. The host interface card contains hardware FIFOs which allow sends and receives without assistance from the host. To send, the host simply fills a buffer and tells the host interface to inject the message. When messages are received, they are stored in a receive FIFO, which the host can poll at a later time. Thus the host is capable of asynchronous sends and receives, so other J-machine nodes see it as an ordinary node. The message passing interface is accessible at the Unix application level through a library. Coupled with the standard Unix networking facilities, this allows the J-machine to participate in a conventional LAN through software on the host without LAN hardware or a full TCP/IP layer.

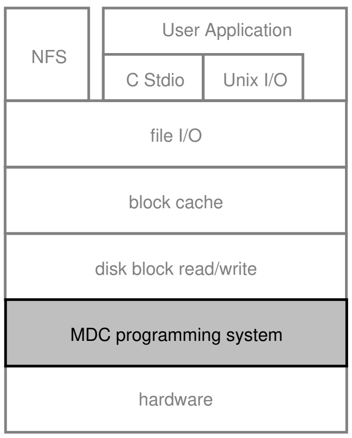
The disk subsystem is designed to interact with the J-machine solely through message passing. The disk interface consists of a set of *disk nodes* attached to the main J-machine through a set of edge connectors on one end, and attached to disks through a SCSI bus on the other. Each disk node consists of a basic J-machine node (an MDP and memory) coupled with a SCSI controller and Direct Memory Access (DMA) hardware. Multiple disks can be attached via the SCSI bus. Currently, four 402Mb disks are attached to each bus.

This design provides many levels of concurrency. Each disk can be used concurrently: unlike having one larger disk, the disk heads for each disk are not coupled so one can be performing a write while another is reading and neither operations needs to be in the same area of the disk. Exploiting this concurrency requires the use of two important SCSI features: target initiated disconnect/reselect and synchronous transfer. The SCSI protocol allows the disk (the "target" in SCSI terminology) to release the bus in the middle of transaction (the "disconnect") and regain it later (the "reselect") to complete the transaction, thus splitting the request from the response. This allows the controller to send requests to multiple disks on one bus before the first request completes, thus hiding seek times. Each disk has an on-board 64kb buffer which allows it to read and write data without data loss while disconnected from the bus. To provide opportunities to fruitfully use such split transaction, data transfers over the bus must be fast. The synchronous transfer option of the SCSI protocol allows for very fast burst transfers by removing some of the request/acknowledge overhead involved in its normal asynchronous transfer protocol. Note that both of these features are currently available using industry standard SCSI disks and controllers.

Disk nodes gain additional concurrency from the DMA hardware, which allows the SCSI controller to transfer data to and from memory without going through the CPU. The MDP only needs to set up the transfer and issue a command to the controller. It is then free to perform other tasks (such as sending messages or moving data) while the controller finishes the transfer.

Even more concurrency is gained by having multiple disk nodes. Each disk node has its own SCSI bus, so the potential bandwidth is the product of the number of disk nodes and the bandwidth of the SCSI bus. Each disk node can process requests independently, and transfer data to and from the message passing network. Having multiple disk nodes also leads to a higher bandwidth as each disk node has its own connection to the main mesh.

2.2 Message Driven C



More complete information on the Message Driven C (MDC) programming system is available in [13] and [14].

MDC makes available to the C programmer many of the hardware features of the MDP. MDC allows multiple processes per node with all processes on one MDP node sharing a global address space. MDC does not provide for address space sharing among nodes. The basic communication and synchronization mechanism is the "spawn" call, an active message based remote procedure call which creates a process on the destination (see figure 2.2). Spawn is used to create new processes within one node as well as on other nodes. This programming model maps directly to the hardware mechanisms in the MDP. Spawn is implemented as a library of C preprocessor macros that map directly to in-line assembly code producing an extremely low overhead software interface. These macros support the creation of custom versions of spawn that make sending a complex data structure convenient.

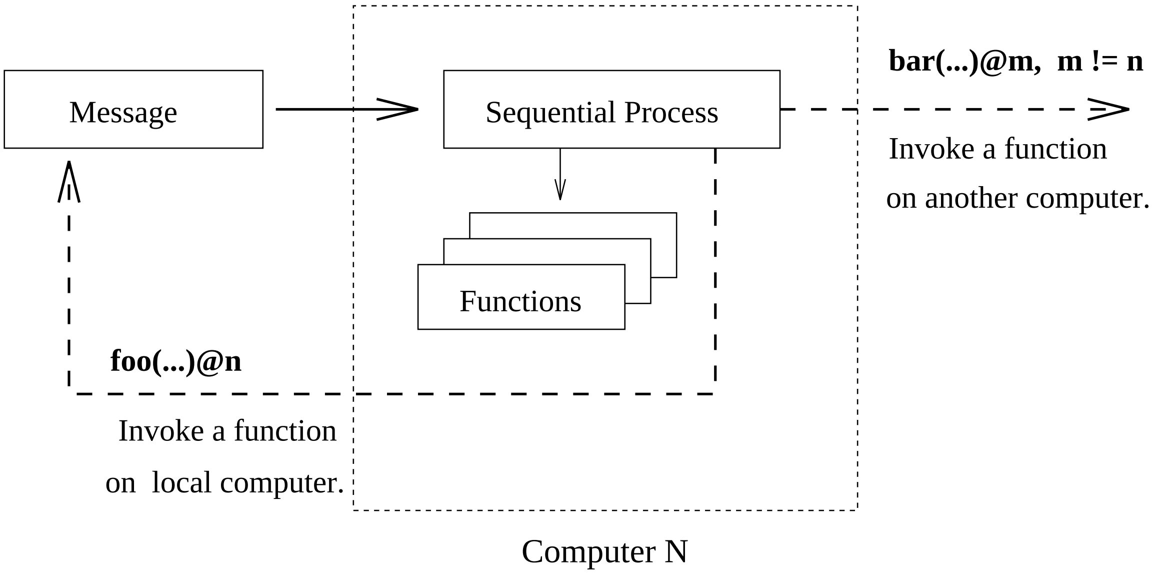


Figure 2.2: Single computer operation in MDC

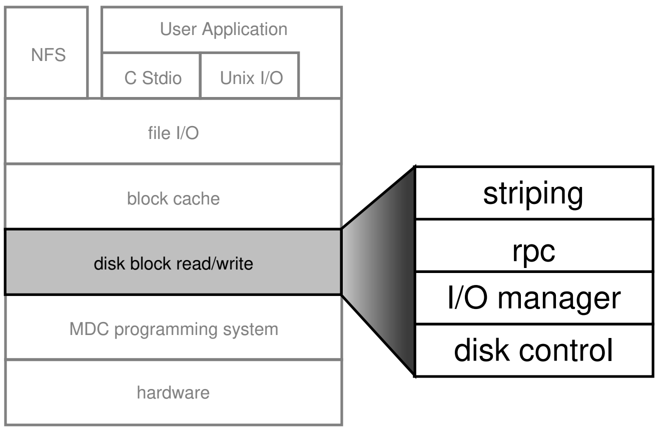
Unlike the basic hardware message passing which addresses nodes via their physical three dimensional location, MDC uses logical node numbers within logical machine partitions. This allows multiple programs to run on the J-machine simultaneously if they do not need the entire machine by splitting the machine into a number of partitions, one for each application. MDC provides the functions `node` (the logical node number of the MDP it's executed on) and `nodes` (the number of nodes in the logical partition) allowing programs to be written so that they will run without modification on different sized partitions, while still exploiting all resources available.

Spawn supports synchronization via an optional return value: the calling process continues executing until it reads the return value, at which point it waits until the value is defined. The hardware tag bits allow this to happen without explicit checks by the user program. To make this mechanism efficient, the basic J-machine model is enhanced to include the possibility of process suspension: under MDC, a process may allow other processes to execute before it has completed. The MDC runtime suspends processes when they access an as yet undefined return value by utilizing the hardware future variable mechanism. When the data arrives, the process wakes up and resumes execution. Note that the process does not suspend until it actually needs the return value. Thus, communications latency can be hidden by further computation within the same process as well as by other processes. The suspension mechanism is more efficient than a simple spin wait since it allows other computation to take place during the wait. Another benefit of this mechanism is that it allows a nested function to spawn and efficiently wait for a reply multiple times without any special code in its callers: with this mechanism, spawning a function and calling a function are interchangeable except for the use of global variables and concurrency issues.

MDC also enhances the basic J-machine model by allowing for code (but not data) to be distributed among nodes by the system. The runtime retrieves non-resident code from a remote location automatically. Unlike data, code does not change within one program execution so no coherency mechanism is needed to preserve the standard semantics. The MDC linker and runtime cooperate to allows one program to contain both code replicated on all nodes and code distributed among the nodes. This allows the programmer to trade time and space efficiency, keeping frequently called functions local on all nodes for speed while distributing infrequently needed functions to save space. Code retrieval latency can be hidden by using the process suspension mechanism to overlap communications and computation.

Beyond these basic enhancements, MDC improves the programmer productivity immensely by being a complete programming system. It supports structured programming in C and the standard C library, allowing for readable and portable code. It provides a standard C preprocessor to allow shared declarations among modules. The MDC compiler and linker support multiple compilation units and object code libraries. Additionally, the linker provides needed support for code distribution and selectable code replication. While most of these features are not new, they simplify the implementation of any program as large and complex as a file system.

2.3 Block Level I/O



Just as the MDC layer simplifies the programmers' view of basic MDP features, the block I/O layer presents a simplified interface to the disk subsystem. Rather than interacting with DMA buffers and memory mapped hardware registers spread among many different nodes, layers above the block I/O level use a procedural interface to one large disk composed of (hardware determined) fix sized blocks, grouped into *stripes*. While operating on one or more contiguous stripes produces the best performance, the procedural interface supports operations on any contiguous set of blocks.

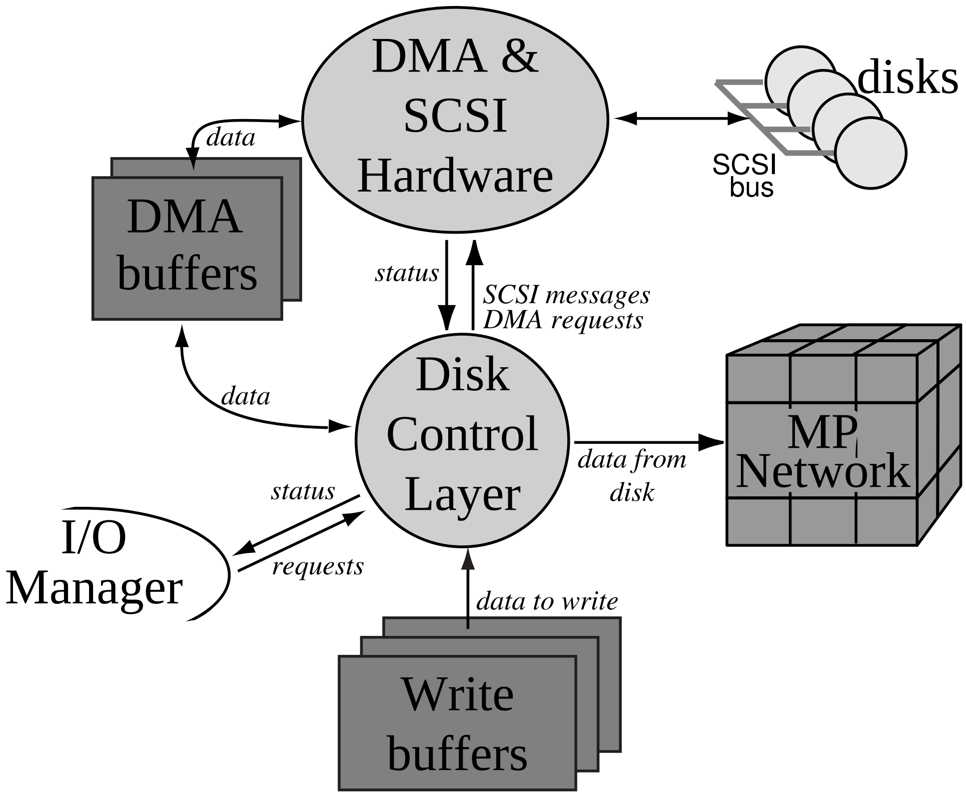


Figure 2.3: Data and control flow through the disk control layer

The block I/O layer is actually composed of four sub-layers: a disk control layer, the I/O manager layer, an RPC interface, and a striping layer.

2.3.1 The Disk Control Layer

The disk control layer implements the SCSI protocol with assistance from the SCSI controller and DMA hardware. Although the controller handles many of the low level features of SCSI, it leaves much of the higher levels to the software. The basic data and control flow is illustrated in figure 2.3. Requests flowing into this layer are translated into a series of SCSI messages and requests for DMA transfers. The layer returns status information when a request completes, an error occurs, or a request is split.

One of the more important tasks of this layer is the detection of SCSI split transactions. After receiving a request, the controller hardware on the disk itself decides how quickly it can service the request. If the data is not immediately available or the disk head is not in the appropriate position, it will send a message to the SCSI controller asking to split the transaction. The control layer will, upon receiving this

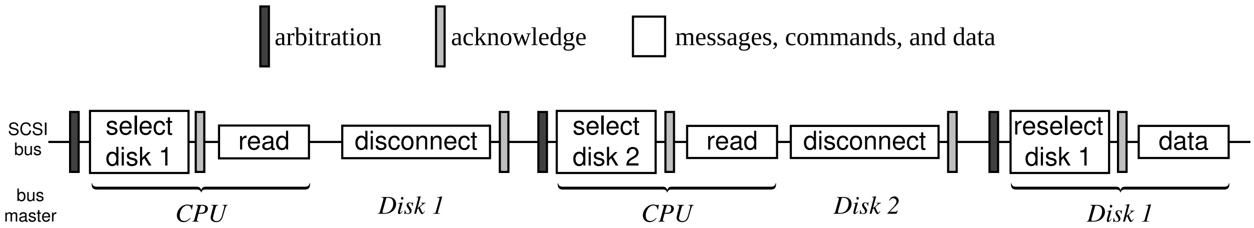


Figure 2.4: An example of SCSI bus activity during a split transaction. An operation on a second disk is initiated before the first operation completes.

message, save its state, release the SCSI bus, and return an "in progress" status. At a later time, the disk will gain control of the bus and attempt to resume the transaction by using the SCSI protocol's low level arbitration mechanism. The control layer will detect this condition and resume the request. An example of this interaction is presented in figure 2.4. A disk may choose to split a transaction many times to compensate for mismatches between SCSI bus transfer rates and physical disk transfer rates.

For most common requests, the control layer must transfer a large amount of data, a task handled most efficiently using the DMA hardware. The control layer must set up the DMA transfer, both within the DMA hardware and the SCSI controller. For disk reads, the data will be sent to a remote destination using an optimized block send routine directly from the DMA buffer. Data must be copied into a DMA buffer from an intermediate buffer in local memory for write operations because of disk timing constraints and the limitations of the DMA hardware.

To decrease latency on both the send and copy operations, these operations are overlapped with DMA by pipelining. This process is illustrated in figure 2.5. Each disk transfer is split into many DMA transfers along block boundaries. During a read operation, all DMA transfers after the first are overlapped with a send of previously read data by starting a DMA operation to retrieve the next block, sending the most

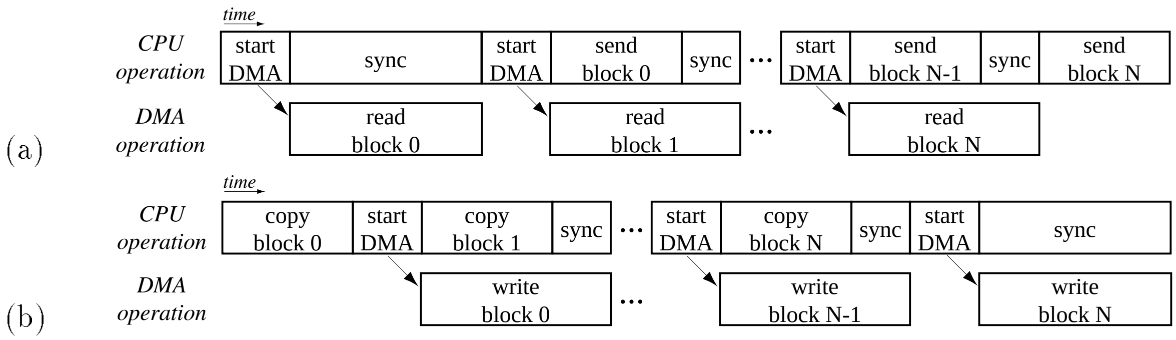


Figure 2.5: Overlap of DMA and CPU activity in (a) reads and (b) writes

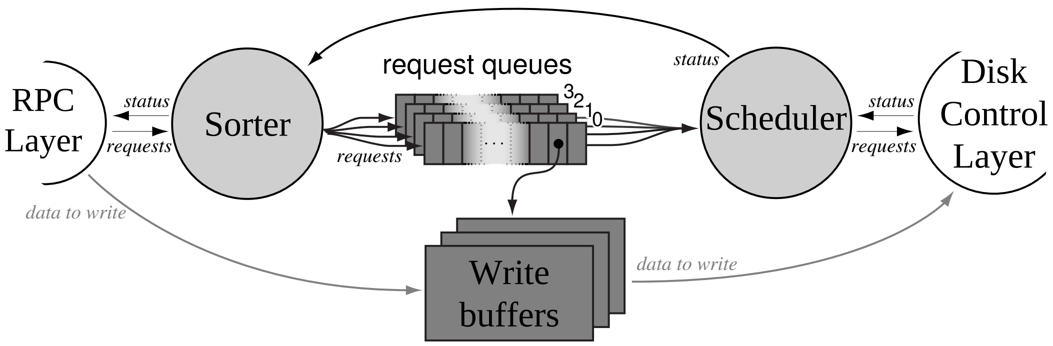


Figure 2.6: Request scheduling in the I/O Manager sublayer. Included is an example of a write request with a pointer to a write buffer.

recently read block, and then synchronizing. Multiple buffers are used to safely overlap the operations. For large reads, the inter-node transfer will overlap with the disk to memory transfer. Similarly, during a write operation, all memory-to-memory copy operations after the first are overlapped with a DMA transfer by starting a DMA operation to write the most recently copied block, copying the next block, and then synchronizing. This process overlaps transfers from memory to memory and memory to disk.

This design allows us to concurrently use the CPU resources (including the message network) and the SCSI bus. Support for SCSI split transactions allows higher levels to exploit the available inter-disk concurrency. Each request to the disk control layer corresponds to all or part of one transaction with one disk, allowing higher layers to manage this concurrency at a fine-grain level.

2.3.2 The I/O Manager

The main task of the I/O manager is to fairly control access to the SCSI bus. When multiple requests for each of the disks are outstanding, it is important to ensure that each disk is given one request at a time and that the idle times of disks with outstanding requests are minimized. However, starting a request requires having control of the SCSI bus, and disks may need access to the bus to complete requests.

This complexity is managed in a fairly straightforward way, as illustrated in figure 2.6. For each disk, the I/O manager maintains a queue in which it stores requests for that disk on a FIFO basis. As requests come in, they are placed in the appropriate queue by a sorting process. A scheduling process cycles through each of the disks which are idle, starting the request at the head of the queue for that disk. Requests are removed from the head of the queue when they complete. If a disk needs the bus to resume a transaction, it may gain it at the end of any active transaction using the SCSI arbitration process, thus ensuring fairness among disks. If the I/O manager detects outstanding messages from the message passing network, it will suspend between operations and allow other processes to execute.

This architecture allows concurrent use of the disks by building on the lower level split transaction concept. Assuming that most SCSI requests will be split, one request per disk (the hardware limit) may start before waiting for the first request to complete, thus masking disk seek times and utilizing the SCSI bus at peek bus transfer rates instead of physical disk transfer rates. Scheduling is handled on a fine grain level, one request per scheduling decision. By cycling through the queues, this architecture schedules bus access fairly among disks and thus the disks are utilized efficiently. Additional concurrency is achieved by overlapping communications over the message passing network and the SCSI bus: the MDP hardware queues incoming messages automatically while the CPU is busy. By periodically suspending, the I/O manager allows new requests to enter the request queues and/or new data to arrive while the disks prepare for transfers.

2.3.3 The Block I/O RPC Interface

The Block I/O Remote Procedure Call Interface layer is designed to allow any node on the J-machine to treat the disk nodes as a set of servers of physical disks. This layer has only a few small tasks to perform. On the client side, it just needs to send a message. On the disk node side, read messages are translated directly into requests that are placed in the I/O manager queues. Since writes must occur from local memory, additional work is required to allocate a buffer and transfer the data from the remote location. Only when all data is present does the write request enter the queue. This is handled by the RPC interface transparently. If a disk node has too many outstanding write requests buffer memory will run out. In this case, the client is asked to retry later.

This simple interface has very low overhead at the client: it just needs to send requests. Other processes on the client node can continue processing. Additionally, it supports concurrency by easily accommodating multiple overlapping requests from different nodes or different processes within one node. This interface decouples clients (which may be on any node) from servers (which must have access to physical disks) yet allows any process in the machine to conveniently access the disks. Since all needed data is present on the disk node once a request is in the queue, lower layers need only manage one nodes resources, greatly simplifying their design.

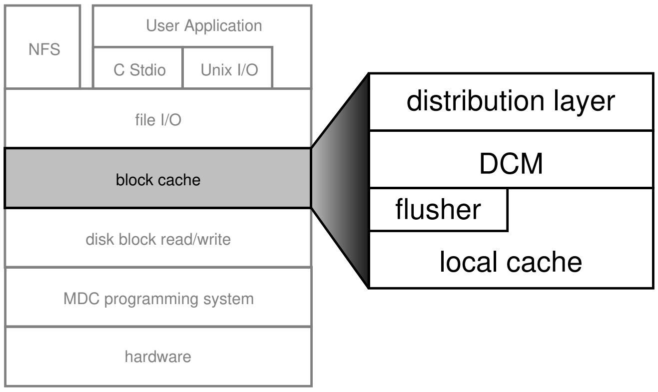
2.3.4 Disk Striping

The disk striping layer joins all the separate physical disks among all of the disk nodes into one logical disk, simplifying the view of the disk subsystem. To optimize the utilization of the disks, each disk is divided into a number of stripes. Logical blocks are translated first into logical stripes, and then blocks within those stripes. Requests spanning multiple stripes are split automatically into multiple requests, each entirely within one stripe. Each of the resulting requests is handled concurrently. Each request, now addressed to one logical stripe, is distributed among disk nodes and disks within these nodes based on a logical stripe number. Stripes are first distributed among nodes, and then among disks within nodes. Thus, a large requests for many consecutive blocks will access one stripe on one disk within each node, then a second (different) disk on each node, and so on, until all requested blocks are handled. Except for the last stripe on each disk, the stripe size used is 64kb. This corresponds to the size of the memory buffer on each disk, the optimal transfer size for the J-machine disks. The last stripe on each disk may be shorter since the capacity of a disk need not be a multiple of 64kb.

This architecture has several benefits. First, it splits requests into fine-grain tasks which are mapped directly to a particular node. This limits the complexity of lower levels by limiting operations to take place within one node. This also ensures that the task given to lower levels are sufficiently fine-grain that the scheduling mechanisms work well. Striping also enables concurrency among SCSI busses and disks by mapping each stripe to a particular disk on a particular bus, splitting requests along stripe boundaries, and executing requests concurrently. This is in addition to the more basic concurrency gained by allowing multiple processes to issue requests concurrently.

The striping mechanism used also aids in load balancing, so that the concurrency provided by higher levels can be fully exploited: striping maximizes bandwidth in a scalable manner. Two related properties of the striping algorithm ensure this. First, for sequential accesses, no disk node will receive a second request before all disk nodes receive at least one request (thus SCSI busses are load balanced) and no disk within the whole system will receive a second request before all disks in the system receive at least one request (thus balancing load among disks). Second, for random accesses there is an equal probability that any SCSI bus and any disk will receive the request, so over sufficiently many accesses, each disk and each bus will receive approximately the same number of requests. Load balancing is important to achieving maximum bandwidth in periods of high activity: the observed bandwidth of many sequential or random requests starting roughly simultaneously should be the peak combined bandwidth of the disk system (the minimum of the peak rates for all communications, the peak SCSI bus rate times the number of buses, and the peak disk transfer rate times the number of disks).

2.4 The Distributed Block Cache



The block cache layer presents the same interface to the upper layers as the block I/O layer. As such, it is not strictly necessary and was not present in early implementations of MDFS. Its main purpose is to enhance performance by reducing the load on the disk systems, optimizing transfers, and softening the "bursty" nature of checkpointing.

Caches reduce load by employing the principle of locality: most programs tend to work in nearby regions of space over short periods of time. Repeated reads of the same data can be avoided if the accesses happen closely in time, thus better dealing with "hot spots" generated by global data such as directories and the free block list. Similarly low lifetime data (from writes) is dropped as repeatedly overwritten data need not be flushed until it has stabilized. More importantly, nearby reads will be combined into one read: the first of a set of many small reads will bring in an entire cache line thus avoiding reading the disk on subsequent nearby requests. Similarly, adjacent writes can be aggregated into one large write by delaying the flush to disk and joining requests.

These same properties lead to optimized access to disks. Transfers will occur in larger units, more fully utilizing burst transfer modes. Also, the write pattern can be tuned by adjusting the flushing algorithm.

These properties are designed into the MDFS block cache. The cache is designed as a distributed cache (i.e., it is a global structure spanning many MDP nodes) to take advantage of the increased memory bandwidth provided by the fast communication rates but low memory of the fine-grain MDC environment. To ease the design of the cache, it consists of several layers: a local cache, a set of migrating flusher processes, a distributed cache management (DCM) layer, and a distribution layer.

2.4.1 The Local Cache Layer

The local cache layer manages the cache within one node. Each cache entry is an entire disk stripe. Read operations for data not already in the cache always read in the complete stripe. Write operations to stripes not in the cache present a problem when not writing a complete stripe: should the entire stripe be read before executing the write in order to fill in the missing data, or should it be assumed later writes will do this? The MDFS cache is designed to avoid a read on a partial stripe write. This is accomplished by keeping track of "valid" portions of the stripe, i.e. those that have been written or read. If it is necessary to flush an incomplete stripe (one that is not all valid), the system attempts to write a partial stripe. Read requests trigger an actual read only if it is not in the valid portion (i.e., what has been written). Also, a "write zeros" operation is provided to aid in preventing reads of newly allocated blocks: the file system can clear the entire block when it allocates a block so that subsequent reads or writes won't fetch old data from disk.

The local cache layer operates concurrently. It handles multiple requests in parallel: later requests will execute while earlier requests are waiting for responses from the disk system. In most cases, this is easily handled in MDC by simply ensuring that all functions are written in a re-entrant manner (i.e., they do not rely on global data for their operation) with an "in-use" counter to prevent replacement of cache lines on which operations have not yet completed. A "state" field in the cache line prevents concurrent operations on a single cache entry from interfering with each other: prior to reading or writing data for a cache entry, the state is set to "reading" or "writing" causing subsequent conflicting operations to suspend until the cache line returns to a normal state.

2.4.2 Flusher Processes

The cache flusher processes ensure a safety property of the cache: all new data is eventually written to disk. This property is necessary to ensure data retention through system crashes and power downs. A least recently used algorithm is used to select among dirty blocks, those whose data has been written and thus no longer mirror the contents of the disks.

Data for each disk is flushed concurrently with data for other disks. This is accomplished by having one flusher process per disk. Processor nodes participating in the cache are logically organized in a ring, with flusher processes migrating around the ring. At each processor a flusher process writes at most one dirty stripe before migrating to the next processor node. This orchestrates the write pattern so that a requests is sent to each disk having dirty stripes, but avoids flooding the disk nodes with write requests. It also load balances flusher processes among the processors with dirty cache entries.

To lower overhead, the flushing algorithm runs only when the block cache is otherwise idle. Each flusher must wait until it detects an idle condition on all nodes for some minimal time. This is accomplished using a simple algorithm. The local cache counts the number of requests received. At each node, the flusher waits until the node has been idle for a minimum time period. It then migrates to the next node in the ring. When the request count does not change at any node over three cycles through the ring, it is assumed that all nodes are idle. At that point, the flusher begins flushing blocks by migrating around the ring flushing one block per node until no blocks are left, or the block cache is not idle. This process stops when a request count on any node in the ring changes, at which point the idle detection algorithm is restarted.

Flusher processes consume very little resources. The MDP background context is used to detect the idle period, so idle detection only executes on a node when all message queues on that node are empty, a condition that occurs only when an MDP has no work that is not in transit. The only data that needs to be transmitted when processes migrate is the idle state (one integer) and a couple of parameters.

2.4.3 The Distributed Cache Management Layer

This layer is analogous to the block I/O RPC and striping layers. Its purpose is to package requests into RPC calls to make each of the local caches on each node available to all nodes, and to implement the distribution strategy specified by the upper layers. It also handles allocation and deallocation of memory for data on the client side.

The DCM layer is written in an object-oriented manner for data reuse and flexibility. By using object oriented techniques, one framework works for multiple types of cache. These same routines are used in the file I/O layer for implementing a distributed cache for meta-information (see below). Additionally, these techniques allow us to swap out distribution strategies to allow alternate implementations for our experiments.

2.4.4 Request and Data Distribution

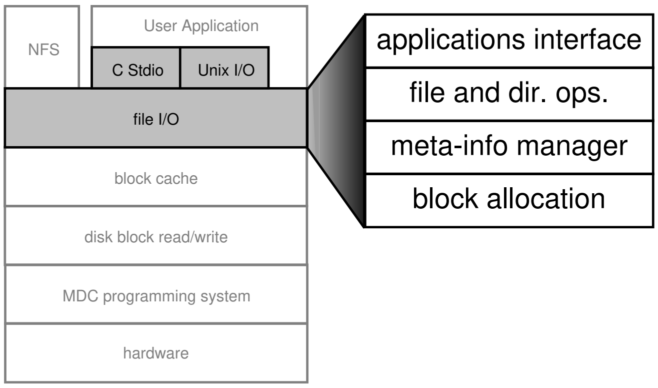
Data is distributed over all file system nodes, thus using all of the otherwise unused memory in the file system. Requests execute on the node containing the data. A request will migrate to the appropriate node via the DCM layers RPC system.

This distribution strategy is based on a hashing technique: requests contain a key which is hashed into an integer using a call-back routine unique to each type of cache. This integer is then mapped to a node number via the modulo operation, thus striping requests and data among nodes.

For the block cache, the key used is the stripe number portion of the the logical block number. This is a very low overhead technique, in that calculating the hash value of a key (and thus the location of the requested data if it is in the cache) is a simple integer operation on the block number. It also produces a roughly load balanced distribution of data (and thus workload from cache requests) for reasons very similar to those given in the discussion of striping in the block level.

This technique propagates the concurrency inherent in lower layers up to higher layers. Requests for different data can execute concurrently as they will often map to different nodes. Since lower levels can handle multiple requests concurrently, the upper levels need not restrict their calls to prevent multiple requests to one node. Requests for the same data will share disk transactions because of the design of the local caches.

2.5 File Level I/O



The file I/O layer allows application programs to treat the disk subsystem as a set of named files. Files can be created, written, read, and deleted independently of each other. Multiple files can be read or written concurrently.

To simplify this complex task, the file I/O layer is composed of a number of sub-layers. The bottom-most layer deals with disk allocation and partitioning: it divides the unified logical disk into a few partitions, and then subdivides each partition into fix sized allocation units (logical blocks). The next layer manages a number of data structures (the file meta-information) necessary to keep track of a file's attributes (such as its length and date of creation) and structure (where the data is located for a given offset into the file). Other layers provide routines to manipulate files and directories (the file/directory layer) and provide user applications access to files via RPC (the applications layer).

2.5.1 The Block Allocation Layer

The block allocation layer provides a number of routines to partition the logical disk presented by the block I/O and cache layers. This disk can be divided into a number of logical partitions, allowing different file systems to coexist on the same set of disks so that applications and file systems research may continue independently.

More importantly, each partition is further divided into blocks whose length is some multiple of the hardware block size. The size of a logical block is constant within a partition, but can vary among partitions. Logical blocks are the minimal allocation unit within the file system: files are allocated space in multiples of the logical block size. In MDFS, the logical block size is typically (but not necessarily) smaller than the stripe size.

Once a block is allocated to a file, that block can be read or written concurrently with any other blocks within that file or any other. However, the allocation algorithm itself relies on mutual exclusion for correctness and thus is effectively sequentialized.

2.5.2 The Meta-Information Management Layer

The meta-information management layer manages the data structures necessary to maintain file attributes and file layout on disk. The top level structure for this information is an "inode", similar to the corresponding data type in the Unix file system. Inodes are numbered by a (block, entry) pair which gives their location on disk. A number of file attributes, including the nominal length of the file, are stored within the inode. Additionally, inodes maintain the basic information on the structure of the file. MDFS files are structured as a tree of blocks, with data blocks at the leaves. The logical block size determines the upper bound on the number of branches within a non-leaf node. Some portion of the tree can be left unallocated allowing for sparse files. The root and depth of this tree are stored in the inode entry for the file. The meta-information management layer handles the tasks of creating and deleting inode entries, changing attributes within these entries, mapping offsets within a file to blocks on disk, and allocating new blocks to fill out the tree as needed.

Most of the necessary information for managing meta-information is cached within the layer to reduce disk accesses. MDFS uses a stateless call paradigm to improve opportunities for concurrency, but this architecture necessitates re-reading much of the meta-information for a file on every call. The meta-information cache prevents this from being a burden on the lower levels: caching this information reduces contention for inode blocks in the block cache. Also, the data can be re-distributed to fit the needs of the meta-information management layer decreasing communication: meta-information can be kept on the node which is most likely to need it rather than the node maintaining the block cache entries for the blocks in which it is stored on disk.

The meta-information cache actually consists of two independent caches: one for partition-wide information which doesn't change often, and one for file level information. The partition information cache is local and replicated in each node since the few entries in it take up little memory and change infrequently.

The file information cache is a distributed cache using the same design as the block cache: it is further layered into a local cache, a DCM layer, and a request/data distribution layer. The local cache can handle information for multiple files concurrently. The DCM layer is the same as the DCM layer in the block cache. The same hashing-based request/data distribution scheme is used, with the hash key derived from inode location information, i.e the partition id, inode block number, and entry number within the block. In addition to the information contained in the inode, the cache contains the block numbers around the most recently accessed section of the file, along with the locations of all the index blocks required to get to that section.

Since the meta-information cache reuses the distributed cache management layer from the block cache, it gains many of the advantages of the block cache. The local cache can execute multiple concurrent requests. When coupled with the distributed cache management layer, many requests may execute concurrently among processors in the system and may overlap execution within a node using suspension. Requests execute on the node containing the data to lessen communication overhead. Mapping is based on hashing the inode location, a low overhead technique which just needs a few arithmetic and bitwise operations. Since different files have different inodes, this tends to distribute handling of files among all processors in the machine, leading to good load balancing behavior. This latter effect is enhanced by checkpointing: during checkpointing behavior, the files are all created at approximately the same time so they will be allocated by adjacent inodes because of the simple "next available space" algorithm used to allocate inodes. The hashing-based mapping strategy converts this spatial locality into an even load distribution.

2.5.3 File and Directory Operations

The file and directory operations are written in terms of the meta-information management layer. The directory operations provide for mapping human readable names to inodes. This layer creates a file by creating an inode and inserting a map entry from the desired file name to that inode. Similarly, it deletes a file by removing its entry and deleting the corresponding inode. A lookup operation returns a "handle" to the file (really just the inode location) which allows read and write operations on it. The read and write operations are all written in terms of the file meta-information layer. Thus they are fairly simple.

All read, write, and lookup requests within the layer are handled concurrently. However, directory operations require mutual exclusion on writes for safety. To partially alleviate this problem, operations only lock one section of the directory at a time, thus pipelining create and remove operations.

2.5.4 The Applications Interface

The applications interface provides user applications access to the file system. It consists mainly of an RPC interface to the the file and directory operations layer. This RPC interface decouples the application from the file system, allowing the system to change without changing the application. This takes the place of the traditional system call mechanism used for this purpose in sequential operating systems such as Unix and MS-DOS. Under MDC, the file system is not part of the kernel: it is an "application" with some very unique capabilities.

The RPC interface has very little overhead in the client applications. Only a small amount of state information (file handles for the open files) needs to be stored in the application, although some buffering reduces call overhead for small requests. Code overhead is minimal: the application only needs to link in small routines to pack requests and unpack responses.

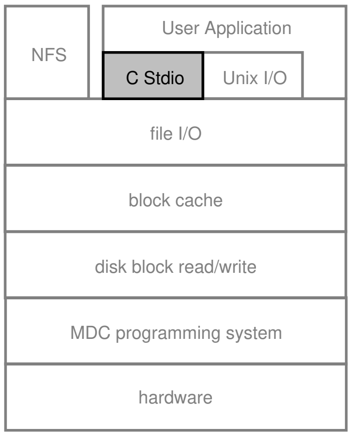
The RPC interface has another benefit: it eases the NFS implementation. Since the file system interface it based on an RPC interface implemented over the native message passing system, any node that can exchange messages with the file system nodes can make file system requests. Since the host is such a node, it can simply send requests to the file system for its NFS implementation.

The other major task for the applications interface is request distribution. Since there are many file system nodes in a system, the applications interface must choose one to send a request to. This choice is made based on a hashing scheme similar to the one used in the caches. The hash key is based on the file name for directory operations, and the file handle for read and write operations. Again, a hashing-based request distribution scheme has little overhead: only a few arithmetic operations to compute the hash key. For directory operations, the overhead is slightly greater because the hash is based on a string, so the time to compute is based on the length of the string. Since these operations are rare compared to read and write operations, this is not very important. Also, as in the same scheme for the caches, hashing leads to a first order load balancing among file system nodes. This is important, in that if all requests during a checkpoint were to go to one node, the volume of requests would quickly overwhelm it. In addition to the speed penalty associated with having only one node handle requests, having many outstanding requests may exhaust the memory of a single processor node.

By carefully choosing the hash key and hashing method, we can improve the locality of operations and thus the speed of the system. Since the file handle is just an opaque representation of the file inode location information, we choose the same hashing scheme as in the meta-file information cache. Thus only local meta-information cache requests are necessary to implement read and write. However, for directory operations, such requests may be remote.

Like previous RPC interface layers, the applications interface layer executes concurrently and enables upper layers to execute concurrently. It can handle multiple outstanding requests. Thus, many processes, whether on one node or distributed among several nodes, can each make a request to the file system at approximately the same time. Since calls into the RPC layer are stateless, the application could issue multiple requests without interference between requests, allowing parallel execution among many processes within one application.

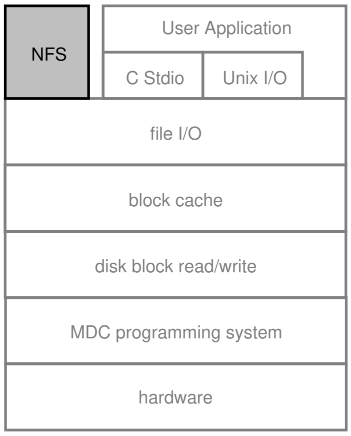
2.6 The C Stdio Interface



The C stdio interface provides a portable interface to the file system by implementing the ANSI C Standard I/O Library. An application may choose to interface to the file system either through this interface or directly through the applications interface. While presenting a portable interface, the C interface limits concurrency because of the semantics of operations on its file pointers. The semantics of this interface specify that a file pointer does not update its location if an error occurs. Since success or failure of an operation is not easily predictable, file pointers update only after an operation completes thus sequentializing file operations. However, the concurrency provided by having multiple processes access different files is still available as long as each process maintains its own file pointer (i.e., they do not share a global location for this pointer). Another drawback to the C interface is that its buffering mechanism adds memory overhead: if the data is read or written in large chunks, as would be the case in writing an entire array of data to disk with one call, the buffer allocated by the library remains unused.

The C stdio interface is implemented entirely as a library written on top of the MDFS applications interface by porting the GNU C standard I/O routines. These routines implement the C I/O library in terms of a few basic routines (`stdio_open`, `stdio_read`, `stdio_write`, and `stdio_close`) which were easily implemented in terms of the file systems applications interface. The ease of this task was mainly due to the structure of the GNU library which was designed specifically to simplify porting to new operating systems. Other file interfaces, such as the Unix I/O calls or the forthcoming MPI I/O standard, could be added to MDFS just as easily given a good reference implementation.

2.7 The Network File System



The NFS layer provides the LAN connectivity for MDFS. Any workstation on the LAN can mount a J-machine's file system directly using NFS. The user can then access this file system using standard commands. As long as the relevant formats are binary compatible (using only 32 bit integers, 64 bit IEEE floating point numbers) simply copying a file into the appropriate directory makes it available to J-machine applications. Files on the J-machine can be examined using standard tools, either by accessing them directly or copying them to local disk. The former is generally more convenient, while the latter is useful for backup purposes and is more efficient if the file is to be read multiple times. Utilities are provided for converting pure text files to and from the J-machine's native format (a matter of inserting or deleting the extra zero bytes needed because the J-machine uses only word addressing.)

The NFS layer is treated as if it were just another application by the Unix kernel and the other layers of MDFS. It executes on the host as a user level process under Unix, accepting RPC requests via the standard networking libraries. A standard SBus driver provides all the hardware access necessary, so no special kernel code is needed. For its MDFS related operations, the NFS layer simply sends messages to the MDFS applications interface just as an MDC application would. A messaging library and a version of the client side of the interface recompiled for a Sun host simplify this task.

The NFS layer has a fairly simple task. It translates NFS requests into messages that follow MDFS applications RPC interface. Similarly, it must translate the responses back into NFS return values. Since both the MDFS and NFS are based on stateless RPC calls this is a relatively straightforward process. The applications interface was specifically chosen so that most NFS requests map directly to one or two MDFS requests, making this translation even easier.

To ensure that multiple NFS requests can take advantage of the concurrency in the file system, the NFS layer uses a multi-threaded implementation based on the active messages paradigm. Each NFS RPC request triggers the creation of thread within the system. These threads suspend when waiting for responses for the file system, allowing other requests to continue. Responses from the J-machine resume execution of the suspended thread. When all necessary information is received, the NFS layer responds to the NFS RPC request and destroys the thread for it. By using this technique, the NFS layer can have multiple outstanding requests within one Unix process. It begins processing an NFS request as soon as it arrives, allowing the underlying file system to handle all requests concurrently. Multiple Unix processes can take advantage of the concurrency in MDFS, but a single Unix process typically has only one thread of execution and uses synchronous file system calls, so to see this concurrency requires many processes (either on the same or different workstations) to issue commands simultaneously.

Chapter 3

J-Machine Implementation Experiences

3.1 Node Partitioning

Except for the association of the low level block I/O with disk nodes, there has been no mention of the mapping of file system layers to physical processors. Ideally, all processes not needing direct access to disk hardware could be anywhere in the machine. However, the small memory in each processor node and lack of a memory management unit in the MDP architecture means that it is not possible for MDC to execute multiple applications on one node. As a result, we divide the J-machine into disk nodes, file system nodes, and application nodes at boot time using the MDC loader's ability to logically partition the machine among different applications. Each set of nodes runs a particular set of layers in the file system. Figures 3.1 and 3.2 illustrate this arrangement. The disk nodes consist of all processor nodes with attached disks. Most of the block I/O layer runs on the disk nodes. The file system nodes form a slice of the machine adjacent to the disk nodes. The middle layers up to the client side of the applications interface reside on the file system nodes. The NFS layer runs solely on the host, with the client side of the application layer linked into it. The remainder of the machine is left unallocated at boot and can be used by any application. The client side applications interface and the C stdio library link into the user application and reside on the application's nodes when an application is run. Multiple applications can run in the applications nodes by using MDC to further partition the machine.

This approach allows us to reserve resources for a specific task. Thus, if the file system has inadequate resources, we can simply increase the size of its logical partition, giving it more memory and processors.

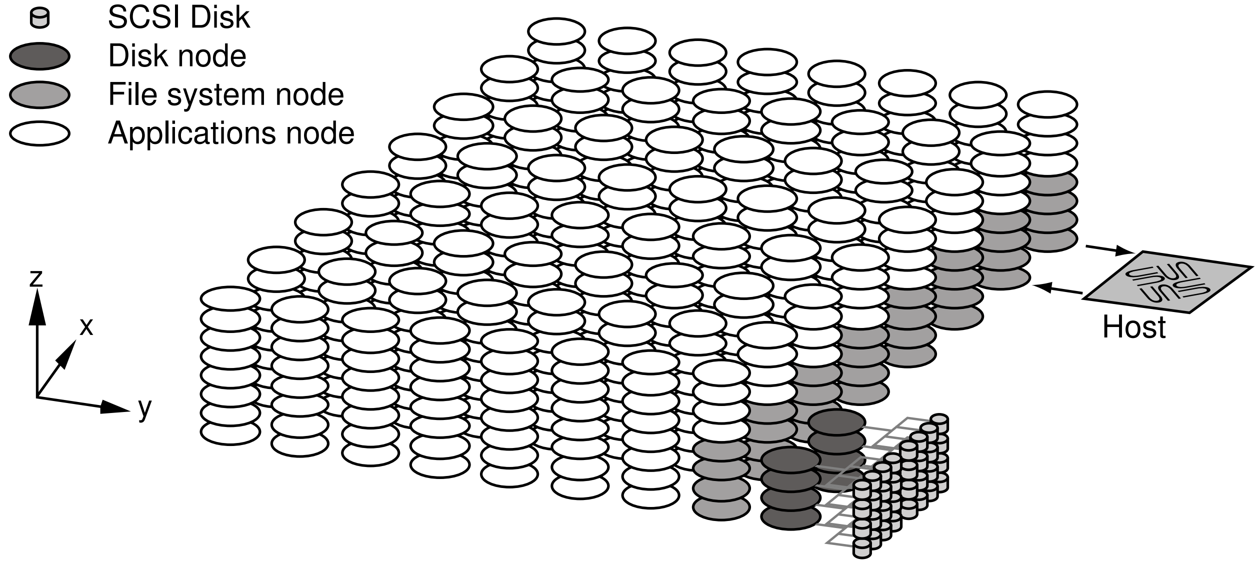


Figure 3.1: Node partitioning in a 512 node J-machine with 32 disks.

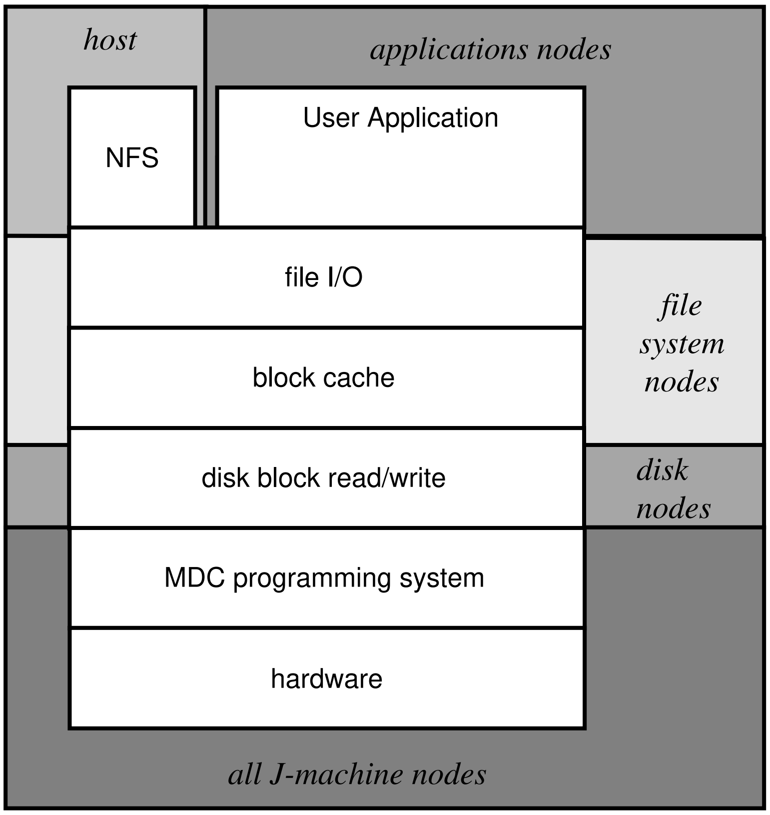


Figure 3.2: Distribution of layers among node partitions.

A minor difficulty with this approach is that MDFS needs to circumvent MDC's default mapping of node numbers to physical nodes to achieve inter-application communications. MDC only maps nodes within one partition into its node numbering sequence for that partition. Thus, to access nodes outside the current partition, the client side of the applications interface and the block I/O interface need to use an alternate map which includes the nodes they need to access. This is accomplished by modifying some of the macros used by the MDC spawn macro library to allow the use of an alternative mapping of node addresses within client side interfaces.

3.2 Message Forwarding

One of the premises of our system is that all nodes should be accessible from any other processor node. This is not the case in the J-machine. To understand why, recall that the J-machine disk nodes are only connected to the rest of the system via Y edge connectors. This means that there is no channel connecting them in the Z direction to any other node. Additionally, since there are only two disk nodes per physical board, each disk node is only connected to the other node on its board in the X direction. Given a sophisticated routing system, this would not be a problem as there exists a path from any node to any other. However, the J-machine employs a deterministic routing technique: each message first travels along the X direction, then the Y direction, and finally the Z direction. This means that a node on the bottom board can not send a message directly to a disk node connected to the next board up, because that disk node has no Z connection to the disk board below it. Similarly, a disk node can not send a message to any node whose position along the X axis is not 0 or 1. This problem is only relevant for communications between file system and disk nodes: the file system and applications nodes are in the main J-machine board stack which has all possible X, Y, and Z connections available.

To solve this problem, the MDFS kernel was augmented to implement *forwarding*. When a node needs to communicate with another node to which it can not send directly, it computes the address of an accessible intermediate node (the *forwarder*) which can send to the ultimate destination. Instead of sending directly to the desired node, it sends the message to the forwarder with a header indicating the message destination. The forwarder then resends the message (minus the header) to the true destination. Note that since the X,Y,Z pattern of the J-machine's routing may choose different paths for different directions, some candidates for forwarders are not suitable as forwarders in the return direction. In practice, this situation is avoided by appropriate choice of a forwarding node. Also note that any message can be delivered by passing through at most one forwarder due to the pattern of partial connections within the J-machine and the deterministic routing scheme.

Forwarding is handled at a very low level by adding a message handler to the MDC kernel. This message handler simply re-sends a part of itself to the physical node given by its first argument. By taking advantage of the high priority queue and the MDP's ability to dispatch a message before the entire message arrives, this process increases latency only minimally and does not decrease bandwidth. However, a forwarding node pays a small price in computational time, so we limit forwarders to be within the file system partition to avoid impact on applications.

3.3 NFS

The host interface in the J-machine was the key to the NFS strategy chosen. It simplified the network implementation by allowing messages to enter the network directly from the host. The alternative would have required Ethernet hardware and a full TCP/IP design from the Ethernet hardware interface up to Sun RPC. However, the given hardware design did present some difficulties: to send and receive messages, a process needed exclusive access to the hardware. Since both NFS and the MDC loader needed to use the facility this was not ideal. Also, Unix allowed us to block until until the arrival of either LAN messages or J-machine messages, but not both without resorting to polling. These problems were easily solved by sending all messages through an intermediate process. This process has exclusive access to the hardware, so it ensures that message sends and receives are implemented correctly. It uses a combination of polling and the Unix select mechanism to receive messages through both the J-machine hardware and Unix sockets. By using a socket interface, the Unix select call can be used to wait on J-machine messages, LAN messages, or both without polling within NFS and the loader, so such polling was restricted to one process where it can be carefully controlled.

Another minor difficulty was the complexity of Sun's NFS and RPC. To decrease development time, we chose to use Sun's RPCgen compiler and RPC library. RPCgen automated the tedious work of implementing packing and unpacking routines for the various NFS data types and call sequences. RPCgen also produced the necessary code to register the NFS protocol, initialize its RPC interface, and dispatch request handlers.

A drawback of the Sun RPC library is that it is poorly suited for multi-threaded applications: it stores return information in globals within the library preventing context switches within a RPC request handler. Since we implemented our own simple threads package based on active messages, this did not prove to be a major obstacle. Two routines were added to the RPC library to save and restore the appropriate context information needed to implement RPC return messages. When creating a context for a new NFS request, the threads package calls the first of these routines to save all needed information in the context. This information is restored prior to an RPC return.

Chapter 4

Evaluation

4.1 Performance

In terms of performance, the goals of MDFS were modest. Emphasis was placed on designing implementation techniques. As such, algorithms were chosen under a "good enough" criteria: preference was given to simple algorithms with adequate performance rather than difficult to implement algorithms that have optimal or nearly optimal performance. The performance results should be evaluated in terms of the scalability demonstrated rather than absolute numbers.

4.2 Hardware and MDC Performance Limits

Hardware limits on MDFS performance can be classified into three categories: disk bandwidth, interprocessor communications, and CPU speed. The disk drives used have average seek time of 12.6 milliseconds [10]. Given the 64k stripes used, random disk seeks should limit transfer rate to approximately 5Mb/s when transferring whole stripes. Sustained transfer rates through the internal disk controller vary from 1Mb/s to 1.6Mb/s depending on the location accessed on disk, and maximum SCSI bus transfer rates are quoted at 10Mb/s. The SCSI controller used in the disk interface cards can achieve a maximum transfer rate of 5Mb/s [15]. Thus sustained disk transfer should peak at 4Mb/s if all disks can be kept busy.

Due to the architecture of the MDP, interprocessor communications are actually limited by memory bandwidth. The communications network is designed to send at the maximum rate instructions can be issued: one send of two words of data per cycle, with each word consisting of four tag bits and thirty-two data bits. However, the data sent either must be loaded from memory or generated via arithmetic operations. In the case of disk I/O, data will come from a disk buffer. An access to memory takes 5-8 cycles so we expect a peek transfer rate of about 2Mb/s out of a 20MHz[1] MDP. Thus, we can see that with no processing time in MDFS, we would at best see a 2Mb/s transfer rate.

The impact of CPU speed on MDFS is hard to quantify beyond the memory access rate. The MDP is not a fast processor by modern standards: it was designed over three years ago using a commodity fabrication process. Thus, compared to modern chips, it runs at a low clock rate and does not have enough registers or on-chip cache. MDC increases these liabilities somewhat by attempting to support multiple processes without a memory management unit: MDC adopts a heap-based rather than stack based architecture[13], thus making the implementation of dynamic memory allocation very important. Even with these deficiencies, we would still expect that a highly tuned SCSI implementation should achieve at least 50% of peek bandwidth.

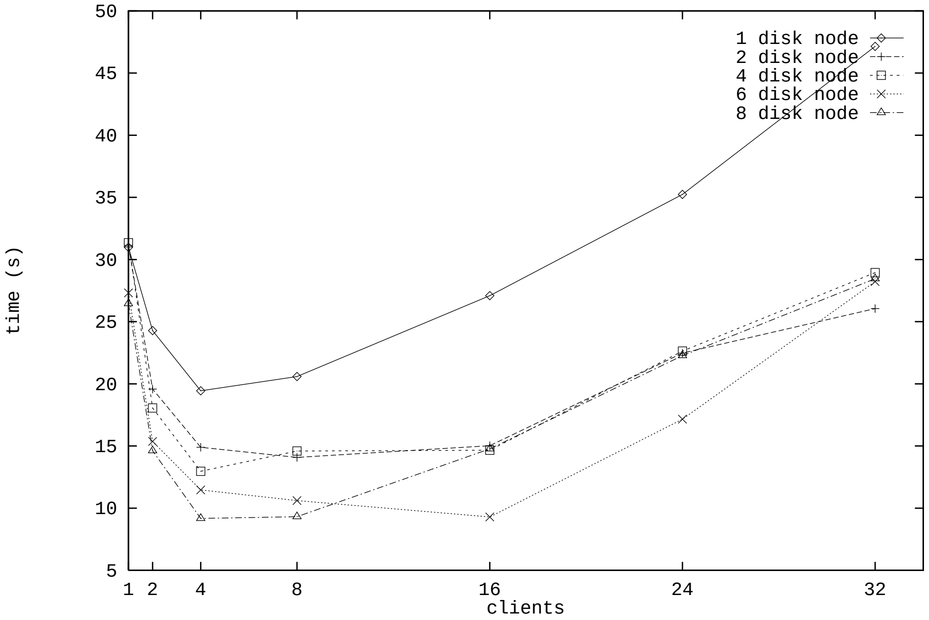
4.3 Block I/O performance results

Performance was measured using a 512-node J-machine running at 20MHz. Figures are given for a test program which was designed specifically for the purpose of measuring read/write performance. The number of "clients" is the number of nodes generating requests and data. Accesses are distributed among these nodes roughly equally. The number of "disk nodes" given is the number of MDP with attached disks. This number ranges from one to eight disk nodes (half to four disk boards). Times given are for writing 4Mb of data. Read performance is similar.

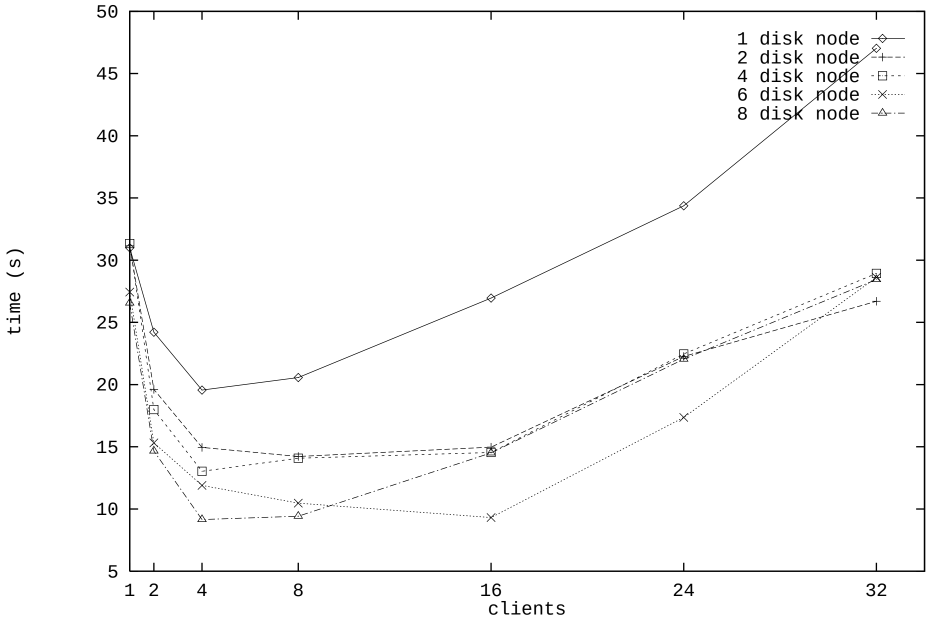
Figure 4.1 shows the performance of block I/O against request load. The data is divided into 64kb chunks. Each chunk is either written to a consecutive set of blocks distributed among processors in a round-robin fashion or written to a random group of blocks. Notice I/O performance increases until the disk system is saturated. After that point, extra load no longer fills unused bandwidth but instead delays processing of existing requests by consuming CPU resources. More work needs to be done to limit requests sent to avoid overloading the disk nodes. For some unknown reason, six disk nodes are more efficient than eight on large workloads. This is likely due to network utilization patterns, but it is unclear. Also notice that sequential access is no more efficient than random access. This indicates that seek time is not a limiting factor on I/O bandwidth.

Figure 4.2a shows the time under optimal load for different numbers of disk nodes

[1]The MDP divides its external clock by two so it is actually running at 10MHz internally



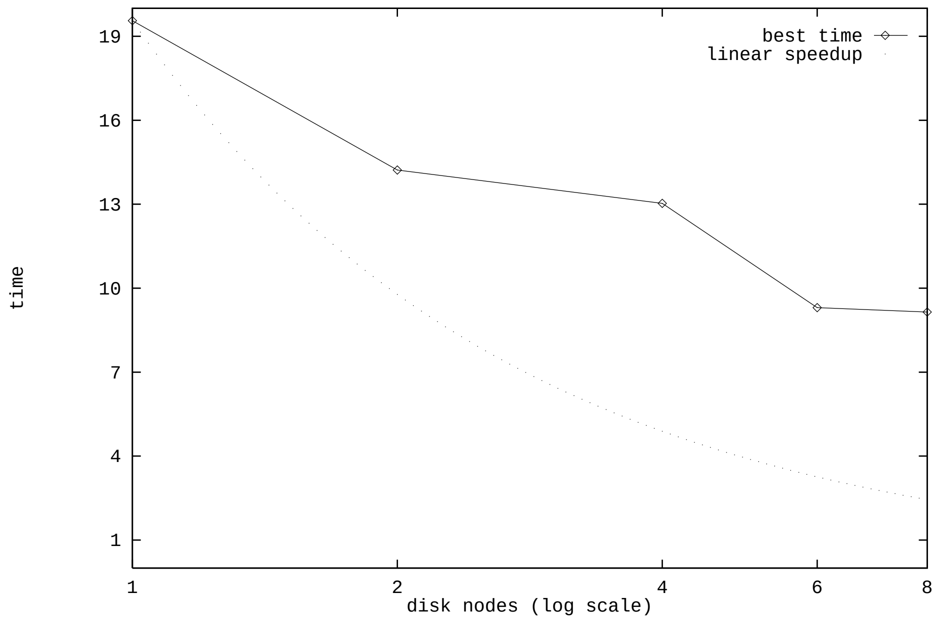
(a)



(b)

Figure 4.1: The affect of load on run-time in the block I/O layer with (a) a round-robin distribution of consecutive blocks and (b) a random distribution

(a)



(b)

disk nodes	kb/s
1	209
2	287
4	314
6	440
8	447

Figure 4.2: Block I/O performance at optimal load for different numbers of disk nodes.

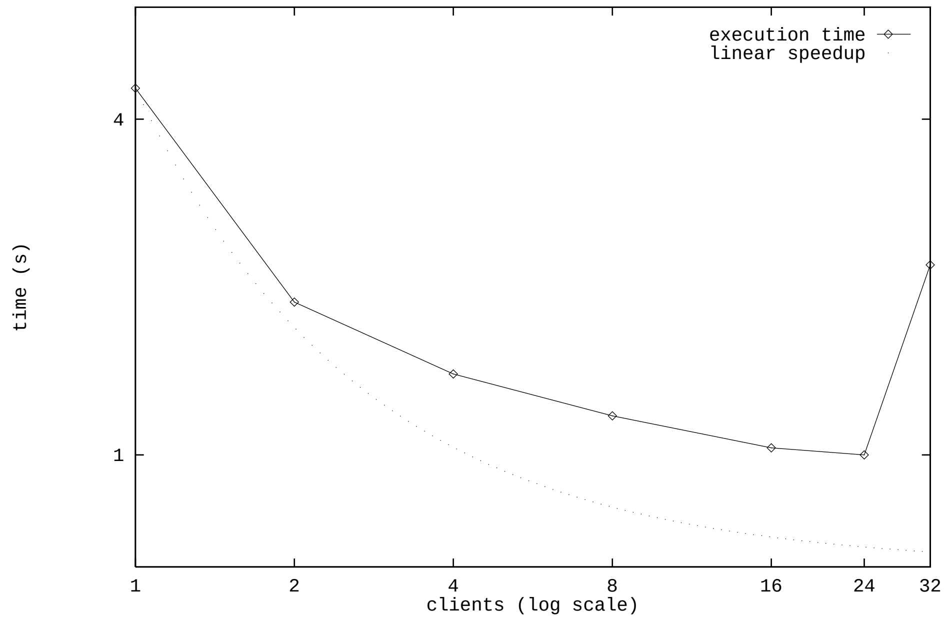


Figure 4.3: Performance of 32 node block cache on repeated writes of same cache line for various client loads.

on a linear-log scale. Also shown is the line of linear speedup. Although it does scale, there seems to be significant overhead involved in adding more disk nodes. If performance on high workloads for six and eight boards could be improved, then the speedup curve would show more improvement. Figure 4.2b gives the performance of the disk I/O system in kb/s.

4.4 I/O performance with the distributed block cache.

Figure 4.3 shows the scaling characteristics of the block cache decoupled from the disk system. Each client writes 64kb of data repeatedly to the same cache line, thus avoiding disk flushes. Different clients write to different cache lines to avoid data dependencies. As can be seen from the graph, the block cache scales well up to 24 client nodes. Somewhere between 24 and 32 nodes, blocks begin to map to the same processor node. This behavior is similar to collisions in hash tables.

Figure 4.4 shows performance of the block cache and disk I/O subsystem in various configurations on round-robin writes to consecutive blocks. Performance generally improves with increasing numbers of nodes and clients until either disk or

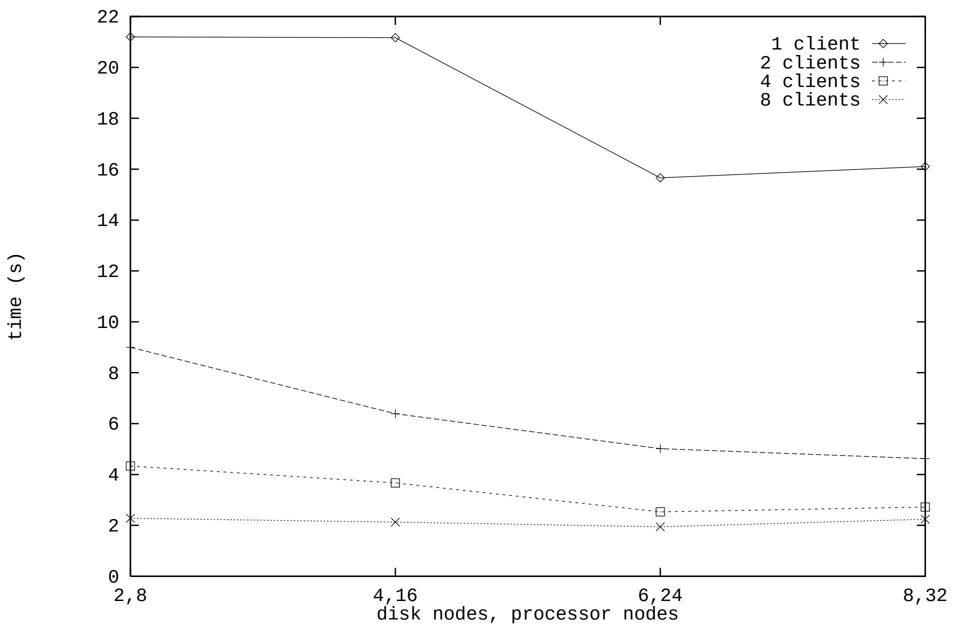


Figure 4.4: Block cache performance for round-robin block writes. The number of processor nodes participating in the block cache and the number of disk nodes are given along the horizontal axis with time along the vertical axis. Curves are given for 1 to 8 client processes.

clients	time
1	2.421
2	2.579
4	2.547

Figure 4.5: File system read/write performance writing eight 64k files in 2k chunks.

network bandwidth is saturated.

4.5 File system I/O performance

Figure 4.5 shows times for writing files. The performance of the file system does not scale for this test. This is mainly due to the sequential nature of the block allocation and directory search routines: the allocation routines need exclusive access to the free list while manipulating it, and the directory search routines compete for the same shared directory blocks. Alternative allocation routines are needed, as attempting to parallelize conventional ones proved inadequate. For directory operations, a change to a hash based directory would improve matters some. Further improvements might be made by attempting to handle multiple create and lookup operations at once. This could be done by scanning and changing the directory once for each burst of create and/or lookup operations instead of once for each operation. Thus less time would be spent transferring data to and from the block cache during directory operations.

4.6 Implementation Experiences

MDFS provided a good framework for addressing most of the issues involved in writing a file system in a scalable manner. Layering and the use of fine-grain parallelism provide for a clean, scalable, structured design. However, the design leaves unaddressed the issue of resource allocation. Fair, scalable allocation techniques are needed for memory and disk allocation. The conventional techniques used were not appropriate for a parallel design where multiple processes share one processor's resources. For example, memory allocation proved difficult because the space available depended on the number of processes running. There was no convenient way to block until adequate memory was available. Similarly, block cache requests needed explicit in-use flags to prevent other processes from changing data. There is often a tradeoff between the simplicity of enforcing atomic operation and the efficiency gained through interleaving multiple processes. The later is especially important for hiding the latency of disk I/O within the cache, but complicated the management of global data structures needed to avoid repeated requests for the same blocks.

Chapter 5

Related Research

Due to the increasing mismatch between CPU and disk throughput, much research has focussed on increasing I/O performance. Approaches to this problem vary widely, from focusing on improving single disk and disk array performance to improving the applications interface to better reflect parallelism.

Early research focused on scheduling accesses to disk to minimize seek time. Often, this is accomplished by queuing requests as they come in and then choosing a request from this queue based on some optimality criterion, such as the request for a block nearest the current position. One of the earliest method employed was the shortest-seek-time-first (SSTF) algorithm, where seek time is considered proportional to the distance from the current head position. This algorithm may lead to starvation of some requests: request from the current position may never be serviced because of more requests for nearby blocks are generated. To avoid this problem, alternatives modify the algorithm by servicing seeks in phases of inward movement and outward movement (the SCAN or LOOK algorithms), or by grouping requests into blocks and using SSTF only within a block (BSSTF algorithm). A more extensive discussion of disk-arm scheduling algorithms can be found in most good operating systems texts (for example [7]). MDFS does not use these techniques since the request queues for each disk are relatively short. For disk arm scheduling to be effective, a significant number of requests for one disk must accumulate during the processing of a request for that disk. Since requests in a parallel file system such as MDFS are distributed among several disk and computational nodes, request queues remain small. This reduces the opportunities to improve performance by reducing seeks times, the main purpose of disk arm scheduling. Additionally, the SCSI features which allow overlapping requests to disk make seek time less of an issue.

More recently, efforts have been made to increase disk bandwidth by exploiting parallelism in arrays of disks. The basic idea, called striping, involves distributing requests among disks so that multiple requests and multiple portions of large requests are serviced simultaneously [2]. MDFS uses this technique to improve block I/O bandwidth. Gibson [8] noted that one could exploit striping in disk arrays to improve reliability by writing redundant stripes to disks. The resulting array is known as a Redundant Array of Inexpensive Disks (RAID). He presents various methods for accomplishing this, each with its own performance, cost, and reliability trade-offs and labels each with a RAID level. An updated version of this discussion is present in [2]. RAID techniques increase space requirements to improve reliability. Additionally, some RAID methods can exploit redundancy to improve disk bandwidth for reads. However, without hardware support, RAID techniques have lower write performance than simple striping. Because of the intended use of MDFS as a temporary store, reliability due to disk degradation was not a significant consideration. Thus, redundancy for reliability is not present in the current MDFS design, but it could easily be incorporated by introducing a new layer either in software above the block I/O (like in [1]) or cache layers, or in hardware by replacing each disk with a RAID array and a faster driver implementation. The chief difficulty in a software implementation would be minimizing the impact of generating redundancy codes on write performance, both by optimizing communications and processor use. Additionally, atomicity in write operations may prove to be a problem since under a RAID approach the file system would need to ensure consistency of the redundant blocks distributed among disks.

At higher levels of implementation, much work has been devoted to alternative interfaces to the file system to improve efficiency. This work is based on the assumption that Unix and/or C Standard Library file models no longer match the hardware models adequately, especially in the case of parallel computing. These attempts range from simple extensions to the Unix model (as in [6]) to complete reworking of the system model, as in ELFS [9], the Alloc Stream Facility [12], and the forthcoming MPI-IO standard[3]. ELFS replaces the Unix's untyped byte stream files with typed object based files. The intent is for file objects to exploit their knowledge of internal structure and data layout: typed objects can selectively apply techniques such as prefetching, parallel asynchronous file access, and caching to improve performance as well as allowing the user to provide "oracles" which predict access patterns. A distributed object, such as the two or three dimensional grid found in typical scientific codes, can optimize writing itself to disk by taking advantage of its knowledge of the distribution scheme. A different approach is taken by the Alloc Streams Facility: it attempts to minimize copying by exposing buffering within the operating system through an interface modeled on dynamic memory allocation. Support for parallelism is improved because of better control on atomicity. MPI-IO attempts to extend the Message Passing Interface (MPI) standard's data types to support file access. Parallelism is supported explicitly by mapping in-core data layout to file data layout, allowing collective operations to efficiently transfer data to and from disk. All of these alternative interfaces require substantial work to port existing applications. Except for the Alloc Stream Facility, none provide a good backward compatibility layer that actually improves performance. In MDFS, we attempt to use already defined interfaces to minimize the changes required to port applications. This makes a variety of applications immediately available.

Many of the issues confronting implementation of a parallel file system are also relevant to distributed file systems. For instance, the caching issues we confronted are also present in the Andrew File System [11], Sprite [16], and LOCUS [17]. Most of this work assumes workstation-like nodes with large memories, slow communications, and local disks. Thus, many of the choices are not appropriate to a fine-grain environment like the J-machine. For instance, in LOCUS, files are cached both at the "storage site" (a server) and the "using site" (a client). Requests are file (not block) based so that a file can move. On the J-machine, two separate buffer caches like this would consume too much memory resources. Furthermore, communications is fast enough that sharing a single cache is practical. Also, caching must be done at the block level since file replication requires local disk. Some work has been done on distributed caches, similar to the one in MDFS, but in a network of workstations framework. Dahlin, et. al. [4] assume a fast communications substrate and simulate various distributed cache algorithms using synthetic workloads. Such simulations prove the value of a distributed cache over strictly local caches when communication is fast. As the main focus of the MDFS work is implementation techniques, we did not explore many alternative cache strategies.

Our choice of NFS as a network protocol was largely based on its availability on all of our workstation platforms. There has been recent work on improving the NFS protocol for greater client cache efficiency, file lengths of greater than two gigabytes, and better support for non-Unix file systems[19]. The NFS layer of MDFS could be extended to support this interface, or replaced with an alternate interface (such as AFS[11]) as needed.

Chapter 6

Conclusions

This work demonstrates that it is possible to construct scalable fine-grain parallel code in a structured, easily understood manner by using the techniques described. The division into small, well defined layers coupled with remote procedure calls for communication and synchronization leads naturally to a well structured system. When sufficiently well-defined and small in scope, layering produces code that is inherently fine-grain. The resulting code maps well to the J-machine by exploiting its hardware support for fine-grain concurrent execution with active message based communications and synchronization. When data dependencies are avoided, the resulting code scales well. Hashing based request and data distribution proved to be important as a low-overhead technique for parallelizing operations on global data in scalable manner.

These techniques, however, are not applicable to all algorithms: as with standard methods, algorithms with inherent data dependencies do not parallelize well. This is born out in the good scalability of the block I/O layer compared to the poor scalability of the file layer. The later could not be written without using mutual exclusion to solve the allocation issues, and thus scaled poorly. Further research is needed to find scalable resource allocation algorithms.

Bibliography

[1] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems 12*, 3 (August 1994), 236–269.

[2] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (June 1994), 145–185.

[3] Corbett, P., Feitelson, D., Hsu, Y., Prost, J.-P., Snir, M., Fineberg, S., Nitzberg, B., Traversat, B., and Wong, P. MPI-IO: a parallel file I/O interface for MPI. Tech. Rep. NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.

[4] Dahlin, M., Anderson, T., Patterson, D., and Wang., R. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings 1994 Operating Systems: Design and Implementation Conference* (November 1994), pp. 267–280.

[5] Dally, W. J., et al. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89* (North Holland, IFIP, 1989), G. X. Ritter, Ed., Elsevier Science Publishers B.V.

[6] DeBenedictis, E., and del Rosario, J. M. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)* (April 1992), pp. 0117–0124.

[7] Deitel, H. M. *An Introduction to Operating Systems*, 2nd ed. Addison Wesley, Reading, Mass., 1990.

[8] Gibson, G. A. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* ACM Distinguished Dissertations. MIT Press, 1992.

[9] Grimshaw, A. S., and Loyot, Jr., E. C. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (1991), p. 177.

[10] HEWLETT PACKARD. *HP C2230 Series 3.5-inch SCSI Disk Drives Technical Reference Manual*, 2 ed., June 1991.

[11] HOWARD, J. H., ET AL. Scale and performance in a distributed file system. *ACM Transactions on Computer Science 6*, 1 (February 1988), 51–81.

[12] KRIEGER, O., STUMM, M., AND UNRAU, R. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer 27*, 3 (March 1994), 75–82.

[13] MASKIT, D., AND TAYLOR, S. A message-driven programming system for fine-grain multicomputers. *Software - Practice and Experience 24* (1994), 953–980.

[14] MASKIT, D., TAYLOR, S., AND ZADIK, Y. System tools for the J-Machine. Department of Computer Science Technical Report CS-TR-93-12, California Institute of Technology, 1993.

[15] NCR MICROELECTRONICS DIVISION. *NCR 53C90 Enhanced SCSI Processor Data Manual*, revision 3.0 ed., March 1989.

[16] NELSON, M., ET AL. Caching in the sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[17] POPEK, G., AND WALKER, B. J. *The LOCUS distributed system architecture.* MIT Press series in computer systems. MIT Press, Cambridge, Mass., 1985.

[18] SUN MICROSYSTEMS, INC. NFS: Network file system protocol specification. Request For Comment 1094, March 1989.

[19] SUN MICROSYSTEMS, INC. NFS version 3 protocol specification. Request For Comment 1813, June 1995.