The SMart Autonomous Storage (SmAS) System*

V. V. Dimakopoulos, A. Kinalis, S. Mastrogiannakis, E. Pitoura
Computer Science Department,
University of Ioannina, GR 45110 Ioannina, Greece
E-mail: {dimako, pitoura}@cs.uoi.gr

Abstract

The increasing demand for storage capacity and throughput has generated a need for storage architectures that scale their processing power with the growing size of datasets. In this paper, we give an overview of the SmAS *system that employs network attached disks with processing capabilities. In the* SmAS *system, users can deploy and execute code at the disk. At-the-disk executed application code is written in a stream-based language that enforces code security and bounds the code's memory requirements. The* SmAS *disk's system software 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.*

1 Introduction

There is an increasing demand for storage capacity and storage throughput [4]. 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.

Recent disks embed powerful ASIC designs in order to deliver their high bandwidth; such chips are capable of doing considerable processing. 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, 5] or intelligent [3] disks. They 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.

Active disks have been mainly envisioned as attachments to the local bus of a server [6]. 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.

In this paper, we present the SMart Autonomous Storage (SmAS) system and give an overview of its implementation. SmAS devices are autonomous (i.e. network-ready) disks that have significant processing capabilities like active disks. 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 take processing load off the client by undertaking part of the computation.

2 The SmAS System API

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.

Client applications for such a system consist 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 a *filter*, and is written in a special-purpose, C-like language, which we will present shortly. 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 client's side. Their executable code can then be downloaded ('registered') to the smart disk. The client-side portion of the application invokes filters through a specific series of function calls. The filter code is executed at the disk and relevant data are passed to the client-side program.

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

```
#define FILENAME "smas.cs.uoi.gr:testfiles/test1"
typedef struct
        { int  key; char data[96]; } MyRecord;

main()
{
  int      fd, k, filtid, rsize;
  MyRecord r;

  /* PART1: create a sample file */

  fd = smas_open(FILENAME, O_CREAT | O_RDWR);
  for (k = 0; k < 1000; k++)
  {
    r = randomrecord();   /* Creates a record */
    smas_write(fd, (char *) &r, sizeof(r));
  }
  smas_close(fd);

  /* PART2: filter the records */

  fd = smas_open(FILENAME, O_RDONLY);
  /* Download and utilize a filter */
  filtid = smas_registerfilter(fd, "simpleselect");
  smas_applyfilter(fd, filtid);

  while (smas_nextrec(fd, &r, &rsize) > 0)
    showrecord(r);        /* Get & show records */
}
```

Figure 1: An example application

2.1 An example application

In Fig. 1, we present an example of a simple application to demonstrate the SmAS API. The first part of the application creates a file with 100-byte records at the smart disk using standard file operations.

The second part of the application demonstrates the use of a SmAS filter. The precompiled "`simpleselect`" filter is first registered at (i.e. downloaded to) the smart disk and then utilized (`smas_applyfilter()`) in order to process data.

The application accesses the file records that satisfy the filter's condition one-at-a-time through the repetitive use of `smas_nextrec()`. Each `smas_nextrec()` call returns to the client the next record that satisfies the "`simpleselect`" filter's condition. This is exactly the point where SmAS disks reveal their potential: not all records are passed back to the application; only those accepted by the filter. This has the desired effect of reducing the traffic on the network.

2.2 Filters

Filters are coded in a special-purpose language which resembles C to a high degree. However, there are particular requirements from the filters structure, which are enforced by the filter compiler. In particular, filters are not allowed to utilize pointers and/or dynamically allocate memory; all needed storage must be declared in the form of global variables. This way, apart from the obvious security benefits, we can statically calculate the exact memory requirements of each filter.

```
typedef struct
        { int  key; char str[96]; } myrec;

init() { }

body(myrec r)
{
  if (r.key <= 15)
    pass(r, sizeof(r));
  else
    nopass();
}

tini() { }
```

Figure 2: An example filter

A filter consists of four parts. The first part includes all (if any) variable declarations, which indirectly determine the filter's memory consumption. The second part, `init()` is a function which is executed upon a `smas_applyfilter()` request from the client. Its main purpose is to initialize the filter's variables.

Each subsequent `smas_nextrec()` call from the client's part causes the disk to execute the third part of the filter, its `body()` function. Notice that filters cannot directly access file data; the disk reads the next record from the file and hands it over to filter's `body()` for processing. The filter then decides whether the record is acceptable or not, i.e., whether it satisfies some condition (e.g., in Fig. 2 whether the value of the key is smaller than or equal to 15). If the record is acceptable, the filter specifies exactly what data to send back to the client's side through the use of the `pass()` primitive. If the record is unacceptable, an invocation of the `nopass()` primitive results is a new execution of `body()` on the next record.

Finally, the fourth part of the filter, its `tini()` function, is called by the disk's system software upon meeting an EOF condition.

The `simpleselect` filter for our example application is shown in Fig. 2.

3 SmAS Implementation

We have implemented an initial prototype of a SmAS device and its programming interface (API) using an old Pentium-based PC, running at 166 MHz, with 32MB 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 written in C and is running as a Linux daemon, awaiting at a particular port for client connections. Communication is handled by the standard `sockets` library.

The services offered by SmASOS, except for networking, are simple process (filter) management and memory management. Process management is required in order to schedule the filter execution, since many clients may be actively connected to the SmAS device. A simple run-to-completion policy is currently used, but we also study the effects of other scheduling strategies.

Memory management is probably the most important service since disk's memory is usually its most limited resource. In our implementation, SmASOS acquires, upon startup, a 16MB memory chunk, and uses it for implementing its own memory management. We use a first-fit memory allocation strategy.

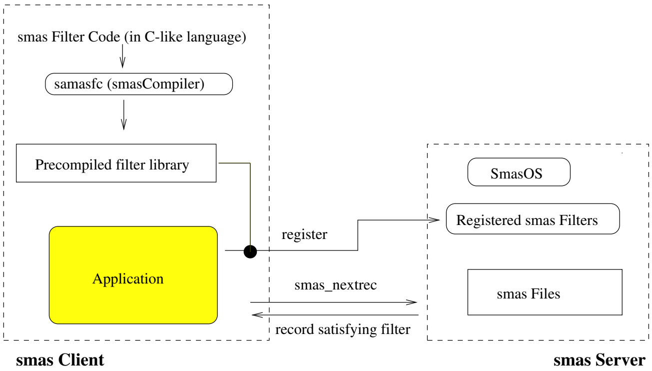

Figure 3: The SmAS Architecture

The SmAS system is complemented with `smasfc`, the filter compiler. The compiler we have implemented, takes filter code written in the C-like language that we described earlier and produces an intermediate file of standard C code. The intermediate file is, then, cross-compiled to produce the final filter code. This filter code is in the Linux shared library format; it can be directly downloaded to the SmAS disk at run time, and be executed on demand.

The overall architecture of SmAS is depicted in Fig. 3.

4 Experiments and Performance

We have tested our implementation exhaustively both for asserting its functional correctness as well as for studying its performance potential. SmAS performance was compared to the traditional NFS approach. The results we report here are (a) for an SQL-select like filter that returns all records that satisfy some condition (such as the example filter in Fig. 2) and (b) for a simple implementation of the `grep` facility which searches a text file for a particular pattern.

4.1 Experiment 1: select

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%).

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. 4, for various record sizes. We experimented for file sizes in the area of 100MB. The performance results show 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.

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.

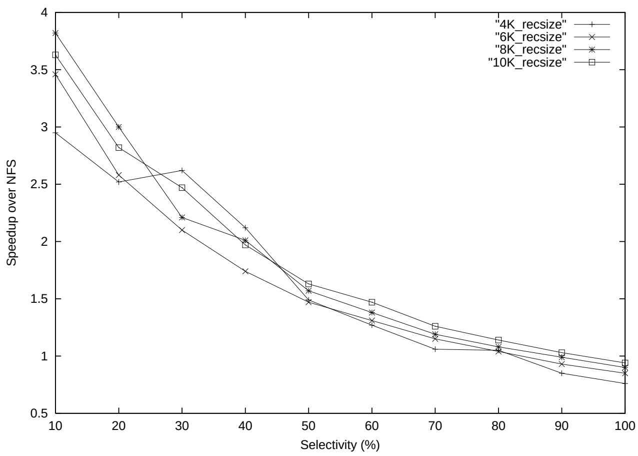

Figure 4: Performance of the select filter

4.2 Experiment 2: the grep facility

We examined files of sizes 100MB and 200MB stored at the server's disk, consisting of 100-character lines. The search pattern was 10-characters long. Our implementation of the `grep` facility was rather straightforward, it did not take advantage of any advanced string matching algorithms. Performance results are depicted in Fig. 5. The experiment demonstrates SmAS scalability: for file sizes of 100MB, SmAS `grep` is about 3 times faster than the system `grep`; for files sizes of 200MB, SmAS `grep` becomes more than 5 times faster than the system `grep`.

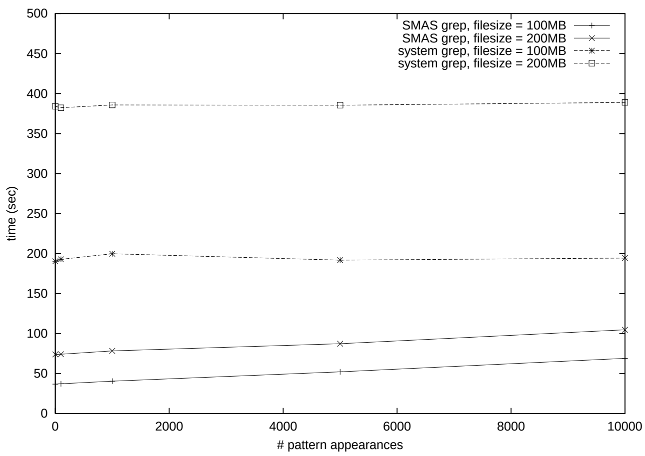

Figure 5: Performance of the grep facility

5 Related Work

A stream-based programming model for disklets (i.e., disk-resident code) is presented in [6, 1, 7]. 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. The IDISKs (Intelligent disks) architecture proposed in [3] 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 [5]. The authors of [5] 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.

6 Summary and Future Work

In this paper, we presented the SmAS network attached disk architecture with programming functionality on the disk. As compared to a classical file server, SmAS 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.

Our work currently focuses on the development of efficient operating system support at the disk. In particular, we are looking into efficient scheduling techniques for the SmAS filters. We are working on a new SmASOS implementation which is based on modifying the GNU Hurd operating system [2]. Hurd is a collection of servers that run on the Mach microkernel to implement file systems, network protocols, file access control, and other features that are implemented by the Unix or similar kernels. We have chosen Hurd over other available kernels such as Linux because Hurd is easily extensible and modular. In the future, we intend to investigate optimizations for pipelining disk access, processing at the disk and communication.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS '98, 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–217, San Jose, California, October 1998.

[2] HURD Operating System http://www.gnu.org/software/hurd/

[3] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3):42–52, July 1998.

[4] G. Lawton. Storage technology takes central state. *IEEE Computer*, 32(11), November 1999.

[5] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB '98, 24th Int'l Conference on Very Large Data Bases*, pages 62–73, New York, USA, August 1998.

[6] 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.

[7] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *HPCA*, 2000.