Active Disks: Programming Model, Algorithms and Evaluation

Anurag Acharya
Dept. of Computer Science
University of California
Santa Barbara

Mustafa Uysal
Dept. of Computer Science
University of Maryland
College Park

Joel Saltz
Dept. of Computer Science
University of Maryland
College Park

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 six 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 and feasible 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 [25] 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 [12, 13]. 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 [6]; 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 [7, 28]. 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 [6].

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

[1] The 9.1 GB, 16 MB/s Cheetah 9 [6] is currently available for about $1100 [21], the 18.2 GB, 21 MB/s Cheetah 18 [6] is available for about $1750 [17].

components are integrated with the drives, the processing capacity will evolve as the disk drives evolve. 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 in 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 [32] 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. However, it is allowed to skip parts of its input streams. 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 partitioned versions of a suite of algorithms that process the datasets of interest. We present six algorithms: SQL select,SQL `group-by`, external sort, the *datacube* operation for decision support [14], image convolution and generation of earth images from raw satellite datax. The first four 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 algorithms from the literature [3, 4, 11] and tried to keep the modifications to the minimum.

Third, we compare the performance of the partitioned algorithms running on Active Disks to the performance of the original algorithms running on conventional disks. 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. In addition, we evaluate the sensitivity of these results to large variations in the interconnect bandwidth and host processor speed. These experiments help identify the bottlenecks. They also help understand the impact of unbalanced upgrades. Given the long life of large datasets and cost of replacing a large number of disks, it is possible that other parts of the system are upgraded more frequently than the disks.

Our results are encouraging. For all the algorithms used in our evaluation, active-disk architectures outperformed conventional-disk architectures. Our results indicate that 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. However, for algorithms that redistribute all or most of their input data, routing all data through the host (in active-disk architectures) could become a bottleneck for large configurations. Finally, we have shown that active-disk architectures retain most of their advantage even if the host processor is upgraded more frequently than the disks.

2 Programming model

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. 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.

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 need not provide file-system functionality. While a disklet is not allowed to initiate I/O operations, it is allowed to *skip* subranges in an input stream by notifying the operating-system layer on the disk. The skipped subranges are not delivered to the disklet. This allows disklets to safely implement algorithms in which future I/Os depend on data from previous I/Os. For example, an algorithm that uses a disk-resident index to decide which chunks of data are to be read can be implemented by a disklet with two input streams – one corresponding to the index file and the other corresponding to the data file. It uses the data delivered on the index stream to decide which parts of the data stream are to be read and which are to be skipped.

A disklet cannot allocate or free memory. All memory management is done by the operating-system layer on the disk. 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 [32]); disklets that *may* violate memory-safety are rejected. The stream-based programming model simplifies the 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 the 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. A disklet can be written in any language. However, it is required to adhere to the safety constraints described above. These constraints are enforced by a combination of download-time analysis and a restricted runtime environment.

3 Operating-system support

Active Disks require operating system support both at the host and on the disk. 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 provide the needed functionality and yet keep the 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 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. Currently, we assume that the scheduling discipline is run-to-completion and that at any given time, only one host-resident program can download disklets. We don't expect the second assumption to be a problem as the applications under consideration are extremely large and usually run on dedicated machines. The run-to-completion scheduling, however, could become a problem for large installations that run multiple concurrent jobs, e.g. multiple decision-support queries. We are currently investigating ways to relax this restriction.

Host-level OS support: limited new host-level OS functionality is needed – support for installation of disklets and management of host-resident streams. Disklet installation requires analysis of disklet code to ensure memory safety [32], linking against the DiskOS environment and downloading the code to the disk. Creation and management of host-resident streams is relatively simple. 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 by the operating-system and are freed implicitly. A buffer for a `read` stream "s" is allocated using `getNextBuffer(s)` and is freed (implicitly) by the next call to `getNextBuffer(s)`, `close_stream(s)` or program exit. Buffers for `write` streams have similar semantics.

4 Algorithms

In this section, we describe conventional-disk and active-disk versions of the algorithms in our suite. We present six 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 chunks 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 these 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 can build indices on one or more attributes to speed up SELECTs; SELECTs on non-index attributes, however, are fairly frequent and 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 Figure 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 `group-by`: The `group-by` operation allows users to compute a one-dimensional vector of aggregates indexed by a list of attributes [20]. It partitions a relation into disjoint sets of tuples based on the value(s) of index attribute(s) and computes an aggregate value for each set of tuples. The SQL standard provides five aggregation functions: `MIN` `MAX`, `SUM`, `AVG` and `COUNT`. Aggregation functions are order-independent and need only a small amount of intermediate storage. Graefe [11] shows that hashing-based techniques out-perform sort-based and nested-loop-based techniques for implementing `group-bys`. Accordingly, we used the hashing-based algorithm from [11] 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

```
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
```

(a) disk-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]);
}
```

(b) host-resident code

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

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: We used `NOWsort` [4] as the starting point for both versions of external sort. NOWsort is based on a long history of external sorting research in the database community (e.g. [2] and [22]) and currently holds the record for the fastest external sort (the Indy MinuteSort record [16]). We used the pipelined version of the two-pass single-node sort [4] for the conventional-disk version. The first phase uses 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 write buffers; a *writer* writes buffers to disk.

[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.

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 used to allow 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 aggre-

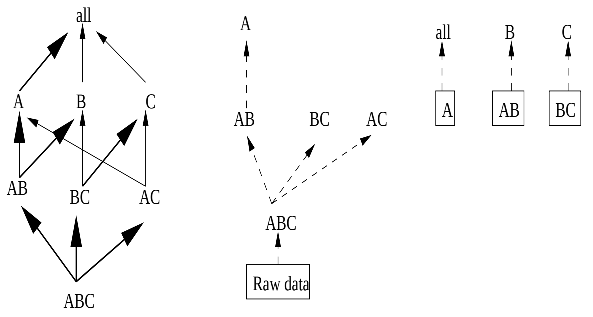


Figure 2: Search lattice and pipelines for the PipeHash algorithm. The bold lines in the search lattice indicate the minimum spanning tree computed by the algorithm using estimated sizes of individual group-bys. The right hand side shows four pipelines. In each pipeline, the data placed in a box at the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

gates [14]. 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 Figure 2 for an example). It schedules the group-bys as a sequence of pipelines; all the group-bys in a pipeline are computed as a part of 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 Figure 2 for examples of pipelines). For individual group-bys, PipeHash uses a hashing-based technique [11]. 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 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 stores the final results 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 [8, 10, 31]. It 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 performs pre-processing and mapping at the disk. In addition, it takes advantage of the fact that the composition is order-independent (it uses a complex `max`-like operation) to perform most of the composition locally as well. 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 [31]. To deal with this, the downloaded disklet allocates space for a sub-image 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 sub-image 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 sub-image is received at the host, it is composed into the final image. Both partitions of the algorithm use the same per-pixel composition operation mentioned above. Figure 3 provides an illustration.

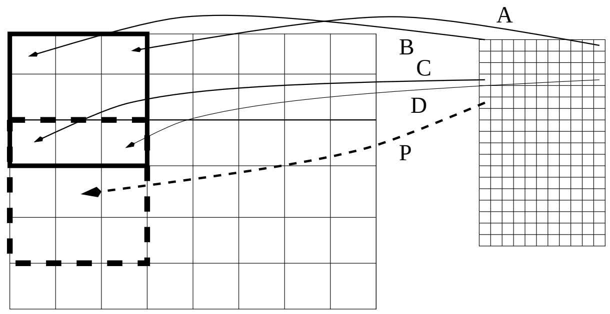


Figure 3: Shifting the subgrid for the `earthsc` disklet. In the figure, values A, B, C, and D map into the original subgrid; P does not. When P is processed, the original subgrid is sent to the host and the subgrid shifted as shown.

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. To explore the scalability of the two architectures, we varied the number of disks in each configuration from 4 to 32. In addition, we evaluated the sensitivity of these results to large variations in interconnect bandwidth (40 MB-400 MB) and host processor speed (350MHz-3 GHz). These experiments help identify the bottlenecks. They also help understand the impact of unbalanced upgrades. Given the long life of large datasets and cost of replacing a large number of disks, it is possible that other parts of the system are upgraded more frequently than the disks.

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 when we did these experiments; 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 [30]. 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 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 running uninterrupted. 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. These runs generated an activity trace that contained time-stamps for scanning the base relation, filling of hash tables, executing the group-by pipeline, and saving the results. This activity trace is replayed by the simulator during simulation.

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. It does not yet support the *skip* operation which allows disklets to selectively receive the data from a stream. We plan to add this in the next version of the simulator.

5.2 Datasets

Select, Group-by: we used two datasets for these algorithms. The smaller dataset was 4 GB (67 million tuples) and the larger dataset was 8 GB (134 million tuples). The tuplesize for both datasets was 64-bytes. 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 assumed a selectivity of 1%.

Sort: we used two datasets with 100-byte tuples and 10-byte uniformly distributed keys. We created these datasets based on the description in [4]. 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

[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.

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 [31]. The size of the 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 4 compares the performance of all six algorithms on conventional-disk and active-disk architectures for 4-disk and 32-disk configurations. We note that for all algorithms and both configurations, active disks outperform conventional disks. Our results indicate that active disks achieve performance improvements between 1.07 times and 3.15 times for 4-disk configurations with `Today`'s components and between 1.18 times and 3.2 times for `Future` components. For 32-disk configurations, active disks outperform conventional disks by between 3 times and 30 times with `Today`'s components; and by between 2.9 times and 29 times with `Future` components. For the core configurations, `select` and `group-by`, which perform little computation per byte of data, achieve small benefits. This is to be expected as both `Today` and `Future` configurations have high-bandwidth I/O interconnects and the conventional-disk algorithms take full advantage of them.

To determine the impact of variation in interconnect bandwidth, we repeated the experiments for the `Today` configuration replacing the 200 MB/s FiberChannel loops with: (1) 40 MB/s Ultra-SCSI and (2) a hypothetical 400 MB/s interconnect. Figure 5 presents the results for 4-disk and 32-disk configurations. It shows that in the presence of a low-bandwidth I/O interconnect, active-disk architectures outperform conventional-disk architectures by a factor of 1.27-3.15 for 4-disk configurations and by a factor of 8.7-30 for 32-disk configurations. This figure also shows that: (1) for all algorithms and both configurations, active-disk architectures with a 40 MB/s interconnect outperform conventional-disk architectures with much faster interconnects; and (2) for algorithms with significant computation per byte transferred (`cube`, `sort`, `conv` and `earth`), conventional-disk architectures cannot take advantage of high-bandwidth interconnects. The range in Figure 5(b) is too large to show the impact of variation in interconnect bandwidth on the performance of active-disk architectures for 32 disks. Figure 6 presents the same information in greater detail. It shows that the performance of for conventional-disk architectures, the interconnect bandwidth is important only for algorithms that perform little computation per byte whereas for active-disk architectures, the interconnect bandwidth is important only for algorithms that redistribute their dataset (e.g., `sort`).

The execution time for all algorithms in our suite increased linearly with an increase in dataset size. This is to be expected as all of the algorithms, except `sort`, are linear. For `sort`, the contribution of the $\log n$ factor is small as the size of the datasets size differ only by a factor of two.

5.4 Scalability

To explore the scalability of the two architectures, we compared the performance of each algorithm on varying number of disks for `Today` configurations. Figure 7 presents the results. We note that increasing the number of disks beyond four (or eight for `select` and `group-by`) 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 [27]. On the other hand, active-disk architectures scale perfectly up to 16 disks. Other than `sort`, other algorithms scale well even up to 32 disks.

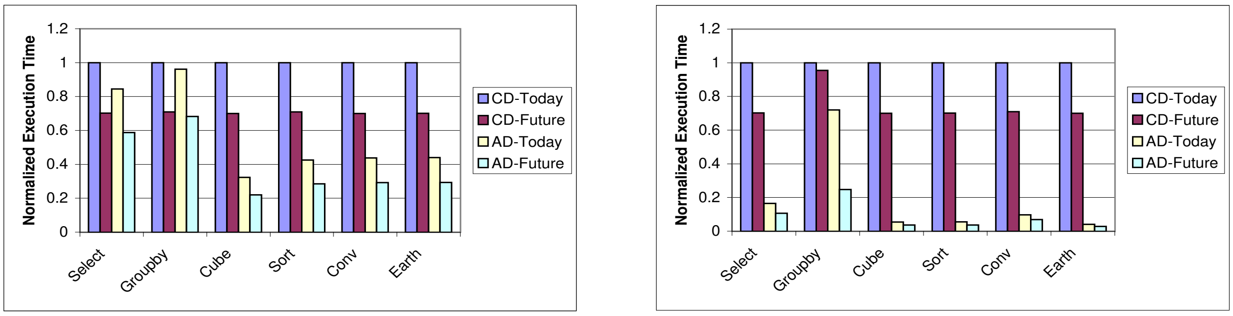
5.5 Impact of upgrading the host processor

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. In order to explore the impact of more frequent upgrades to the host processor, we repeated the experiments for the `Today` configuration replacing the host processor with: (1) a 1 GHz processor and (2) a 3 GHz processor. Note that the disk processor in active-disk architectures for the `Today` configuration is assumed to be 200 MHz. These experiments also allow us to determine if the host processor becomes a bottleneck for active-disk architectures. Figure 8 present the results. It shows that for large configurations, active-disk architectures outperform conventional-disk architectures for "reasonable" relationships between the host processor and the disk processor (a 3 GHz host processor is 15 times faster than the 200 MHz disk processor). We note that a faster host processor allows `sort` to improve its performance on a 32-disk active-disk configuration. This indicates that for large configurations, the host processor can become a bottleneck for algorithms like `sort` which redistribute disk-resident 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 be downloaded and can act in concert and the buffer size can be adjusted to meet the needs of the application.

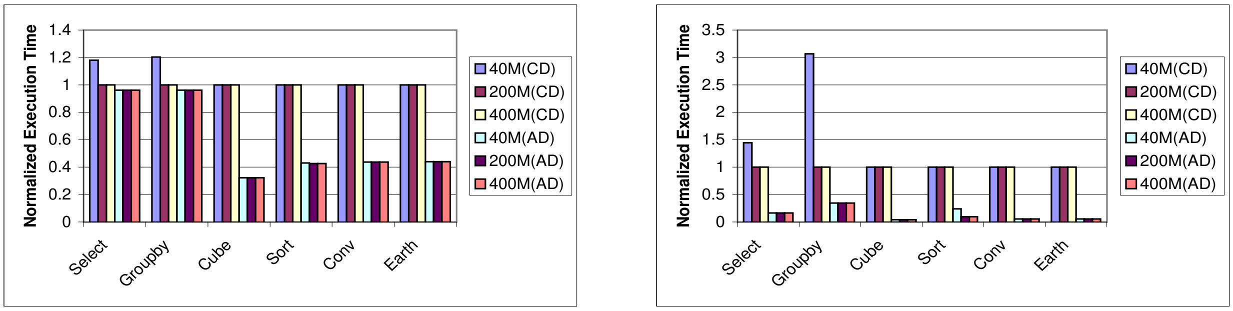
The Active Disk architecture proposed in this paper assumes that disks can communicate only with the host and that all communication between the disks happens via the host. On large configurations, this can lead to the host becoming a bottleneck for algorithms that redistribute their datasets. In our experiments, only `sort` shows this effect for



(a) 4-disks

(b) 32-disks

Figure 4: Comparison of Active Disks with conventional disks. The legend indicates the architecture type (CD/AD) and the configuration (Today/Future). These results are for the smaller datasets.



(a) 4-disks

(b) 32-disks

Figure 5: Impact of variation in the interconnect bandwidth. These results are for `Today` configurations and the smaller datasets.

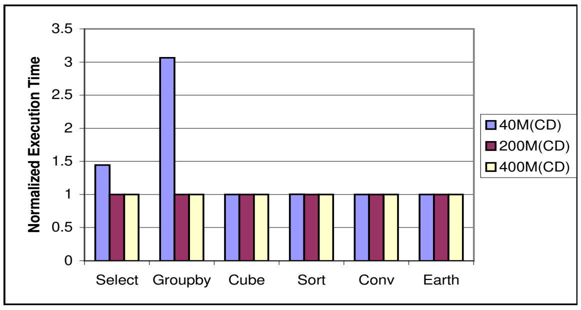
the 32-disk configuration. Note, that since it redistributes almost its *entire* dataset, `sort` is a worst-case example of such algorithms. It is reassuring that such an algorithm can derive a significant performance benefit from Active Disks even for low-bandwidth I/O-interconnects (with an Ultra-SCSI interconnect, `sort` on Active Disks is 2.3 times faster for 4 disks and 4.2 times faster for 32 disks). Other algorithms that redistribute data (e.g. join, materialized view, etc) usually redistribute only a part of the dataset (depending on the selectivity and the kind of join operation). We plan to evaluate the performance of more such algorithms.

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 [26]. 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.

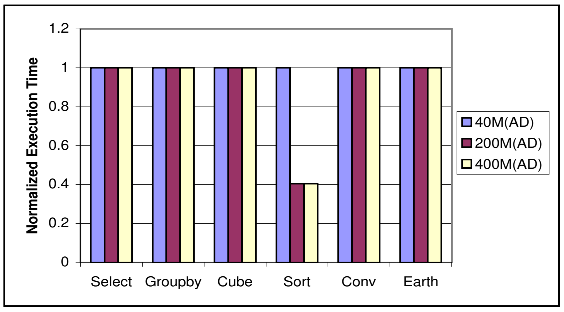
As a part of the *Network-attached Storage Devices* effort,[5] several researchers are exploring the use of processors that are integrated with disk drives. Borowsky et al [5] are investigating the use of disk-processors to implement quality-of-service guarantees for data retrieval. Gibson et al [9] are investigating the use of disk-processors for performing file-system and security-related processing on network-attached disks.

This paper presents one of several independent proposals for architectures whose goal is to scale processing power with dataset size by embedding programmable processors into disk units. Riedel et al [29] propose a model much similar to ours and evaluate its performance for data-mining and image-processing algorithms. They show that these algorithms can achieve significant gains from the use of Active Disks. Keeton et al [18] propose an architecture (IDISK) in which a processor-in-memory chip (IRAM [25]) is integrated into the disk unit and the disk units are connected by a crossbar. They compare this architecture with a conventional

[5] *http://www.nsic.org/nasd/*

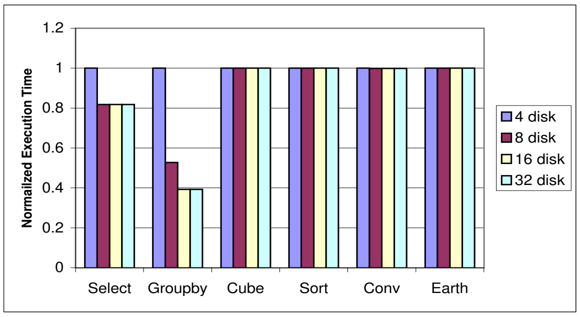


(a) Conventional disks

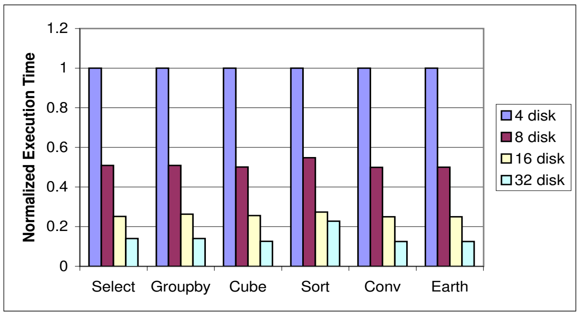


(b) Active disks

Figure 6: Impact of variation in interconnect bandwidth for 32 disks. These results are for `Today` configurations and the smaller datasets. These graphs are a more detailed version of Figure 5(b). Note that the graphs are separately normalized to better show the impact of interconnect bandwidth. Graph (a) is normalized with respect to a conventional-disk architecture with a 200 MB/s interconnect whereas graph (b) is normalized with respect to an active-disk architecture with a 200 MB/s interconnect.

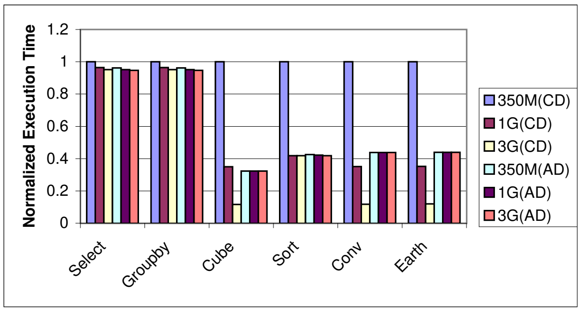


(a) Conventional disks

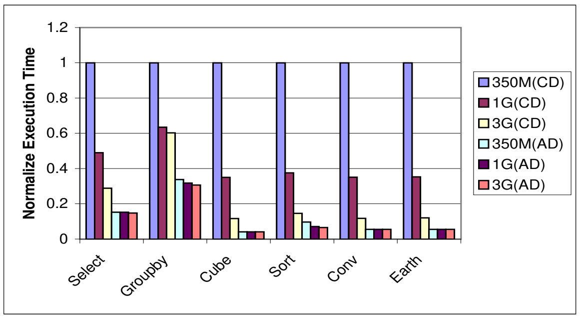


(b) Active disks

Figure 7: Scalability of conventional disks and Active Disks. These results are for `Today` configurations and the smaller datasets.



(a) 4-disks



(b) 32-disks

Figure 8: Impact of variation in the central processor. These results are for `Today` configurations and the smaller datasets.

high-end SMP using analytic techniques. Their results suggest that for a set of database operations, an IDISK-based architecture can be significantly faster than a high-end SMP-based server. Jim Gray [15] proposes an architecture which contains no front-end host and which integrates a state-of-the-art processor, a large memory and a network interface into the disk unit. We plan to evaluate the proposed architectural alternatives and compare their performance in near future.

The growing *processor-memory gap* [33] has lead several proposals for integrating processing logic into DRAM [23, 25]. Chong et al [23] have proposed that data-intensive programs should be partitioned and their data manipulation 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 [25] will probably be the most suitable way of embedding processing power in disk drives.

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. However, it can skip parts of its input streams. These restrictions limit the amount of damage that can be done by a disklet. They also simplify the operating-system support required on disks.

To demonstrate that the Active Disks are suitable for rapidly growing datasets we have presented partitioned versions of several algorithms that process such datasets. We have compared the performance of the partitioned algorithms running on Active Disks to the performance of the original algorithms running on conventional disks.

Our results are encouraging. For all the algorithms used in our evaluation, active-disk architectures outperformed conventional-disk architectures. This performance advantage had two contributing factors: parallelism and avoiding the I/O interconnect. Algorithms that perform large amounts of computation per byte and deliver their data to the host (such as image convolution, datacube and satellite data processing) derive their advantage primarily from parallelism. This is true for algorithms that achieve no data-reduction (e.g., image convolution) or significant data-reduction (e.g., satellite data processing). Algorithms that perform little computation per byte and achieve significant reduction in the data delivered to the host (such as select and group-by) derive their advantage primarily from avoiding the interconnect. Algorithms that redistribute most of their dataset (such as sort) derive their advantage from a combination of the two factors – after redistribution, such algorithms (e.g., sort, database join, materialized views) can usually localize the computation to data on individual disks and can avoid the I/O interconnect completely.

Our results indicate that 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. However, for algorithms that redistribute all or most of their input data, routing all data through the host can become a bottleneck for large configurations. Finally, we have shown that active-disk architectures retain most of their advantage even if the host processor is upgraded more frequently than the disk processors.

We plan to extend this work in four ways. First, we plan to investigate other data-intensive algorithms such as data mining, database join, materialized views and transcoding images. Second, we plan to extend our suite to include index-based algorithms (such as indexed joins and nearest-neighbor search). Third, we plan to evaluate alternative architectures including a network of cheap PCs, IDISKs and the architecture proposed by Jim Gray [15]. Finally, we plan to investigate ways to allow concurrent execution of multiple disklet-groups.

Acknowledgments

We would like to thank Jim Gray for asking us to think *What Happens When Processors Are Infinitely Fast and Storage Is Free?* We would also like to thank him for detailed comments on a previous version of this paper and for pushing us to look further into the future. We would like to thank Huican Zhu for helping us with the graphs. We like to thank Sanjeev Setia, Guy Edjlali and anonymous referees for their comments on different versions of this paper.

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. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*, 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 *Proceedings of the 22nd International Conference on Very Large Databases*, pages 506–21, 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 International Conference on Management of Data*, 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 *Proceedings of the 5th International Workshop on Quality of Service*, 1997.

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

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

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

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

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

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

[12] J. Gray. Some Challenges in Building Petabyte Data Stores. Distinguished Lecture, University of California, Santa Barbara, Oct 1997.

[13] J. Gray. What Happens When Processors Are Infinitely Fast and Storage Is Free? Keynote Speech at the Fifth Workshop on I/O in Parallel and Distributed Systems, Nov 1997.

[14] 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.

[15] Jim Gray. Put EVERYTHING in the Storage Device. Talk at NASD workshop on storage embedded computing[6], June 1998.

[16] Jim Gray. The Sort Benchmark Home Page. Available at *http://research.microsoft.com/research/barc/-SortBenchmark/*, 1998.

[17] HyperMedia Communications Inc., 901 Mariner's Island Blvd, San Mateo CA 94404. The 1998 New Media Hyper Awards. *http://newmedia.com/NewMedia/98/-03/feature/storage.html/*, March 1998.

[18] K. Keeton, D. Patterson, and J. Hellerstein. The intelligent disk (IDISK): A revolutionary approach to database computing infrastucture. Unpublished White paper.[7], Feb 1998.

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

[6] *http://www.nsic.org/nasd/1998-jun/gray.pdf*
[7] *http://www.cs.berkeley.edu/ kkeeton/Papers/idisk98-draft.ps*

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

[21] The MicroWarehouse Online Fixed Drive Catalog. *http://www.microwarehouse.com/MicroWarehouse/-Storage/Fixed_Drives/*, Feb 1998.

[22] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: a RISC machine sort. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minniapolis, May 1994.

[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.

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

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

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

[27] 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.

[28] PC Progress - Memory for the Next Generation. *http://www.pcprogress.com/simms.htm*, 1998. Following link from *http://www.lowerprices.com*.

[29] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, 1998. To appear.

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

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

[32] 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.

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