Energy Savings through Embedded Processing on Disk System*

Seung Woo Son   Guangyu Chen   Mahmut Kandemir   Fehui Li

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16802, USA
Tel: 1-814-863-1047
Fax: 1-814-865-3176
e-mail: {sson,gchen,kandemir,feli}@cse.psu.edu

Abstract— Many of today's data-intensive applications manipulate disk-resident data sets. As a result, their overall behavior is tightly coupled with their disk performance. Unfortunately, most of these applications quickly become disk bound since disk I/O times, the communication latencies, and energy consumption required to transfer disk data to the host machine can be very large. A promising solution to this problem is to embed computational power into the disk storage system. This paper concentrates on such a smart disk based architecture and proposes an automated approach that partitions a given application code between the host machine and the smart disk. The main goal is to perform data filterings, identified at compile time, on the smart disk, thereby reducing the energy spent in communicating disk data to the host unit for processing. To achieve this, the proposed approach uses integer linear programming to identify the code fragments that perform significant data filtering and assigns such fragments to the smart disk for execution. In addition to the communication energy benefits of the proposed approach, we show in this paper that this approach can also help us better exploit the low-power management capabilities provided by the system. Our experiments with four data-intensive applications indicate significant energy savings.

I. Introduction

Applications from different domains that make use of disk-resident data are increasing in both size of the data they manipulate and code complexity. Therefore, disk system performance is critical in shaping both performance and power consumption of such applications. Our analysis of several array-intensive applications shows that a significant fraction of computations that depend on disk data are of *filtering type,* that is, the amount of the input data is larger than that of the output data. Consequently, it is possible to reduce the amount of data to be communicated between the disk system and the host system by executing this filtering type of computations on the disk system. This means performing some form of embedded processing on the disk system, and requires the employment of a processing element on the disk along with its memory. In this paper, we use the term "smart disk" to refer to such a disk storage system equipped with processing capabilities (an embedded CPU) and a memory unit.

Consider, for example, an image processing application such as edge detection. The input to this application is an image (or a series of images) and the output is typically a list of the edges detected. For example, [20] studies real images from IBM Almaden's CattleCam and attempts to detect cows in the landscape above San Jose. The application processes a set of 256 KB images and returns only the edges found in the data using a fixed 37 pixel mask. The intent is to model a class of image processing applications where only a particular set of features such as edges in an image are important, rather than the entire image. This potentially huge data reduction in transforming input to output presents an important opportunity for reducing the amount of data communication between the disk system and the host system.

The prior efforts from the domain of high-performance computing and databases studied this problem of embedded processing on the disk system from the performance angle. We refer the reader to [1, 3, 4, 11, 15, 19, 20, 24] and the references therein. Code partitioning has also been considered in the context of memories that employ embedded processing capabilities [6, 13]. However, to the best of our knowledge, no past study investigated how this embedded processing on disk system can affect power consumption. Also, none of the prior studies discusses a fully automated mechanism for identifying the computations to be performed on the embedded processor in the smart disk. This is the problem attacked in this paper. Specifically, concentrating on a set of array-intensive applications that make frequent use of the disk system, this paper makes the following two contributions:

- We present an approach which, given a data-intensive application code, determines automatically the parts of the application code that can be executed on the smart disk system. In other words, this approach, which is based on integer linear programming (ILP), partitions the application code between the host system and the smart disk system.
- We study the power consumption behavior of this approach under two different scenarios using a simulation-based platform. In the first scenario, the components of the system do not employ any power-saving mechanisms. In contrast, in the second scenario, we assume that both the host processor and the embedded processor on the disk system have low-power operating modes, which can be activated depending on the current loads of the processors. In addition, the disk itself and the interconnect between the disk and the host processor can be shut-down when they are not in use.

To test our approach, we made experiments with four benchmark codes extracted from the SPEC2000 floating-point and Perfect Club suites, which were modified to operate on disk-resident data sets. These experiments reveal that the proposed approach is very successful, under the scenarios mentioned above, in reducing power consumption without noticeably impacting performance. In the first scenario above, our approach reduces the communication traffic between the smart disk and the host, and this leads to savings in power. In the second scenario, our approach helps to increase the idleness of the host processor, thereby enabling a more effective power management via low-power operating modes.

The remainder of this paper is organized as follows. In the following section we briefly describe the disk architecture

*This work is supported in part by NSF Grants 0444158, 0406340 and 0093082 and a grant from GSRC.

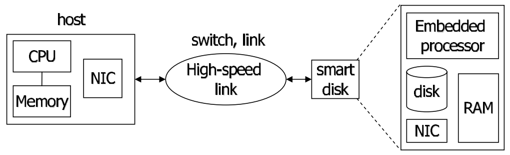


Fig. 1. The smart disk (SD) based system configuration.

equipped with an embedded processor and compare it with a traditional host-based system. In Section III, we present our ILP-based approach and give an example that illustrates how it works. In Section IV, we describe our experimental platform, define the methodology used in the experiments, and present the simulation results. In Section V, we conclude the paper.

II. Smart Disk Based Architecture and Execution Model

Figure 1 shows a smart disk based computing platform. This platform has two major components: a *host* and a *smart disk*. The host is the platform where application execution normally takes place in a system without smart disks (i.e., the one with a conventional disk system). In a smart disk based architecture, on the other hand, the host executes only some parts of the application code (not the entire code), depending on the code partitioning strategy adopted. In general, some parts of the application code are mapped to the smart disk and executed there. This smart disk based storage architecture, which contains a disk system (which can actually be a RAID based disk array), an embedded processor and a memory component, communicates with the host through a communication link, whose exact details are dependent on implementation.

In our execution model, the operation of the smart disk is controlled by the host. Suppose for now that we ran a *code partitioner* and determined the code fragments that are mapped to the host and those mapped to the smart disk (the rest of the paper will discuss our ILP-based code partitioning strategy in detail). When the execution begins, the host first sends the smart disk the code fragments that are mapped to the smart disk. These fragments are typically generated via a cross-compiler at the host system. Following this, the host also sends an *activation signal* to the smart disk, allowing it to start its execution. From this point on, the smart disk and the host can potentially execute in parallel. When the smart disk prepares an output (intermediate or final), it sends the output over the communication link to the host. This requires a synchronization between the host and the smart disk. Similarly, the host can send data to the smart disk during the course of execution.

The main advantage of this type of storage system is to allow the smart disk to execute some parts of the applications that perform data filtering. In this context, a code fragment is said to perform *data filtering* if its output data is much smaller than its input data. For instance, a code fragment that takes a three-dimensional array of size $N \times N \times N$ as input and generates a two-dimensional array of size $N \times N$ as output can be viewed as performing data filtering. If a computation that performs data filtering is executed on the smart disk, one can expect significant reductions in communication energy, as compared to the conventional scenario where the entire computation is executed on the host. Continuing with the example mentioned above, for instance, if we do not employ a smart disk, the three-dimensional input array (a total of N^3 array elements) needs to be transferred from the disk to the host for processing. On the other hand, if we perform filtering on the smart disk, the volume of communicated data is only N^2 elements (i.e., the output array for the code fragment), which represents significant savings in communication energy consumption. There is also a side benefit of employing a smart disk as far as energy saving is concerned. Many hardware components today support several low-power operating modes. Executing some code fragments on the smart disk allows the host CPU to switch itself to a low-power mode if it needs to wait for the results from the smart disk. This can also help increase the overall energy savings. In our experimental evaluation, we consider both these scenarios: one without any low-power management and one with low-power management.

III. Our Approach

We focus on data-intensive applications that manipulate array data. In this section, we explain how we can reduce power consumption by dividing a given code fragment that accesses array data into two parts, host-resident codes and *disklets*[1], i.e., the code portions assigned to smart disks. We want to emphasize that our approach tries to minimize the overall energy consumption of the given application, i.e., it does not only try to reduce the amount of communication. After all, if our objective was just minimizing communication, we could execute everything on the smart disk. However, this would increase program execution time and leakage energy consumption dramatically.

We assume that each array contains a set of *subarrays*. Each subarray $\mathcal{A}_i$ used by a program $\mathcal{P}$ can be represented using tuple $(Q, \alpha_i, \vec{L}_i, \vec{U}_i)$, where Q is the parent array of subarray $\mathcal{A}_i$ (i.e., the array from which $\mathcal{A}_i$ is extracted), and function α_i maps the each subscript vector of subarray $\mathcal{A}_i$ to a subscript vector of parent array Q. Vectors $\vec{L}_i$ and $\vec{U}_i$ are the lower and the upper bounds for the subscript vectors for subarray $\mathcal{A}_i$, respectively. The set of elements captured by subarray $\mathcal{A}_i$ can be expressed as:

$$A_i = \{Q[\alpha_i(\vec{I})] \mid \vec{L} \preceq \vec{I} \preceq \vec{U}\}. \qquad (1)$$

We use an ILP formulation to find the optimal execution strategy for the given program $\mathcal{P}$. We assume that the program $\mathcal{P}$ accesses m subarrays and consists of n loop nests. Before discussing our ILP formulation, let us first define some variables. The values of the following variables can be determined using a compiler by statically analyzing the source code of a given program and/or through profiling:

- $J_{i,j}$: $J_{i,j} \in \{0, 1\}$. If $J_{i,j} = 1$, this indicates that subarrays $\mathcal{A}_i$ and $\mathcal{A}_j$ share some elements, i.e., $A_i \cap A_j \neq \phi$. On the other hand, we have $J_{i,j} = 0$ if subarrays $\mathcal{A}_i$ and $\mathcal{A}_j$ do not share any data elements.
- N_i: the number of iterations for loop nest $\mathcal{L}_i$.
- X_i, E_i: per iteration execution time and dynamic energy consumption for executing loop nest $\mathcal{L}_i$ on the host processor.
- X'_i, E'_i: per iteration execution time and dynamic energy consumption for executing loop nest $\mathcal{L}_i$ on the embedded processor (in the smart disk).
- $W_{i,j}$: this is set to 1 if loop nest $\mathcal{L}_j$ updates the values of some elements of subarray $\mathcal{A}_i$.
- $R_{i,j}$: this is set to 1 if loop nest $\mathcal{L}_j$ reads the values of some elements of subarray $\mathcal{A}_i$.

The values of the following variables, on the other hand, are determined by the ILP solver:

- H_i: $H_i \in \{0, 1\}$. If $H_i = 1$, this indicates that loop nest $\mathcal{L}_i$ is assigned to be executed on the host processor. $H_i = 0$ indicates that loop nest $\mathcal{L}_i$ is to be executed on the embedded processor.

[1]The term is due to Acharya et al [1].

• $M_{i,j}$: $M_{i,j} \in \{0,1\}$. $M_{i,j} = 1$ indicates that subarray $\mathcal{A}_i$ is in the main memory of the host system at the entry of loop nest $\mathcal{L}_j$. $M_{i,j}$ takes the value of 0 if this is not the case.

• $D_{i,j}$: $D_{i,j} \in \{0,1\}$. $D_{i,j} = 1$ indicates that subarray $\mathcal{A}_i$ is dirty at the entry of loop nest $\mathcal{L}_j$. That is, the host processor has updated the values of some of the elements of subarray $\mathcal{A}_i$, and $\mathcal{A}_i$ has not been written back to the disk. Otherwise, we have $D_{i,j} = 0$.

In our formulation, we do not capture the disk energy consumption explicitly, as our approach does not change the original disk I/O activity. However, in our experiments, we also considered the disk power consumption. Let us assume that the total main memory of the host available to the program $\mathcal{P}$ is B. The following expression captures this memory constraint:

$$\textstyle\sum_{i=1}^{m} M_{i,j}|A_i| \le B, \quad \forall j. \tag{2}$$

Since all the subarrays are initially on the disk (i.e., at the beginning of execution), we have:

$$M_{i,1} = D_{i,1} = 0, \quad \forall i. \tag{3}$$

Since a dirty subarray $\mathcal{A}_i$ must be in the main memory, we have the following constraint:

$$D_{i,j} \le M_{i,j}, \quad \forall i, j. \tag{4}$$

If we execute loop nest $\mathcal{L}_j$ on the embedded processor, all the dirty data that may be accessed by $\mathcal{L}_j$ must have been written back to the disk. Therefore, we have the following constraint:

$$(1 - H_j) + \textstyle\sum_{k=1}^{m} D_{i,j} J_{i,k}(R_{k,j} + W_{k,j}) \le 1, \quad \forall i, j. \tag{5}$$

If we execute loop nest $\mathcal{L}_j$ on the embedded processor, any data that may be updated by $\mathcal{L}_j$ cannot be in the main memory of the host. Therefore, we have the following constraint:

$$(1 - H_j) + \textstyle\sum_{k=1}^{m} M_{i,j} J_{i,k} W_{k,j} \le 1, \quad \forall i, j. \tag{6}$$

If we execute loop nest $\mathcal{L}_j$ on the host processor, the size of the data that needs to be loaded into the main memory is:

$$\textstyle\sum_{i=1}^{m}(1 - M_{i,j})R_{i,j}|A_i|. \tag{7}$$

After executing loop nest $\mathcal{L}_j$ on the host processor, the size of the data that needs to be written back to the disk is:

$$\textstyle\sum_{i=1}^{m} W_{i,j}(1 - D_{i,j+1})|A_i|. \tag{8}$$

Assuming the data transfer rate for the communication link is r, the total time required for executing loop nest $\mathcal{L}_j$ on the host processor (including the time to transfer the data over the communication link) can be calculated as:

$$T_j = r(\textstyle\sum_{i=1}^{m}(1-M_{i,j})R_{i,j}|A_i| + \sum_{i=1}^{m} W_{i,j}(1-D_{i,j+1})|A_i|) + N_j X_j. \tag{9}$$

Assuming that the per byte energy consumption of the link is p, the link energy spent for transferring data for executing loop nest $\mathcal{L}_j$ on the host processor would be:

$$E_j{}^* = p(\textstyle\sum_{i=1}^{m}(1-M_{i,j})R_{i,j}|A_i| + \sum_{i=1}^{m} W_{i,j}(1-D_{i,j+1})|A_i|). \tag{10}$$

On the other hand, the total time required for executing loop nest $\mathcal{L}_j$ on the embedded processor would be:

$$T'_j = N_j X'_j. \tag{11}$$

Therefore, the execution time for the entire program can be expressed as follows:

$$T = \textstyle\sum_{j=1}^{n}(H_j T_j + (1 - H_j)T'_j). \tag{12}$$

Also, the leakage energy for the entire system can be written as:

$$E_{leakage} = P \textstyle\sum_{j=1}^{n}(H_j T_j + (1 - H_j)T'_j), \tag{13}$$

where P is the leakage power for the entire system.

The dynamic energy for executing the entire program (excluding the energy spent on the communication link) is:

$$E_{dynamic} = \textstyle\sum_{j=1}^{n}(H_j N_j E_j + (1 - H_j)N_j E'_j). \tag{14}$$

The link energy for the entire program, i.e., the energy spent on the link in communicating data between the host and the smart disk, is:

$$E_{link} = \textstyle\sum_{j=1}^{n} H_j E_j{}^*. \tag{15}$$

Therefore, the total energy consumed by the system for executing program $\mathcal{P}$ can be expressed as:

$$E = E_{link} + E_{leakage} + E_{dynamic}. \tag{16}$$

We use the Xpress-MP solver [17] to determine the values for binary variables H_j, $M_{i,j}$, and $D_{i,j}$ ($i = 1..m$ and $j = 1..n$) such that all the constraints above are satisfied and the overall energy (E) is minimized[2]. It should be emphasized that our solution considers three energy components: the energy for transmitting data through the I/O link between the host and the smart disk, the dynamic energy consumed by both the host system and the smart disk for executing the code fragments mapped to them, and the leakage energy consumed by the entire system during the execution of the application program.

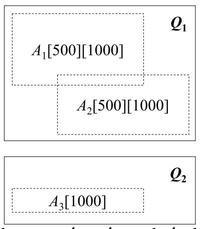
```
Input:
  $H_j$, $M_{i,j}$, and $D_{i,j}$ – the values of binary variables determined by the ILP solver;
  $\mathcal{P} = \{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n\}$ – input program;
Output:
  Transformed program $\mathcal{P}'$

for $j = 1$ to $n$ {
  if($H_j = 1$) {
    // insert code before loop nest $\mathcal{L}_j$
    for $i = 1$ to $m$ {
      if ($M_{i,j} = 0 \wedge M_{i,j-1} = 1$) {
        if ($dirty_i = 1$) {
          $dirty_i = 0$;  output "write $\mathcal{A}_i$ back to disk";
        }
        output "release memory occupied by $\mathcal{A}_i$";
      } else if ($M_{i,j} = 1 \wedge M_{i,j-1} = 0$)
        output "load $\mathcal{A}_i$ into memory";
    }
    $<$ the code for executing loop nest $\mathcal{L}_j$ on the host processor $>$
    // insert code after loop nest $\mathcal{L}_j$
    for $i = 1$ to $m$ {
      if ($W_i, j = 1$) {
        $dirty_i = 1$;
        if ($D_{i,j+1} = 0$) {
          $dirty_i = 0$;  output "write $\mathcal{A}_i$ back to disk";
        }
      }
    }
  } else {
    $<$ the code for executing loop nest $\mathcal{L}_j$ on the embedded processor $>$
  }
}
```

Fig. 2. Rewrite the code of program $\mathcal{P}$ according to the values of H_j, $M_{i,j}$, and $D_{i,j}$.

[2]Please note that the right hand side of Equation (16) is not a linear function of variables H_j, $M_{i,j}$, and $D_{i,j}$. However, Xpress-MP [17] allows the target function to contain products of up to two variables.

```
$\mathcal{L}_1$: for $i = 0$ to 999
      for $j = 0$ to 499
         $A_1[i][j] = g(A_3[i], j)$;
$\mathcal{L}_2$: for $i = 0$ to 999
      for $j = 0$ to 499
         $A_3[i] = A_3[i] + A_2[i][j]$;
$\mathcal{L}_3$: for $i = 0$ to 999
         $A_3[i] = h(A_3[i])$;
```

(a) Code fragment.



(b) Subarrays $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ belong to parent arrays Q_1 and Q_2.

$$J_{i,j} \quad \begin{array}{c|ccc} j \backslash i & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 0 \\ 2 & 1 & 1 & 0 \\ 3 & 0 & 0 & 1 \end{array} \qquad W_{i,j} \quad \begin{array}{c|ccc} i \backslash j & 1 & 2 & 3 \\ \hline 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 \end{array} \qquad R_{i,j} \quad \begin{array}{c|ccc} i \backslash j & 1 & 2 & 3 \\ \hline 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 3 & 1 & 1 & 1 \end{array}$$

i	N_i	X_i	X_i'
1	500,000	100	800
2	500,000	10	80
3	1000	200	1600

(c) Parameters obtained by statically analyzing the source code.

i	H_i
1	1
2	0
3	1

$$M_{i,j} \quad \begin{array}{c|ccc} i \backslash j & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 \end{array} \qquad D_{i,j} \quad \begin{array}{c|ccc} i \backslash j & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{array}$$

(d) The values of the variables determined by our ILP solver.

```
$\mathcal{L}_1$: for $i = 0$ to 999
      for $j = 0$ to 499
         $A_1[i][j] = g(A_3[i], j)$;
   write $A_1$ back to disk;
   signal embedded processor
      to start $\mathcal{L}_2$;
   wait for signal;
   load $A_3$ into memory;
$\mathcal{L}_3$: for $i = 0$ to 999
         $A_3[i] = h(A_3[i])$;
```

```
   wait for signal;
$\mathcal{L}_2$: for $i = 0$ to 999
      for $j = 0$ to 499
         $A_3[i] = A_3[i] + A_2[i][j]$;
   signal end of $\mathcal{L}_2$;
```

(e) The partitioned code generated by our approach. Left: loop nests $\mathcal{L}_1$ and $\mathcal{L}_3$ executed on the host processor. Right: loop nest $\mathcal{L}_2$ executed on the embedded processor in the smart disk.

Fig. 3. An example application of our approach.

Once the values for binary variables H_j, $M_{i,j}$, and $D_{i,j}$ have been determined, our approach rewrites the original code of program $\mathcal{P}$ using the algorithm shown in Figure 2. Specifically, before each loop nest $\mathcal{L}_j$ that is determined to be executed on the host processor, our approach inserts code to release the memory occupied by each subarray $\mathcal{A}_i$ with $M_{i,j} = 0$. If the subarray to be released is dirty, our approach also generates code to write the dirty value back into the disk. If subarray $\mathcal{A}_i$ is written by loop nest $\mathcal{L}_j$, "$D_{i,j+1} = 0$" means that this subarray might be used by a loop nest that is executed on the embedded processor. In this case, our approach inserts code to write subarray $\mathcal{A}_i$ back to the disk immediately after the execution of loop nest $\mathcal{L}_j$. In addition, we also invoke cross compilation for each loop nest that is determined to be executed on the embedded processor.

Figure 3 gives an example application of our approach described above. The original code fragment shown in Figure 3(a) contains three separate loop nests ($\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$), and manipulates three different subarrays ($\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$) belonging to two parent arrays (Q_1 and Q_2), as shown in Figure 3(b). By analyzing this code fragment, the compiler extracts the required parameters, given in Figure 3(c), which are subsequently fed to the ILP solver. The ILP solver then determines the H_j values (see Figure 3(d)), which indicate that, in this particular example, loop nests $\mathcal{L}_1$ and $\mathcal{L}_3$ are to be executed on the host machine and loop nest $\mathcal{L}_2$ is to be executed on the smart disk; i.e., loop nest $\mathcal{L}_2$ is a disklet. Since only $\mathcal{L}_2$ performs data filtering in this example, our approach assigns this loop nest to the smart disk. And, according to the $M_{i,j}$ values obtained through the ILP solver, we know that the result of loop nest $\mathcal{L}_2$, $\mathcal{A}_3$, should be transferred to the host machine. Figure 3(e) gives the partitioned code generated by our approach.

IV. Experimental Platform and Empirical Results

A. Setup and Benchmarks

To evaluate the effectiveness of our approach, we wrote a trace-driven simulator using CSIM [21][3]. In this setup, all the disk I/O and data communication are assumed to be done at a page granularity (default page size of Solaris is 8KB). The cycle time for each instruction type (e.g., load/store, arithmetic, etc.) is obtained from the processors' manual [9, 10]. The simulator generates energy consumption and performance statistics. The energy statistics are calculated based on the figures extracted from the datasheet of each system component or from the previously-published studies [5, 7, 8, 9, 10, 12, 14, 16, 18, 25], and are given in Table I. More specifically, our power model is as follows. While processing the given input traces, the simulator keeps track of time values (stamps) for all activities involved in processing each trace. These recorded time values indicate the states of each component (i.e., when a particular component is used and how long it is used) throughout the entire simulation time, and then we calculate power values from these determined states of each system component. The required hardware components for this calculation are different for each scheme we considered, which will be explained later in this section. In this study, we assume that each component in our target system has its own power transition state diagram. That is, each component has at least one low-power operating mode (state) as well as active and idle modes, and it takes a certain amount of time and energy to transition from one state to another. Based on these energy and time values, one can determine the *break-even threshold*, i.e., the minimum amount of idle time required to compensate the cost of transitioning a given component into a low-power mode, and this threshold is used in this work in deciding whether it makes sense to put the component into a low-power mode when it becomes idle.

The different components have different low-power modes and usually different names for these modes. For example, a modern server disk has typically only one low-power mode, called shut-down, whereas a DRAM has three low-power modes, named standby, napping, and power-down. So, when there is no confusion, in the rest of the paper, we use the term "low-power mode" to denote only one of the low-power modes that each hardware component provides, specifically, the one which consumes the lowest energy.

Table II gives the set of data-intensive applications, used in this study. We chose these benchmarks from the CFP2000 [22] and Perfect Club benchmarks [2], and made the array data manipulated by the benchmarks disk resident. As a result, each array reference causes a disk access unless the requested block is captured in the buffer cache. Also, to complete the simulation within a reasonable amount of time, we concentrated on the most dominant loop nests in terms of the cumulative I/O times and the amount of data manipulated. The second column of Table II gives the total dataset size manipulated by each benchmark, and the next two columns give the total energy consumption and execution time, respectively, for each application, when all computations are executed in the host (this is the HOST version, as will be described shortly). *The energy and performance numbers presented in the rest of the paper are normalized with respect to the values listed in these two columns of Table II.* The fifth column gives the contribution of the energy consumed in the communication link between the host and the smart disk, which takes a significant fraction of the total energy consumption, as can be seen from the table

[3]This should not be confused with the gate-level simulator of the same name [23].

TABLE I
DEFAULT SIMULATION PARAMETERS.

Parameter	Value
Host Processor	
Model	Intel P4 2.0 GHz
Power (active/idle/standby)	100.4/75.3/0.0525 W
Power (standby → active)	0.1 J
Time (standby → active)	1 ms
Power (active → standby)	5.3 uJ
Time (active → standby)	70.38 ns
Embedded Processor	
Model	StrongArm 200 MHz
Power (active/idle/standby)	400/50/0.16 mW
Power (standby → active)	0.064 J
Time (standby → active)	160 ms
Power (active → standby)	0.036 mJ
Time (active → standby)	90 us
Memory	
Model	Rambus DRAM
Capacity	32MB for smart disk and 1GB for host
Power (active/standby)	300/3 mW
Power (standby → active)	15 mW
Time (standby → active)	6000 ns
Power (active → standby)	15 mW
Time (active → standby)	8 memory cycle
Disk	
Model	IBM Ultrastar 36Z15
Storage Capacity	18 GB
RPM	15,000
Average seek time	3.4 msec
Average rotation time	2 msec
Internal transfer rate	55 MB/sec
Power (active/idle/standby)	13.5/10.2/2.5 W
Energy (idle → standby)	13 J
Time (idle → standy)	1.5 sec
Energy (standby → active)	135 J
Time (standby → active)	10.9 sec
Interconnects (Link)	
Model	Infiniband 1X
Bandwidth	2.5G (1X)
Energy	10.21 (pJ/bit)
Power (standby)	ε
Time (standby ↔ active)	800 ns
Energy (standby ↔ active)	0.002 mJ
Switch	
Model	IBM Infiniband 1X
Power (active/standby)	11 / 2 W
Energy (active → standby)	0.18 mJ
Time (active → standby)	0.09 ms
Energy (standby → active)	0.32 J
Time (standby → active)	0.16 sec

(over 20% for all four benchmarks). This suggests that reducing the link power consumption can have a significant impact in practice on overall system power consumption. The last column gives the percentage of the application code mapped to the smart disk (i.e., the disklets) after applying our ILP-based code partitioner.

To quantify the benefits obtained from our approach, we implemented and conducted experiments with different schemes:

• HOST: This is the version where all the computations are performed on the host machine. This scheme requires all system components shown in Table I except the embedded processor and memory in the smart disk, which remain in the idle states. Since the data is stored in the smart disk, this scheme incurs a significant data communication from the smart disk to the host. The energy and performance results with this version are given in Table II.

• SD: This version represents the other extreme, and performs all the computations on the smart disk, that is, the remaining components remain in idle state. Unlike the HOST scheme, it does not incur any communication energy cost due to the disk accesses since the disk is a local resource from the viewpoint of the smart disk. However, this version can increase leakage consumption dramatically over the HOST version, due to the increase in execution time. After finishing execution, the results required by the host are transferred to the host machine, and the communication between the smart disk and the host machine occurs only at this point.

TABLE II
BENCHMARKS AND THEIR CHARACTERISTICS.

Name	Total Data (MB)	Base Energy (J)	Execution Time (sec)	Link Energy	% of Code on on Smart Disk
swim	22.1	736.6	4.4	23.9 %	59.0 %
apsi	2.9	101.6	0.6	23.8 %	74.0 %
mgrid	80.7	2707.1	16.2	23.6 %	54.0 %
bmcm	10.3	457.5	2.6	22.3 %	28.3 %

• OPT: This is an optimized version in which code fragments to be executed on the host and the smart disk are determined by our ILP-based partitioner described in Section III. The computation occurs in both the host and the smart disk, which requires all system components to execute this scheme. When one side is in use during computation, the other side will remain in idle state. In the general case, some code fragments are executed on the host machine and the others on the smart disk. The communication energy is spent whenever there is a communication between the two code fragments mapped onto the different places.

These three schemes (HOST, SD, and OPT) do not make use of any low-power modes of any component in the system. Apart from these three schemes, we also implemented an energy-optimized version, called EOPT, which can be used in conjunction with the above three schemes. In the EOPT scheme, each system component, e.g., CPU, memory, interconnect, etc., can be in a low-power mode when it is not in use. The decision to place a component in the low-power mode is based on the break-even time of each component, as explained earlier. That is, if the idle period of a given component is longer than its break-even time, the component is placed into the low-power mode. Otherwise, it remains in the idle/active state, i.e., without any power management. Therefore, the schemes in conjunction with EOPT do not change their original execution times. The purpose of our experiments with EOPT is to see how our approach interacts with low-power modes. Combining EOPT with the three schemes described above, we conducted experiments with a total of six different schemes: HOST, SD, OPT, HOST+EOPT, SD+EOPT, and OPT+EOPT. Among the different schemes used in this work, the largest ILP solution time was taken by the OPT+EOPT scheme and was around 172 seconds for the *mgrid* benchmark.

B. Results

The graphs in Figure 4 give the total energy consumption of our benchmarks under the different schemes described earlier. As mentioned earlier, all the results are normalized with respect to the HOST version. One can make several observations from these results. First, for *mgrid* and *bmcm*, executing all the loop nests in the smart disk significantly increases overall energy consumption. This is because the computation power in the embedded processor in the smart disk is much less than that in the host processor. Second, for *swim*, there is not a significant difference whether all code fragments are executed in the host or in the embedded processor. This is because the communication reduction and the increase in computation time when all computations are assigned to the embedded processor balance each other. Lastly, even if we do not employ any shutdown policies in any component, the OPT version results in significant amount of energy savings (23% on average). This shows that our ILP-based approach successfully partitions the computations across the host and the smart disk in an energy-efficient fashion.

It is to be noted, however, that the reduction in the amount of data to be communicated, provided by our approach, might affect the potential energy savings when the system components employ low-power operating modes. The potential en-

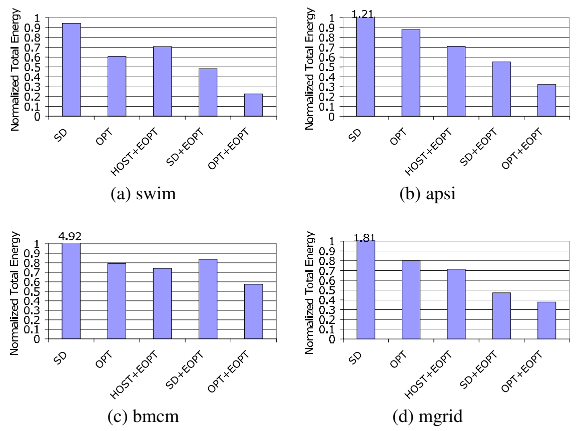


Fig. 4. Normalized total energy consumption of each benchmark.

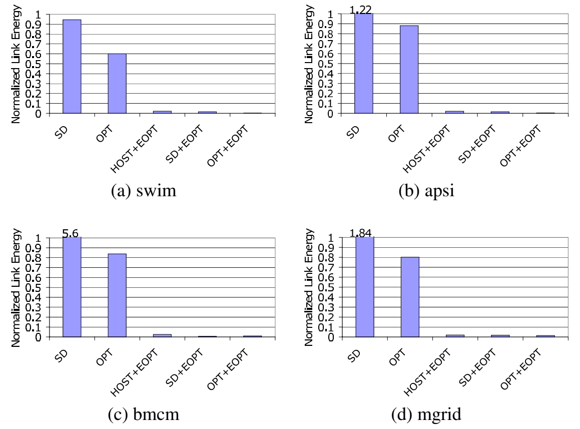


Fig. 5. Normalized link energy consumption of each benchmark.

ergy savings in this case are captured by the right three bars, HOST+EOPT, SD+EOPT, and OPT+EOPT, in each bar-chart shown in Figure 4. One can see from these results that, if we can exploit the low-power mode that each component provides whenever possible, the resulting energy savings are really significant. Overall, the results presented in Figure 4 indicate that our approach reduces energy in both the scenarios studied. These benefits are achieved not only due to the reduced volume of communication between the smart disk and the host, but also due to the increased chances for the host to shut down. To see the details of the energy savings in the communication link, we also collected statistics on link energy consumption with these six different schemes we experimented. The results are given in Figure 5. One can see from these results that most of the energy consumption in the communication links can be eliminated if we shut down the links when they are not in use. Overall, our results indicate that the compiler algorithm presented in this paper can be very useful in practice, and the best energy savings are achieved when our approach is used in conjunction with the power saving mechanisms provided by each component. We also measured the impact of our approach on original execution cycles, however, we do not present the detailed results due to lack of space. We found that the OPT and OPT+EOPT schemes improve the original execution cycles by 23% on average.

V. Conclusions

Many large-scale applications are data-intensive in nature and require manipulation of huge data sets such as multi-dimensional scientific data, image files, satellite data, database tables, and digital libraries. Apart from the high computational requirements, these applications typically involve the transfer of large amounts of disk-resident data back and forth between the secondary storage devices and the processing units. Observing that a large fraction of these computations are of filtering type, this paper proposes and evaluates an ILP-based approach that partitions an application code between the host system and the disk system (equipped with an embedded processor). We test the behavior of our approach using a set of array-intensive benchmarks that frequently exercise the disk system. Our results show that the proposed partitioning approach reduces power consumption significantly.

References

[1] A. Acharya, M. Uysal, and J. H. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, 1998.

[2] M. Berry and et al. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[3] G. Chen, G. Chen, M. T. Kandemir, and A. Nadgir. Compiler-Based Code Partitioning for Intelligent Embedded Disk Processing. In *Languages and Compilers for Parallel Computing*, pages 451–465, 2003.

[4] S. Chiu, W. keng Liao, and A. N. Choudhary. Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads. In *International Conference on Computational Science*, pages 230–241, 2003.

[5] C. Eddington. InfiniBridge: An InfiniBand Channel Adapter with Integrated Switch. *IEEE Micro*, 22(2):48–56, 2002.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[7] IBM hard disk drive. Ultrastar 36Z15, April 2001.

[8] InfiniBand Trade Alliance. The InfiniBand Architecture. `http://www.infinibandta.org`.

[9] Intel. *Intel StrongARM SA-1100 Microprocessor - Developer's Manual*, June 2000.

[10] Intel. *Intel Pentium-4 Microprocessor at 2GHz - Datasheet*, 2001.

[11] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

[12] E. J. Kim, K. H. Yum, G. M. Link, N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, M. S. Yousif, and C. R. Das. Energy Optimization Techniques in Cluster Interconnects. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 459–464, 2003.

[13] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and H. S. E. Pursuing a Petaflop: Point designs for 100TF Computers Using PIM Technologies. In *6th Symp. on Frontiers of Massively Parallel Computation, Annapolis*, pages 25–31, October 1996.

[14] X. Li, Z. Li, F. M. David, P. Zhou, Y. Zhou, S. V. Adve, and S. Kumar. Performance Directed Energy Management for Main Memory and Disks. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–283, 2004.

[15] G. Memik, M. T. Kandemir, and A. N. Choudhary. Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads. In *Proceedings of the International Conference on Parallel Processing*, pages 335–, 2000.

[16] S. S. Mukherjee, P. J. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. *IEEE Micro*, 22(1):26–35, 2002.

[17] D. Optimization. Modeling with Xpress-MP, December 2001.

[18] Rambus. RDRAM. `http://www.rambus.com`, 1999.

[19] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active Disks for Large-Scale Data Processing. *IEEE Computer*, 34(6):68–74, 2001.

[20] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the International Conference on Very Large Data Bases*, pages 62–73, 1998.

[21] H. Schwetman. CSIM19: A Powerful Tool for Building System Models. In *Proceedings of the 33nd conference on Winter simulation*, pages 250–255, 2001.

[22] SPEC. Specfp 2000. `http://www.specbench.org/cpu2000/CFP2000/`, 2000.

[23] P. Stenius. Csim - a simple hp48 circuit simulator for educational purposes, 1992.

[24] M. Uysal, A. Acharya, and J. H. Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 337–348, 2000.

[25] H.-S. Wang, L.-S. Peh, and S. Malik. A Power Model for Routers: Modeling Alpha 21364 and InfiniBand Routers. *IEEE Micro*, 23(1):26–35, 2003.