Active Disks

Anurag Acharya[‡], Mustafa Uysal[†], Joel Saltz[†]

‡ Dept. of Computer Science
University of California
Santa Barbara, CA 93106

† Dept. of Computer Science
University of Maryland
College Park, MD 20742

acha@cs.ucsb.edu, uysal@cs.umd.edu, saltz@cs.umd.edu

TRCS98-06

March 17, 1998

Abstract

Several application and technology trends indicate that it might be both profitable and feasible to move computation closer to the data that it processes. In this paper, we evaluate *Active Disk* architectures which integrate significant processing power and memory into a disk drive and allow application-specific code to be downloaded and executed on the data that is being read from (written to) disk. The key idea is to offload bulk of the processing to the disk-resident processors and to use the host processor primarily for coordination, scheduling and combination of results from individual disks. To program Active Disks, we propose a stream-based programming model which allows disklets to be executed efficiently and safely. Simulation results for a suite of seven algorithms from three application domains (commercial data warehouses, image processing and satellite data processing) indicate that for these algorithms, Active Disks outperform conventional-disk architectures.

1 Introduction

Several application and technology trends indicate that it might be profitable to move data-intensive computation closer to the data that it processes. At the application end, the rate at which new data is being placed online is outstripping the growth in disk capacity as well as the improvement in performance of commodity processors. Furthermore, there is a change in user expectations regarding large datasets – from primarily archival storage to frequent reprocessing in their entirety. Patterson et al [11] quote an observation by Greg Papadopolous - while processors are doubling performance every 18 months, customers are doubling data storage every five months and would like to "mine" this data overnight to shape their business practices [24]. Jim Gray argues that satellite data repositories will grow to petabyte size over the next few years and will require a variety of processing ranging from reprocessing the entire dataset to take advantage of new algorithms to reprojection and composition to suit different display requirements [16, 17]. These trends have two implications: first, large data warehouses will always have a large number of disks and, second, architectures that do not scale the processing power as the dataset grows may not be able to keep up with the processing requirements.

At the technology end, the disk transfer rate has been increasing rapidly. The Cheetah 18 drives from Seagate are capable of delivering up to 21 MB/s [7]; faster drives are on the horizon.

In addition, the power of cheap processors and the size of cheap memory is increasing rapidly. Currently, a 200 MHz Cyrix/IBM 6x86 processor *and* 16 MB of SDRAM can be purchased for about $100 [8, 14]. If current trends continue, by the end of the decade, the same $100 will be able to buy a 266-300 MHz processor with 32 MB of memory. In comparison, the prices of leading-edge disk drives (which would be used in a high-performance data center) are in the $1100-1750 range.[1] These trends have two implications. First, given the improvements in data transfer rates, even a state-of-the-art processor can keep only a small number of drives busy. Second, given the relentless drop in the price of powerful processors and large memory chips, it is becoming economically feasible to place substantial computational capability on individual disks. It is important to note that disk drives already have embedded processors (for servo control) and memory (for disk cache). Current trends indicate that both these components are already scaling up - e.g. disk caches are already up to 4 MB [7].

In this paper, we evaluate *Active Disk* architectures which integrate significant processing power and memory into a disk drive and allow application-specific code to be downloaded and executed on the data that is being read from (written to) disk. To utilize Active Disks, an application is partitioned between a host-resident component and a disk-resident component. The key idea is to offload bulk of the processing to the disk-resident processors and to use the host processor primarily for coordination, scheduling and combination of results from individual disks.

Active Disks present a promising architectural direction for two reasons. First, since the number of processors scales with the number of disks, active-disk architectures are better equipped to keep up with the processing requirements for rapidly growing datasets. Second, since the processing components are integrated with the drives, the processing capacity can be expected to evolve together with the evolution in the disk drives. This is similar to the evolution of disk caches – as the drives get faster, the disk cache becomes larger.

The introduction of Active Disks raises several questions. First, how are they programmed? What is disk-resident code (i.e., a *disklet*) allowed to do? How does it communicate with the host-resident component? Second, how does one protect against buggy or malicious programs? Third, is it feasible to utilize Active Disks for the classes of datasets that are expected to grow rapidly - i.e. commercial data warehouses, image databases and satellite data repositories. To be able to take advantage of processing power that scales with dataset size, it should be possible to partition algorithms that process these datasets such that most of the processing can be offloaded to the disk-resident processors. Finally, how much benefit can be expected with current technology and the foreseeable future?

We address these questions in three ways. First, we propose a stream-based programming model for *disklets* and their interaction with host-resident peers. Disklets take streams as inputs and generate streams as outputs. Files (and ranges in files) are represented as streams. Streams are accessed using a standard interface which delivers the data in buffers whose size is known apriori. A disklet can be written in any language. However, it is required to adhere to certain guidelines. A disklet can not allocate (or free) memory. It is sandboxed [29] within the buffers corresponding to each of its input streams, which are allocated and freed by the operating system, and a scratch space that is allocated on its behalf when it is initialized. A disklet is also not allowed to initiate I/O operations on its own. These restrictions limit the amount of damage that can be done by a disklet. They also simplify the operating system support required on disk-processors.

Second, we present *active-disk* versions of a suite of algorithms that process the datasets of interest. Our suite currently consists of seven algorithms: SQL select, SQL aggregation, SQL

[1] The 9.1 GB, 16 MB/s Cheetah 9 [7] is currently available for about $1100 [21], the 18.2 GB, 21 MB/s Cheetah 18 [7] is available for about $1750 [22].

`group-by`, external sort, the *datacube* operation for decision support [18], image convolution and composition of raw satellite data to generate earth images. The first five algorithms are used in relational databases, the remaining two are used in image databases and satellite data repositories respectively. For each application, we started with well-known and well-optimized algorithms from the literature [3, 4, 15] and tried to keep the modifications to the minimum.

Third, we compare the performance of these algorithms running on active-disk architectures to the performance of the original algorithms running on conventional-disk architectures. Our comparisons use two configurations, one corresponding to current technology and economics and the other corresponding to what is likely to be available by the end of the decade. Given the long life of large datasets and cost of replacing a large number of disks, it is possible that the host system may be upgraded more frequently than the disks. To understand the impact of unbalanced upgrades, we conducted additional experiments with significantly faster host processors. To explore the scalability of the two architectures, we varied the number of disks in each configuration from 4 to 32.

Our results are encouraging. For all of the algorithms, active-disk architectures significantly outperform conventional-disk architectures. Furthermore, active-disk architectures scale well with the number of disks. In comparison, conventional-disk architectures are able to achieve very small improvements from larger disk farms. Finally, active-disk architectures retain their advantage even if the host processor is upgraded more frequently than the disk processors.

2 Programming model

We propose a stream-based model for programming disklets and their interaction with host-resident peers. Disklets take streams as inputs and generate streams as outputs. Streams can be of three types – *disk-resident* streams which are files (or ranges in files), *host-resident* streams which are used by host-resident code to interact with disklets, and *pipe* streams which are used to pipe results of one disklet into another. Streams are accessed using a standard interface which delivers the data in buffers whose size is known a priori.

A disklet can be written in any language. However, it is required to adhere to the safety constraints described above. These constraints are enforced through a combination of download-time analysis and limiting the runtime environment visible to the disklet.

Each disklet must have at least one input stream and at least one output stream. In addition, each disklet must have an initialization function which is run when the disklet is installed. Finally, each disklet must contain a processing function (`read/write`) which is run as data is read/written. A disklet may, optionally, contain long-term scratch space (which is allocated on its behalf before it is installed and is automatically reclaimed after it exits), a set of parameters that can be used to customize its behavior, and a finalization function which is run when the disklet terminates (either by consuming the data on all its input streams or by calling `exit`).

A disklet is not allowed to initiate I/O operations on its own. All I/O operations are initiated by the host-resident program and are checked for validity by the host-resident file-system. This has two advantages. First, disklets can not corrupt the file-system. Second, the operating system layer on the disk-processor need not provide file-system functionality. Requiring the host to initiate all I/O operations can be expensive if individual I/O operations are small. However, algorithms that efficiently process very large datasets can often be restructured to make large I/O requests which allows disklets to run for significant periods of time without synchronization.

A disklet cannot allocate or free memory. All memory management is done by the operating system layer on the disk-processor. Furthermore, all memory accesses by a disklet must be within a sandbox defined by the buffers for its input stream(s) and the long-term scratch space (if any). The disklet binary is analyzed at download-time (as in software fault-isolation [29]); disklets that might possibly violate memory-safety are rejected. The stream-based programming model simplifies this analysis as it defines a natural sandbox for disklets.

Communication between a disklet and its environment is restricted to its input and output streams. The sources and sinks for these streams are specified by the host-resident program as a part of installation of the disklet. A disklet is not allowed to determine (or change) where its input stream comes from or where its output stream goes to. This has two advantages. First, a disklet does not handle buffering and scheduling for its communication, the operating system layer does. This reduces the complexity of disklets. Second, in a heterogeneous environment with both Active Disks and conventional disks, this allows disklets that process data from conventional disks to be transparently executed on the host itself.

Figure 1 presents pseudo-code for a disklet that performs image convolution and the corresponding host-resident code. Note that installation and invocation of disklets are separate operations. This decision is based on the expectation that once downloaded, disklets will be used repeatedly.

3 Operating-system support

Active Disks require operating system support both at the host and on the disk-processor. Design of the OS layer at the disk (or the *DiskOS*) has conflicting requirements. On one hand, we would like the DiskOS to be as thin as possible so that the disklets can make full use of the limited resources. On the other hand, we would like to move as much as possible of the common functionality into the DiskOS so that disklets can be small and easy to analyze. Our design takes advantage of the stream-based programming model to keep the provide the needed functionality and yet keep the memory footprint of the DiskOS small.

DiskOS: the DiskOS provides three services – memory management, stream communication and disklet scheduling. The stream-based model simplifies memory management as all memory is allocated in contiguous blocks whose size is known a priori and the lifetime of all blocks is known. The stream-based model also simplifies the communication support required as all stream buffers are allocated and managed by the DiskOS. Depending on the amount of memory available, it can allocate multiple buffers and overlap data movement and computation. The desire for coarse-grain partitioning results in long I/O requests (e.g., see Figure 1) which take the guesswork out of prefetching and allow the DiskOS to make efficient use of the limited memory. The DiskOS exports an SCSI-level interface to the host – all I/O requests must be translated into requests for SCSI-ranges by the host operating system. The stream-based model also simplifies scheduling for disklets. A disklet is ready to run whenever there is new data available on one or more of its input streams. The scheduling discipline is run-to-completion. At any given time, only one host-resident program can download disklets. We don't expect this to be a limitation as the applications under consideration are extremely large and usually run on dedicated machines. Also, we assume that the I/O interconnect is physically secure and that the DiskOS does not need to authenticate the (single) user. These restrictions may seem onerous but they are naturally satisfied by large data centers, which is where we expect Active Disks to be used (at least at the beginning).

Host-level OS support: two new host-level OS functions are needed for Active Disks – installation of disklets and management of host-resident streams. Disklet installation requires analysis of disklet code to ensure memory safety [29], linking against the DiskOS environment and downloading the code to the disk-processor. Creation and management of host-resident streams is relatively simple.

(a) disk-resident code	(b) host-resident code
see below	see below

(a) disk-resident code

```
disklet convolve(instream in,
                 outstream out)
 integer kernel[3][3], sidelen;
 integer i,j,k,l,m,temp;
 buffer buffer;

 function init(integer filter[3][3],
               integer imgside) {
   copy(filter,kernel);
   sidelen ← imgside;
 }

 function read {
  /* get next buffer */
  buffer ← getNextBuffer(in);

  /* perform convolution */
  for (i ← 0; i < sidelen; i++) {
   for (j ← 0; j < sidelen-3; j++) {
    for (k ← 0; k < sidelen-3; k++) {
      t ← 0;
      for (l ← 0; l < 3; l++) {
       for (m ← 0; m < 3; m++) {
        t += (buffer[j+l][k+m] *
              kernel[l][m]);
       }}
       buffer[j][k] ← t;
  }}}
  send(out,buffer);
 }
end
```

(b) host-resident code

```
integer filter[3][3] = {{-1, -2, -1},
                        {0,   0,  0},
                        {1,   2,  1}};

for (i ← 0; i < NDISKS; i++) {

 /* open image file on every disk */
 sprintf(fname,''/disks%/ifile'',i);
 fd[i] = open(fname,O_RDONLY);
 flen[i] = flength(fd[i]);
 /* create a stream for every disk.
  * output from disklet; 256K buffer */
 sprintf(sname,''/stream/str%'',i);
 stream[i] = open_str(sname,READ,262144);

 /* install the disklet */
 install convolve(filter,512);
 /* invoke disklet; whole file as input
  * input stream spec has descriptor,
  * offset and length; 256K buffer */
 run convolve(mkStr(fd[i],0,flen[i],262144),
              streams[i]);
}

/* while not all streams have terminated.
 * str_select blocks, returns a stream
 * that has new data */
while ((s ← str_select(streams)) != NULL) {
  buf ← getNextBuffer(s);
  processBuffer(buf);
}

for (i ← 0; i < NDISKS; i++) {
  close(fd[i]);
  close_stream(stream[i]);
}
```

Figure 1: Pseudo-code for partitioned version of convolution filtering. The filter used in this example detects horizontal lines in images.

The notion of streams is already being used for I/O in current operating systems. The primary difference between the semantics of streams currently used in operating systems and the semantics proposed for Active Disk streams is that the latter deliver data in a quantized manner – in a sequence of buffers whose size is known a priori. These buffers are allocated and freed by the operating system. A buffer for a `read` stream "s" is allocated using `getNextBuffer(s)` and is reclaimed by the OS at the next call to `getNextBuffer(s)`, `close_stream(s)` or program exit. Buffers for `write` streams have similar semantics. Buffers are allocated using `getNextBuffer()` and are implicitly returned to the operating system by a `send()`, `close_stream()` or program exit. Since all buffers are allocated and freed by the operating system, it is able to implement zero-copy semantics if desired [6, 9]. Given the expected size of the datasets under consideration, this flexibility is likely to be important.

4 Algorithms

In this section, we describe conventional-disk and active-disk versions of the algorithms in our suite. Our suite currently consists of seven algorithms from three application domains - relational database processing, image processing and satellite data processing. The conventional-disk algorithms are optimized for I/O performance. All algorithms except external sort stripe the data across all disks with 256 KB per disk per stripe. For external sort, the data is striped on half of the disks for reading and the other half for writing. In addition, all algorithms use aggressive prefetching issuing up to eight asynchronous I/O requests (two for sort phase one), each request being for a complete stripe across all disks (half of the disks for sort). The combination of large requests and deep request-queues allows the conventional-disk algorithms to take full advantage of the aggressive I/O subsystem.

SQL SELECT: SELECT filters tuples from a relation based on a user-specified predicate [20]. Database administrators usually build indices on one or more attributes to speed up SELECTs; SELECTs on non-index attributes, however, require scanning the entire relation. We focus on such operations. SELECT applies the filtering predicate independently to each tuple and, therefore, is amenable to coarse-grain parallelization. The active-disk algorithm applies the SELECT predicate at the disk and forwards only the successful tuples to the host. The input stream for the downloaded disklet consists of the entire file on its disk (as in Fig 1). The scratch-space consists of a large buffer that is used to collect tuples that are to be sent to the host. The disklet applies the predicate to each tuple in an input buffer and copies it to the outbound buffer if it satisfies the predicate. When the outbound buffer is full, it is shipped to the host. At the host, data from different disks is concatenated in preparation for transfer to the requesting client.

SQL aggregation: The SQL aggregation operations allow users to compute zero-dimensional aggregates (i.e., single values) from a single relation. The SQL standard provides five aggregation functions: `MIN` `MAX`, `SUM`, `AVG` and `COUNT` [20]. Aggregation functions are order-independent and need only a small amount of intermediate storage. The active-disk algorithm utilizes the order-independent nature of aggregation. The input stream for a downloaded disklet consists of the entire file on its disk. Each disklet aggregates the values from its local tuples into a local accumulator. When all local tuples have been processed, the local aggregate is forwarded to the host which aggregates values received from all disks.

SQL `group-by`: the SQL aggregation functions compute a single aggregate value; the `group-by` operation generalizes this to compute a vector of aggregates indexed by a set of attributes [20]. It partitions a relation into disjoint sets of tuples based on the values of index attributes and then aggregates over each set. Graefe [15] shows that hashing-based techniques out-perform sort-based and nested-loop-based techniques for implementing `group-bys`. Accordingly, we used the algorithm from [15] as our conventional-disk algorithm. The active-disk algorithm performs the `group-by` in two steps. The downloaded code performs local `group-bys` as long as the number of aggregates being computed fits in the disk-memory. When it runs out of space, it ships the partial results to the host and reinitializes the disk-memory. The host accumulates the partial results forwarded by all disklets.

External sort: the current record for the fastest external sort is held by `nowsort` [4]. Accordingly, we used it as the starting point for both versions of external sort. We used the pipelined version of the two-pass single-node sort [4] for the conventional-disk version. The first phase uses two threads: a *reader-thread* to read data and move tuple pointers to buckets and a *writer-thread* to sort each bucket with partial-radix sort[2] and write the bucket. The second phase uses three threads to merge the sorted partitions created in the first phase. A *reader* reads one block from each sorted partition into one of eight[3] sets of merge buffers; a *merger* selects the lowest-valued key from the current block of each partition and copies it to one of eight[3] write buffers; a *writer* writes buffers to disk.

The active-disk algorithm uses two disklets for the first phase, the `partitioner` and the `sorter`. The `partitioner` uses its scratch-space to form as many buckets as the number of disks. It examines each record and appends it to the bucket corresponding to its destination disk. When one of these buffers fills, it is forwarded to the host. The host maintains two global buffers for every disk and copies the forwarded records into the appropriate buffer. When a global buffer for a disk fills, the host sends it to the `sorter` disklet on that disk. Two global buffers are allocated for every disk to allow the host to make progress while one of them is being sent to the `sorter`. The `sorter` sorts each buffer using a partial radix sort and writes it to disk by sending it to the output stream. In the second phase, each sorted partition created in the first phase is mapped to a different stream; these streams are attached to a `merger` disklet as its inputs. The `merger` selects the lowest key from *all* of its input streams and copies it to its output stream (which is mapped to the output file). Note that, no data is sent to host in this phase; merging is done locally at each active disk.

Datacube: *datacube* is the most general form of aggregation for relational databases. It computes multi-dimensional aggregates that are indexed by values of multiple aggregates [18]. In effect, a datacube computes group-bys for all possible combinations of a list of attributes. Several efficient methods for computing a datacube are presented in [3]. We use one of these algorithms, called *PipeHash*, as the conventional-disk algorithm. PipeHash represents the datacube as a lattice of related group-bys. A directed edge connects group-by i to group-by j if j can be generated from i and has exactly one less attribute. Each edge has an associated weight which reflects the estimated size of the group-by. PipeHash determines the set of group-bys to perform by computing a minimum spanning tree over the lattice (see Fig 4 for an example). It schedules the group-bys as a sequence of pipelines; all the group-bys in a pipeline are computed with a single scan of disk-resident data. The final results of each pipeline are stored back on disk and are used as input for following pipelines (see Fig 4 for examples of pipelines). For individual group-bys, PipeHash uses a hashing-based technique [15]. Like the other conventional-disk algorithms in our suite, our implementation of this algorithm uses striping, large requests and aggressive prefetching.

The active-disk algorithm uses a separate disklet for every pipeline. Each disklet creates the hash-tables for its component group-bys in its scratch-space. The hash-tables for the different

[2] Making two passes over the keys with a radix size of 11-bits [2] plus a cleanup.
[3] Two buffers in the original algorithm.

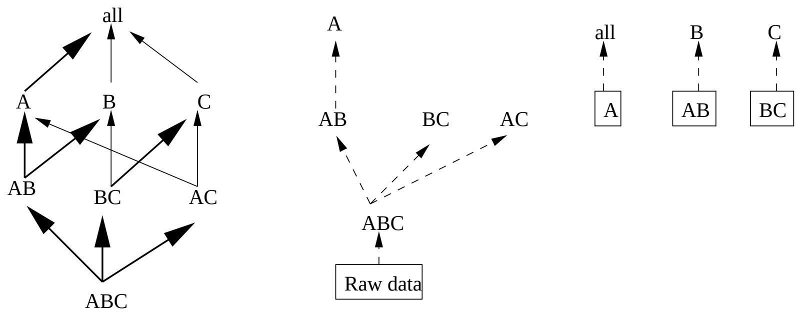


Figure 2: Search lattice and pipelines for the PipeHash algorithm. The bold lines in the search lattice indicate the minimum spanning tree computed by the algorithm using estimated sizes of individual group-bys. The right hand side shows four pipelines. In each pipeline, the data placed in a box at the bottom is read from disk.

group-bys are allocated memory in proportion to the estimated size of the group-bys. Each disklet performs local `group-bys` as long as the number of aggregates being computed fits in the disk-memory. When it runs out of space, it ships the partial results to the host and reinitializes the disk-memory. The host accumulates the partial results forwarded by all disklets and writes the final results to disks (for use in later pipelines).

Image convolution: Convolution is widely used to enhance spatial features or subdue noise in images. Applications include edge detection, gradient detection, smoothing and blurring, image sharpening etc. In general, it is used to implement operators which compute the new value of a pixel as a linear combination of its own value and the values of its neighboring pixels. The coefficients for the linear combination are specified as a matrix (known as the *kernel*). Convolution is performed by sliding the kernel over the image starting at the top left corner. The new value for each pixel is computed as the sum of the point-wise products of the kernel values with the values of the pixels it covers. The conventional-disk algorithm concatenates all the images into a single file and stripes this file across all disks such that each stripe contains an integral number of images. The active-disk algorithm stores a sequence of images on each disk; the buffer for the corresponding input stream is sized to exactly fit one image. The downloaded disklet takes the convolution kernel as an argument and performs the convolution operation in-place on entire images and forwards the processed images to the host. Note that in-place convolution only needs scratch-space proportional to the size of the kernel (e.g., nine values for a 3x3 kernel).

Generating composite satellite images: Earth scientists generate earth images by compositing remotely-sensed data acquired over multiple days from satellite-based sensors. Generating a composite image requires projection of the sensor values onto a two-dimensional grid followed by composition of all values that map onto a single grid point to generate the associated pixel. Sensor values are pre-processed to correct the effects of various distortions. The conventional-disk algorithm is based on the technique used in several programs used by NASA [10, 13, 28]. It maps processes sensor values in large chunks, mapping each value to the output grid and performing the composition operation using an accumulator for every output pixel. The active-disk algorithm takes advantage of the fact that the composition is order-independent (it uses a complex `max`-like operation) to perform as much of the composition locally as possible. The size of the output image, however, is large and cannot be expected to fit into disk-memory – e.g. even a coarse-grained image is about 228 MB [28]. To deal with this, the downloaded disklet allocates space for a subimage corresponding to a contiguous section of the output grid that fits into the disk-memory. As long as the sensor values being read map into this subgrid, the composition operation is performed locally. When it encounters a sensor value that maps outside this subgrid, it ships the entire subimage to the host and shifts the subgrid such that the new sensor value maps into its center. The host maintains the accumulators for the entire image. As each subimage is received at the host, it is composed into the final image. Composition at both the host and the disk use the same per-pixel composition operation mentioned above.

5 Evaluation

To evaluate the utility of Active Disks, we compared the performance of conventional-disk and active-disk versions of the algorithms in our suite for a variety of configurations and datasets. For the core set of experiments, we used two configurations – one corresponding to systems that can be purchased today and the other corresponding to systems that are likely to be available by the end of the decade. Table 1 describes these configurations. Given the long life of large datasets and cost of replacing a large number of disks, it is possible that the host system may be upgraded more frequently than the disks. To understand the impact of unbalanced upgrades, we conducted additional experiments with significantly faster host processors. To explore the scalability of the two architectures, we varied the number of disks in each configuration from 4 to 32.

Parameter	Today	Future
Host CPU	350 MHz	500 MHz
Host Memory	1 GB	1 GB
I/O Interconnect	200 MB/s	300 MB/s
Disk CPU	200 MHz	300 MHz
Disk Memory	16 MB	32 MB
Media Rate	15 MB/s	20 MB/s
RPM	10000	12500
Mean and Max seek time	9 ms, 18 ms	9 ms, 18 ms

Table 1: Configurations for the core set of experiments. The host processor for the `Today` configuration is the Pentium II 350, the fastest Intel processor available today; the disk processor is the Cyrix/IBM 6x86. The disk-processor and disk-memory for both configurations were selected with a cost constraint of $100. The I/O interconnect is assumed to be dual FiberChannel loops with 100 MB/s per loop for `Today` and 150 MB/s per loop for `Future`.

5.1 Simulator

To conduct these experiments, we developed the *ADsim* simulator which simulates both conventional-disk and active-disk architectures. *ADsim* contains a detailed disk model, a preliminary implementation of *DiskOS*, and relatively coarse-grain models of the processor and the I/O interconnect. The disk model is based on the Ruemmler&Wilkes' disk model [27]. We used the implementation by David Kotz [19] as the initial codebase. The disk model includes a fixed cost for the controller overhead, a seek model that models short seeks and long seeks separately,[4] a rotational model and a

[4]The latency of initiating the disk-head moves dominates for short seeks whereas the time to actually move the head dominates for long seeks.

model for disk-geometry that includes multiple zones, inline track-sparing, track skew and cylinder skew.

The *ADsim* processor model characterizes user-level tasks by the time they take to execute if run without interruptions. User-level tasks can be pre-empted by I/O interrupts. In addition to execution times for user-level tasks (which are fed in as a trace file), the model includes time to execute a `null` system call, time to queue an I/O request in the device-driver and time to service an I/O interrupt. The *ADsim* model for the I/O interconnect has three parameters, the latency to initiate a transfer, the capacity of the interconnect and the transfer speed.

To determine the time taken for user-level tasks, we implemented each of these algorithms; ran them on a DEC Alpha 2100 4/275 workstation with 256 MB of memory; and scaled the execution time for the processors in the experimental configurations. For each of these runs, we used the same datasets and the same buffer size as assumed in the simulations. We used separate runs to obtain the execution times for conventional-disk and active-disk versions of the `datacube` algorithm. This was necessary as the amount of memory allocated to hash-tables depends on the data and on the structure of the algorithm.

ADsim includes a preliminary implementation of *DiskOS* which provides support for scheduling disklets as well as for managing memory, I/O and stream communication. Disklets are written in C and interact with *ADsim* using an API. DiskOS initiates I/O operations on behalf of disklets; disk blocks mapped to a disk-resident input stream are automatically read by DiskOS and are delivered to a disklet; buffers for disk-resident output streams are automatically written to disk. The buffers for DiskOS-generated read operations are automatically allocated (since stream buffer size is fixed) and buffers for completed write operations are reclaimed by the DiskOS. The current implementation of DiskOS overlaps I/O and computation at the disk-processor by using two buffers per stream.

5.2 Datasets

Select, aggregation, group-by: we used two datasets with 64-byte tuples. The smaller dataset was 4 GB (67 million tuples) and the larger dataset was 8 GB (134 million tuples). For `group-by`, the smaller dataset had 3.35 million distinct values and the larger dataset had 6.7 million distinct values. For `select`, we varied the fraction of matching tuples from 1% to 80%.

Sort: we used two datasets with 100-byte tuples and 10-byte uniformly distributed keys.[5]. The smaller dataset was 2 GB (21.5 million tuples) and the larger dataset was 4 GB (43 million tuples).

Datacube: we used two datasets with 32-byte tuples. The smaller dataset was 4 GB (134 million tuples) and the larger dataset was 8 GB (268 million tuples). Each tuple has eight 4-byte attributes. We used four attributes as group-by attributes and the remaining four as aggregation attributes with `SUM` as the aggregation function. The number of distinct values for each of the group-by attributes were 1342177, 134217, 13421, and 1342 for the smaller dataset and 2684354, 268434, 26842 and 2684 for the larger dataset.

Image convolution: we used a single dataset consisting of 10,000 512x512 images with one byte per pixel. We used a 16x16 convolution kernel.

Earth science: we used two datasets which correspond to ten-day composites of low-resolution and high-resolution AVHRR images from the NOAA polar-orbiting satellites [28]. The size of the

[5] We have taken these values from the paper describing `nowsort` [4]

smaller dataset was 6 GB and the size of the larger dataset was 12 GB. The output image for the smaller dataset was 228 MB and the output image for the larger dataset was 556 MB.

5.3 Utility of Active Disks

Figure 3 compares the performance of our algorithm suite on conventional-disk and active-disk architectures with four disks each. Each bar presents normalized execution time for *all four architecture-configuration pairs*. For normalization, the performance of the conventional-disk version on `Today` is assumed to be 100%. Note that the sub-bars corresponding to individual architecture-configuration pairs are **cumulative**. The normalized execution-time for a sub-bar is the y-coordinate of its top and not its length. For example, `select` on CD-Future takes 86.4% of the time it takes on CD-Today which is the height of its top and **not** 7%, the length of its sub-bar. Note that each figure presents results from 28 experiments.

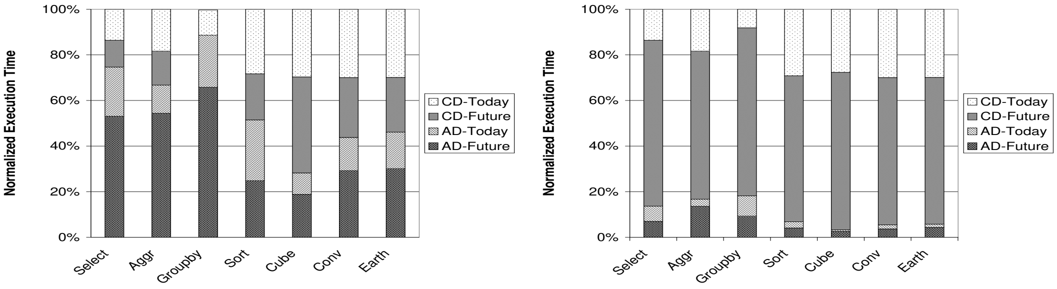
We note that for all applications, Active Disks outperform conventional disks. Active-disk architectures with components available today outperform conventional-disk architectures with components that are likely to be available at the end of the decade. This is true even for the four disk configurations which are the smallest configurations in our experiments and therefore the least favorable to Active Disks.

Our results indicate that for 4-disk configurations, Active Disks are between 12% and 354% faster than conventional disks assuming `Today`'s components; and between 34% and 374% faster assuming `Future` components. For 32-disk configurations, Active Disks are between 5.5 times and 30.1 times with `Today`'s components; and between 6 times and 27.4 times with `Future` components. We also note that the first three algorithms on the left of Figure 3 are the most I/O-intensive with little computation per data transferred while the last two algorithms on the right are significantly more compute-intensive than all the others. As a result, much of the performance improvements for the later comes from the parallelism exhibited by the active-disk algorithms.

Figure 4 shows how each of the algorithms performs as the dataset size increases. Most of the algorithms in our suite are linear and accordingly, their execution time increases linearly with dataset size on both architectures. The only exception is `sort` whose execution time grows in a superlinear (but sub-quadratic) manner for conventional-disk architectures and in a quadratic manner for active-disk architectures. The quadratic growth for the `sort` is due to memory limitations. As the dataset doubles, the active-disk algorithm is forced to generate twice the number of sorted partitions it generated for the smaller dataset. Accordingly, in the second phase, it has to check twice as many keys from each of the partitions to determine the smallest key. The conventional-disk algorithm does not have this problem for the datasets used in these experiments as it can take advantage of the 1 GB host memory.

5.4 Scalability

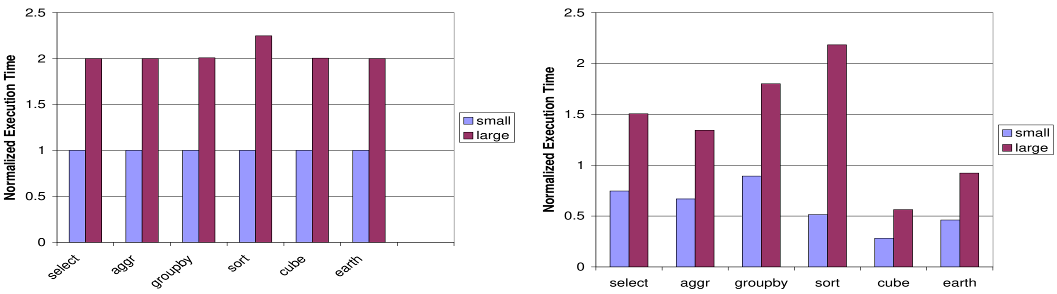
To explore the scalability of the two architectures, we compared the performance of each algorithm for configurations with varying number of disks. Table 2 presents the results. We note that increasing the number of disks beyond four for conventional-disk architectures provides little or no advantage. That is, given applications that have been optimized for I/O performance and an aggressive I/O subsystem, a small number of fast disks can keep the processor busy for the algorithms in our suite. Other researchers have reached similar conclusions [26]. On the other hand, active-disk architectures scale perfectly up to 16 disks. The algorithms that perform significant computation in disklets, i.e., `sort`, `cube`, `conv` and `earth` scale well up to 32 disks. The speedups for the algorithms that do not perform significant computation in the disklets, however, level off



(a) 4-disks, the smaller data sets

(b) 32-disks, the smaller data sets

Figure 3: Comparison of Active Disks with conventional disks. The legend indicates the architecture type (CD/AD) and the configuration (Today/Future). Each bar presents normalized execution time for all four architecture-configuration pairs. For normalization, the performance of the conventional-disk version on `Today` is assumed to be 100%. Note that the sub-bars corresponding to individual architecture-configuration pairs are **cumulative**. The normalized execution-time for a sub-bar is the y-coordinate of its top and not its length. For example, `select` on CD-Future takes 86.4% of the time it takes on CD-Today which is the height of its top and NOT 7%, the length of its sub-bar. Note that each figure presents results from 28 experiments.



(a) Conventional Disks

(b) Active Disks

Figure 4: Impact of scaling datasets. All experiments were conducted with the four disk versions of the `Today` configurations. The execution time of each application is normalized with respect to its execution time for the small dataset on the conventional-disk architecture. For example, the execution time for `select` for its 4 GB dataset running on conventional disks is 1.0.

by 32 disks. Note that `sort` achieves near-quadratic speedup as the number of disks is increased. This is due to the increased disk-memory available in larger configurations that allows each disk to work with small number of partitions in the second phase.

No. of	select		aggr		groupby		sort		cube		conv		earth	
Disks	**CD**	**AD**	**CD**	**AD**	**CD**	**AD**	**CD**	**AD**	**CD**	**AD**	**CD**	**AD**	**CD**	**AD**
4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	1.0	2.0	1.0	2.0	1.0	2.0	1.1	2.9	1.0	2.0	1.0	2.0	1.0	2.0
16	1.0	4.0	1.0	4.0	1.1	4.0	1.1	8.3	1.1	4.0	1.0	4.0	1.0	4.0
32	1.0	7.6	1.0	8.0	1.1	5.1	1.1	17.0	1.1	7.9	1.0	8.0	1.0	7.9

Table 2: Impact of scaling the configuration. The numbers presented are normalized execution times on conventional-disk (CD) and active-disk (AD) architectures with varying number of disks. The execution times in each column are normalized with respect to the first entry. In effect, each number indicates how much speedup can be achieved just by increasing the number of disks. All results in this table are for `Today` configurations.

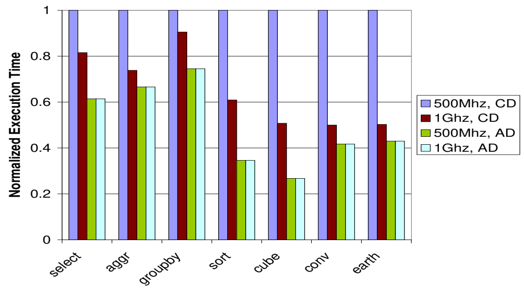
5.5 Impact of upgrading the host processor

Given the long life of large datasets and cost of replacing a large number of disks, it is possible that the host system may be upgraded more frequently than the disks. As the ratio between the speeds of the host-processor and the disk-processors increases, it is possible that offloading bulk of the processing to the slower processors could become counter-productive.

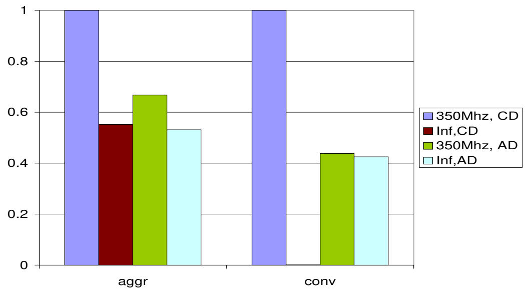
In order to explore the impact of more frequent upgrades to the host-processor, we compared the performance of active-disk architectures with 4 disks to conventional-disk architectures with a significantly faster processor (1 GHz). The results are presented in Figure 5(a). Even though the faster processor significantly improves the performance of conventional-disk architectures, performance of active-disk architectures is still superior. In order to understand the limiting case of this trend, we assumed an infinite speed processor at the host and a 200MHz processor at the disk and compared the performance of this configuration with that of `Today` configurations. We used the two extreme algorithms for this comparison – one with the minimum computation (`aggregation`) and the other with maximum amount of computation (`conv`). Figure 5(b) shows that even for this limiting case, active-disk architectures retain their advantage over conventional-disk architectures for at least one algorithm that is likely to be used for large datasets.

6 Discussion and related work

The fact that we were able to easily convert several I/O-intensive algorithms to a stream-based programming model should not come as a surprise. Given the volume of data processed and the cost of fetching data from disk, optimizing I/O-intensive algorithms often is matter of setting up efficient pipelines where each stage performs some processing on the data being read from disk and passes it on to the next stage [1, 4]. The SQL standard already supports a simpler version of the stream-based model proposed in this paper via the `cursor` interface [20]. This interface allows a client application to ship a query to the server and receive the results of the query one tuple at-a-time. The model proposed in this paper, however, provides greater functionality than SQL `cursors` as the downloaded code is not restricted to relational queries, multiple disklets can



(a) 1GHz CPU



(b) Infinite speed CPU

Figure 5: Impact of using significantly faster host processors. All experiments were conducted with 4-disks configurations and the smaller data sets. The execution time of each application is normalized with respect to its execution time on the conventional disks with `Today`'s configurations. `Convolution` is compute-dominated; its execution time becomes extremely small for the infinite speed processor.

be downloaded and can act in concert and the buffer size can be adjusted to meet the needs of the application.

The idea of embedding a programmable processor in a disk is not new. In fact, the I/O processors in the IBM 360 allowed users to download *channel programs* that were able to make I/O requests on behalf of the host programs [25]. One of the ISAM implementations on the IBM 360 used channel programs to traverse disk-resident linked lists. What is different today is the power of the processor and the amount of memory that can be economically integrated into disk drives.

As a part of the NASD effort,[6] several researchers are exploring the use of processors that are integrated with disk drives. Borowsky et al [5] are investigating the use of disk-processors to implement quality-of-service guarantees for data retrieval. Gibson et al [12] are investigating the use of disk-processors for performing file-system and security-related processing on network-attached disks. In contrast, our research focuses on the performance of large-scale applications and the utility of downloading application-specific code to disk-processors.

The idea of moving computation closer to the data it processes is also being explored for other devices. The growing *processor-memory gap* [30] has lead to several proposals for integrating processing logic into DRAM [23, 11]. Chong et al [23] have proposed that data-intensive programs should be partitioned into processing-intensive and memory-intensive components and the memory-intensive component should be offloaded into logic placed in the memory system.

In this paper, we have assumed a traditional processor-memory organization for the processor and the DRAM that is integrated into the disk drive. While this may be the fastest way to achieve this integration in the short term, we believe that, in the long run, integrated processor-memory chips like the IRAM [11] will probably be the most suitable way of embedding processing power in disk drives.

[6]Network-attached Storage Devices. See *http://www.hpl.hp.com/SSP/NASD*.

7 Conclusions and future work

In this paper, we evaluate *Active Disk* architectures which integrate significant processing power and memory into a disk drive and allow application-specific code to be downloaded and executed on the data that is being read from (written to) disk. To utilize Active Disks, an application is partitioned between a host-resident component and a disk-resident component. The key idea is to offload bulk of the processing to the disk-resident processors and to use the host processor primarily for coordination, scheduling and combination of results from individual disks.

To program Active Disks, we have proposed a stream-based programming model. Disklets take streams as inputs and generate streams as outputs. A disklet can be written in any language. However, it is required to adhere to certain guidelines. A disklet can not allocate (or free) memory. It is sandboxed within the buffers corresponding to each of its input streams, which are allocated and freed by the operating system, and a scratch space that is allocated on its behalf when it is initialized. A disklet is also not allowed to initiate I/O operations on its own. These restrictions limit the amount of damage that can be done by a disklet. They also simplify the operating system support required on disk-processors.

To demonstrate that the Active Disks are suitable for large and rapidly growing datasets, we have presented active-disk versions of several algorithms that process such datasets. We have compared the performance of these algorithms running on active-disk architectures to the performance of the optimized versions of the original algorithms running on conventional-disk architectures.

Our results are encouraging. For all of the algorithms, active-disk architectures significantly outperform conventional-disk architectures. Furthermore, active-disk architectures scale well with the number of disks. In comparison, conventional-disk architectures are able to achieve very small improvements from larger disk farms. Finally, active-disk architectures retain their advantage even if the host processor is upgraded more frequently than the disk processors.

In the near future, we plan to expand our suite of algorithms. In particular, we are working an active-disk algorithm for computing the frequent itemsets needed for data mining of historical transaction data. In the longer term, we plan to further explore operating system issues for Active Disks. In particular, we would like to relax the single-user restriction for Active Disks.

We plan to make *ADsim* publicly available. Our goal is to encourage other researchers to explore active-disk architectures.

Acknowledgments

We would like to thank Jim Gray for asking us to think *What Happens When Processors Are Infinitely Fast and Storage Is Free?*

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[2] R.C. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of 1996 ACM SIGMOD Conference*, pages 240–6, 1996.

[3] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd Int'l Conference on Very Large Databases (VLDB)*, pages 506–21, Bombay, India, August 1996.

[4] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, D.E. Culler, J.M. Hellerstein, and D.A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of 1997 ACM SIGMOD Conference*, Tucson, AZ, 1997.

[5] E. Borowsky, R. Golding, A. Merchant, L. Schrier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *5th International Workshop on Quality of Service*, 1997.

[6] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct 1996.

[7] Cheetah Specifications. *http://www.seagate.com/disc/cheetah/cheetah.shtml*, Feb 1998.

[8] Custom Network Technologies Product Catalog. *http://www.cntwv.com/catalog.htm*, Feb 1998. Following link from *http://www.lowerprices.com*.

[9] P. Druschel and L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proc. of 14th SOSP*, pages 189–202, Dec 1993.

[10] Jeff Eidenshink and Jim Fenno. Source code for LAS, ADAPS and XID, 1995. Eros Data Center, Sioux Falls.

[11] D. Patterson et al. Intelligent RAM (IRAM): the Industrial Setting, Applications, and Architectures. In *Proceedings of the International Conference on Computer Design*, 1997.

[12] G. Gibson et al. File server scaling with network-attached secure disks. In *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, 1997.

[13] Gene Feldman. Source code for the SeaWIFS ocean data processing system, 1995. SeaWIFS group (NASA Goddard).

[14] GR Company Store Memory. *http://www.grcompany.com/grstore/Memory.htm*, Feb 1998. Following link from *http://www.lowerprices.com*.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[16] J. Gray. Some challenges in building petabyte data stores. Distinguished Lecture, University of California, Santa Barbara, Oct 1997.

[17] J. Gray. What happens when processors are infinitely fast and storage is free? Keynote Speech at IOPADS'97, Nov 1997.

[18] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–9, New Orleans, February 1996.

[19] D. Kotz, S. Toh, and S. Radhakrishnan. A detailed simulation model of the HP97560 disk drive. Technical Report PCS-TR94-220, Darthmouth College, 1994.

[20] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufman, 1993.

[21] Microwarehouse online catalog. *http://www.microwarehouse.com/MicroWarehouse/Storage/Fixed_Drives/*, Feb 1998.

[22] The 1998 New Media Hyper Awards. *http://newmedia.com/NewMedia/98/03/feature/storage.html/*, March 1998.

[23] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A computation model for intelligent memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998. To appear.

[24] G. Papadopolous. The future of computing. Unpublished talk at NOW Workshop, July 1997.

[25] D. Patterson and J. Hennessey. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2nd edition, 1996.

[26] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79–95, 1995.

[27] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[28] Peter Smith and Bin-Bin Ding. Source code for the AVHRR Pathfinder system, 1995. Main program of the AVHRR Land Pathfinder effort (NASA Goddard).

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–16, 1993.

[30] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1), 1995.