On Deploying and Executing Data-Intensive Code on SMart Autonomous Storage (SmAS) Disks*

V. V. Dimakopoulos, A. Kinalis, E. Pitoura, and I. Tsoulos

Computer Science Department, University of Ioannina, GR 45110 Ioannina, Greece
{dimako, pitoura}@cs.uoi.gr

Abstract. There is an increasing demand for storage capacity and storage throughput, driven largely by new data types such as video data and satellite images as well as by the growing use of the Internet and the web that generate and transmit rapidly evolving datasets. Thus, there is a need for storage architectures that scale the processing power with the growing size of the datasets. In this paper, we present the SmAS system that employs network attached disks with processing capabilities. In the SmAS system, users can deploy and execute code at the disk. Application code is written in a stream-based language that enforces code security and bounds the code's memory requirements. The SmAS operating system at the disk provides basic support for process scheduling and memory management. We present an initial implementation of the system and report performance results that validate our approach for data-intensive applications.

1 Introduction

There is an increasing demand for storage capacity and storage throughput. This demand is driven largely by new data types such as video data and satellite images as well as the growing use of the Internet and the web that generate and transmit rapidly evolving datasets, take for example the huge amount of data produced by e-commerce transactions [8, 6]. Furthermore, there is growing interest in data mining applications that efficiently analyze large datasets for decision support. Thus, there is a need for storage architectures that scale the processing power with the growing size of the datasets.

Recent disks embed powerful ASIC designs in order to deliver their high bandwidth; such chips are capable of doing considerable processing (for instance, see how complex the SCSI protocol is [2]). In addition, current disks have caches in the order of MB (for example, Seagate's Cheetah has up to 16 MB of cache). This suggests that data can be processed locally at the disk. Such disks are called active [1, 9] or intelligent [7] disks. The idea of data processing at the disk is not a new one. Early proposals include the IBM 360 I/O processors and the

* Work supported in part by the Hellenic General Secretariat of Research and Technology through grant PENED–99/495.

specialized database servers of the 80's [4]. What makes active disk architectures attractive today is that current technology is such that sufficient processing power and memory can be economically embedded in disks. Active disks have been proposed as a cheap replacement to expensive disks; they communicate with the host processor through the local bus (typically SCSI or Fiber Channel). It is envisioned that they will be able to relieve the host processor by acquiring and executing part of the application very close to the data. Such architectures are capable of handling large datasets, since the number of processors scale with the number of disks. In addition, they can effectively reduce the amount of data transfered from the disk to the host processor.

At the same time, distributed file systems are used increasingly nowadays. A good reason is the huge databases maintained by various companies [14]. Thus, a distributed storage architecture is needed to provide efficient and scalable access to data. Such an architecture is provided by NASD [5] coupled with their object-oriented file system for network attached storage.

Active disks have been mainly envisioned as attachments to the local bus of a server [12]. This would however require expensive server architectures especially if one also considers the resources required for supporting multiple interconnected active disks. Furthermore, it is essential, from a practical point of view, to investigate devices that do not require alterations to a given infrastructure. To this end, a smart disk should use part of its processing capabilities to support a simple TCP/IP stack and attach itself on a local network. Apart from the obvious cost reductions, this approach has the additional advantage of true distributed processing, without the bottleneck of a front-end processor, since in a multiple active disk arrangement [12] disks can only communicate with clients through the server processor's connection to the network.

In this paper, we present the SMart Autonomous Storage (SmAS) system. SmAS devices are autonomous (i.e. network-ready) disks but they also have significant processing capabilities like active disks. A SmAS device could possibly include interface support for SCSI or FiberChannel local buses, so as to be attached on a local bus if desired. SmAS disks reduce the communication overhead of transmitting large volumes of data over the network by executing data-intensive applications at the disk. In addition, they release the host processor by undertaking part of the computation.

Our focus is on building a running system. We consider the actual constraints placed on the application programs to be run on the disk and the necessary operating system support for executing them. Portions of the application programs to be executed at the disk, called *filters*, are written in a special-purpose language that ensures code safety, sets bounds on the filter's memory requirements and provides a stream-based interface. Operating system support at the disk is kept at a minimum and includes filter scheduling and memory management. We have tested our initial implementation of SmAS and compare its performance with NFS. The results are encouraging since the measured performance is close to the expected ideal speed-up.

The remainder of this paper is organized as follows. In Section 2, we introduce the SmAS architecture, while in Section 3, we report on the implementation of SmAS. In Section 4, we present related work. Section 5 concludes the paper.

2 The SmAS Architecture

SmAS disks are network-attached disks with processing power. In order for the disk to be employed seamlessly in an existing network, all standard operations are supported (`open()`, `read()`, `write()`, `lseek()` etc.); a SmAS disk can easily replace a conventional NFS server.

The application code consists of two portions: the client-side part, executed at the client, and the SmAS-side part, executed at the smart disk. The latter is called *filter*, and is written in a special-purpose, C-like language. In general, filters are expected to implement processing that reduces the volume of data to be transfered from the disk to the client. For example, instead of transferring the whole file at the client side and performing an SQL-Select there, a filter may perform an SQL-Select locally and thus communicate only the selected data to the application. Filters are cross-compiled at the host disk. Clients can then register the executable code at the client's side. They can then be downloaded (registered) to the smart disk. The client programs invoke filters through a specific series of function calls. The filter code is executed at the disk. Basic minimum operating system support is required at the disk for efficiently executing the filters.

Filter Characteristics. Filters are programs that are executed at the disk. Special requirements imposed on filters include the following: (a) Since disk memory is limited, each filter program should use only a pre-specified portion of this memory. Furthermore, a filter should not dynamically allocate or free memory. (b) Filters should not be allowed to directly read from or write to the disk, to avoid any disk corruption.

We have adopted a stream-based interface for filters similar to the one proposed in disklets [12, 1]. A filter, upon registration, specifies its exact memory requirements. A filter reads from an input stream and writes to an output stream, both of which are controlled by the disk's OS.

Writing and Registering Application Code. Filters are written in a special-purpose language, quite similar to C. This language enforces filters to declare all the variables that they need as global variables, allowing thus for statically calculating their exact memory requirements. Two functions are then expected to be defined: a `filter_init()` function and a `filter_body()` function. The former is used to possibly initialize the filter's global variables. `filter_body()` is the function that implements the core filter processing and has two parameters: an `input_buffer` and an `output_buffer`. Both functions have no local variables; the only variables they can use are the global ones, if any. In addition, they are not allowed to perform any function calls.

A filter is cross-compiled at the client's side. The compiled code can be registered at the smart disk by using a smass_addfilter() call, which results in downloading the executable code of the filter at the disk. Filters thus registered are placed in a filter library.

The client-side part of the application selects the required filter with a smas_usefilter() call; this call identifies the filter to be used and causes the execution of the respective filter_init() function at the smart disk. After that, the client has access to the relevant remote data, one entry at a time, by continuously calling smas_nextrecord(). A call to smas_nextrecord() will ship the next filtered record to the client-side part. What actually happens is that smas_nextrecord() reads a record from the disk and then invokes the filter_body() function, with the read record as a parameter (input_buffer). The filter's function, in turn, decides whether the record meets the criteria, and if so it notifies smas_nextrecord() by returning a value of Send. smas_nextrecord() then sends whatever the filter has placed in its output_buffer to the client. If filter_body() returns a value of NoSend, smas_nextrecord() does not ship anything to the client, loads the next record from the file and invokes the filter again. This is useful when more than one records of the file must be processed before returning control to the client (e.g., computing the maximum value). Fig. 1 provides an example of writing and invoking a filter.

```
/* Define the record structure for easy handling */
typedef struct{ int key; char data[96]; } recstruct;

/* Nothing useful here */
filter_init() {}

/* The filter's body – pass only records with key > 10 */
filter_body(recstruct *in, void *out, int *outbytes) {
   if (in->key <= 10) /* Compare to 10 */
      return (NoSend);
   out = (void *) in; *outbytes = sizeof(recstruct);
   return (Send);
}
```

(a) writing a filter

```
/* Register a filter */
smas_addfilter(SQL_SELECT);

/* Open remote file */
fd = smas_open("zeus.cs.uoi.gr:/pub/testfile", O_RDONLY);
/* Specify the file and filter to use */
smas_usefilter(fd, SQL_SELECT);

/* Now get and process all selected records */
while ( (output = smas_nextrecord(fd)) != EndOfFile )
   process(output);
```

(b) invoking a filter

Fig. 1. Writing and invoking the SQL_SELECT filter.

Disk-side Support. The SmAS operating system (SmAS OS) should be as thin as possible in order to minimize execution overheads and memory costs. Consequently, its functionality should be kept down to a minimum. The only offered services must be networking, process (filter) management, and memory management. Full networking functionality is necessary for attaching the disk to an existing network. Such functionality should be considered rather common place and inexpensive (consider for example the network-ready CD-ROM server by Axis [3]).

A limited form of process management will be required in order to schedule the filter execution. Such a need arises when multiple clients are connected to the disk. Process management should be simple and efficient. However, as discussed in [1], simple strategies like run-to-completion could ultimately limit the disk's performance. In our case, the fact that filters are pre-compiled and embedded in the disk's library allows for accurate estimations of their running times and, consequently, for more informed scheduling strategies, like shortest-job-first [11].

Finally, memory management is probably the most important service since disk's memory is usually its most limited resource. However, memory management is highly simplified due to the filter-based processing: the memory requirements of the filters are known apriori and are satisfied as soon as a filter starts execution. There is no memory swap or process switching. Thus, since the filter cannot allocate dynamically any more memory, it is a simple (and quick) matter to free the allocated memory as soon as the filter completes its execution. We use a first-fit method to allocate memory.

3 Implementation

We have emulated a SmAS device using an old Pentium-based PC, running at 166 Mhz, with 32Mbyte of memory and with a minimized version of Linux as its operating system. The hardware components are analogous to that found inside a present-day disk only much more economical. The SmAS system software (SmASOS) is running as a Linux daemon and awaits at a particular port for a client connection. Communication is handled by the standard `socket` library. At startup, SmASOS acquires a 16Mbyte memory chunk and uses it for implementing its own memory management.

The special-purpose language in which filters are written is an enriched subset of C. We have implemented a compiler for this language, that converts the filter code to standard C code which is then cross-compiled at the client side. Currently, the produced code is in the Linux shared library format which is directly down-loadable to the SmAS disk at run time. To verify the validity of our approach as well as reveal any inefficiencies in the implementation, we have experimented with a number of applications.

Although simple, a particularly illuminating application is the SQL-Select filter shown in Fig. 1. The client utilizes the SQL-Select filter in order to obtain and process relevant records from a file stored at the smart disk. The filtering is done locally at the disk, which returns only the appropriate records to the application, minimizing thus network communication.

```
/* Open the remote file - NFS mounted */
fd = open("/mnt/zeus/pub/testfile", O_RDONLY);

/* Get, filter and process records */
while ( read(fd, buffer, 100) != 0 )
   /* Filter at our (client) side */
   if ( checkcondition(buffer) )
      process(buffer);
```

Fig. 2. Sample application: Traditional NFS approach

We then executed the same application by NFS-mounting the file (Fig. 2) and compared the total running times. If T_{SMASS} and T_{NFS} are the corresponding running times of the two approaches, the observed speedup is defined as T_{NFS}/T_{SMASS} and is plotted in Fig. 3, for various record sizes.

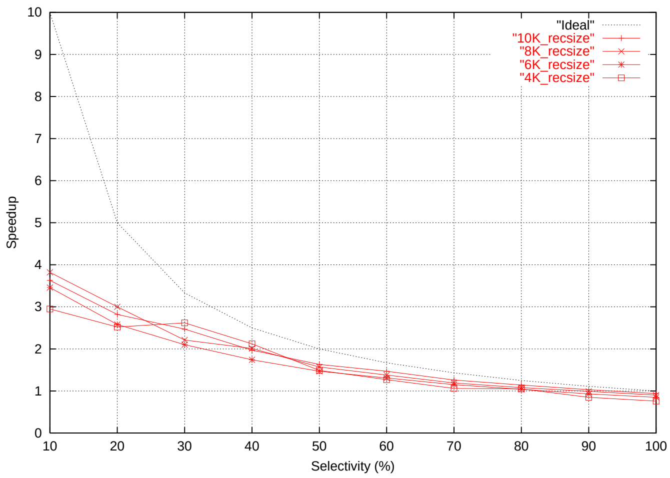

Fig. 3. Speedup with respect to traditional NFS handling

Let s be the *selectivity factor*, that is the probability that a data record is selected by the SQL-Select filter. If a file contains N records then the SMAS code will only deliver sN of them to the application. We generated files that would result in prescribed values for the selectivity factor s (between 10% and 100%). Fig. 3 shows clearly that the SMAS version is able to deliver superior performance, especially for smaller selectivity values. This fact should actually be expected because of the reduced network communication.

The results in Fig. 3 were obtained for file sizes in the area of 100MBytes. In order to determine the socket messaging overheads we experimented with various record sizes. It can be seen from the figure that the performance did not exhibit a wide variance; however, the smaller the record size the smaller the observed speedup. This is due to the headers inserted by the socket library to a message; these extra bytes account for a smaller percentage as the message size grows, thus improving performance.

Assuming that the communication / computation (processing) time ratio per record is greater than one, computation (processing) time, then the total running time should be dominated by the total communication time. Since the SmAS version only communicates a portion s of the total data, while the NFS-mounted file approach communicates all of it to the client, it is expected that (ideally) a speedup of the order of $1/s$ should be observed.

Fig 3 however shows that despite the improvements over NFS, the performance of the SmAS application is actually well below the ideal. This is largely due to the fact that NFS is highly optimized, utilizing prefetching and caching techniques which in effect pipelines computation and communication to a high degree. We are currently optimizing SmASOS and investigate data prefetching techniques. We are confident that performance will move much closer to the ideal. It is also in our future plans to provide kernel support for SmAS in Linux, which will improve performance significantly.

4 Related Work

A stream-based programming model for disklets (i.e., disk-resident code) is presented in [12, 1]. Their active disks are attached to the local bus of the host processor. The disklet programming model is similar to ours. They justify their design decisions by providing a detailed simulation of active disks. In contrast, our focus is on building an actual system which introduces practical restrictions. In [12, 13], an evaluation of the disklet model is provided against two alternative architectures: shared memory multiprocessors (SMPs) and workstation clusters. For most of the applications tested, active disks and clusters significantly outperformed the SMP architecture. The IDISKs (Intelligent disks) architecture proposed in [7] is based on replacing the nodes in a shared-nothing cluster server with intelligent disks that is disks capable of local processing. The main difference in the IDISKs architecture is that the disks are directly connected with each other via switches thus exhibiting much higher bandwidth disk-to-disk communication. The architecture closest to SmAS is the active disks of [9]. The authors of [9] concentrate on developing a number of applications to validate the active disks approach. Their analytical and experimental results promise linear speed-ups in disk arrays of hundreds of active disks for certain data-intensive applications. Instead, the alternative of directly attaching a number of traditional SCSI disks to the local bus of a single server machine caused the server CPU or the interconnect bandwidth to saturate even when a small number of disks (less that ten) was attached. Besides research on active disk, there is some recent interest in shipping application code to the data sources [10]. Application code is in the form of bytecodes. Using bytecodes is not a feasible approach in our case since the limited resources of the disk do not allow for a java execution machine.

5 Summary

In this paper, we introduced the SMAS network attached disk architecture. As compared to a classical file server, an autonomous network-attached device offers significant advantages. First of all, the cost is much smaller, so that one could purchase a number of smart disks for the price of a mid-sized server. Second, SMAS devices have dedicated processing on the disk that provides for efficient distributed processing. A server on the other hand is a general purpose machine that has to deal with many other things apart from file processing. Third, by shipping computation at the disk, network traffic is reduced, since only the relevant data are tranferred to the clients. The results of an initial implementation of SMAS are encouraging and justify our design decisions.

References

1. A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS '98, 8th Conf. on Archit. Support for Programming Languages and Operationg Systems*, pages 212–217, San Jose, California, Oct. 1998.
2. ANSI. Information systems - small computer system interface-2 (scsi-2). Technical report, ANSI X3.131-1994, 1994.
3. Axis Communications. Cd-rom servers, white paper. Technical report, 1996.
4. D. J. DeWitt and P. Hawthorn. A performance evaluation of database machine architectures. In *VLDB '81*, September 1981.
5. G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Sigmetrics '97*, Seattle, Washington, June 1997.
6. J. Gray. What happens when processors are infinitely fast and storage is free? In *5th Workshop on I/O in Parallel and Distributed Systems*, November 1997.
7. K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3):42–52, July 1998.
8. George Lawton. Storage technology takes central state. *IEEE Computer*, 32(11), November 1999.
9. E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB '98*, pages 62–73, New York, USA, August 1998.
10. M. Rodriguez and N. Roussopoulos. Automatic deployment of application-specific metadata and code in mocha. In *7th Conference on Extending Database Technology (EDBT)*, March 2000.
11. A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation. 2nd ed.* Prentice Hall, 1997.
12. M. Uysal, A. Acharya, and J. Saltz. An evaluation of architectural alternatives for rapidly growing datasets: active disks, clusters, smps. Technical report, Dept. of Computer Science, University of California, Santa Barbara, Technical Report TRCS98-27, October 1998.
13. M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *HPCA*, 2000.
14. R. Winter and K. Auerbach. The big time: the 1998 vldb survey. *Database Programming and design*, 11(8), August 1998.