CHAPTER 4

SELECTION OF DISK-SEEK ALGORITHM USING BACK PROPAGATION NEURAL NETWORKS WITH MLM ALGORITHM

4.1 Introduction

Excessive power consumption is becoming a major barrier to extracting the maximum performance from high-performance parallel systems available in the data centre. Therefore, techniques oriented towards reducing power consumption of such systems are expected to become increasingly important in the future. Previous work on disk power management focuses primarily on hardware based schemes. However, since disk access pattern, i.e., the order in which disks on a system are accessed is mainly shaped by the program code access pattern and disk layout of data, software techniques can also play a critical role in disk power management. From the software angle, there are two major parameters that can be tuned for low power: the code structure of the application program and the disk layout of data.

4.2 System Architecture

This research focuses on array based scientific applications executing on parallel architecture which uses file striping technique to divide the large data into small portions and stores these portions on separate disks in a round robin fashion. It is assumed that each data array manipulated by the application is stored in a separate file in the I/O subsystem. The architecture of the proposed system of this research work is given in Figure 4.1.
Figure 4.1 System Architecture
The user request for the data is given to the application program running in the server. The operating system gives the present status of the head position of the disk subsystem. The starting head position, ending position and the distance of the disk head were given to the back propagation network. The Modified Levenberg Marquardt algorithm is used to train the Back Propagation Neural Network (BPN with MLM) for selecting the suitable disk-seek algorithm based on the given input set. The data access pattern and the disk layout are used to construct the Disk Access Pattern (DAP). After extracting the disk access pattern and the selected disk seek algorithm, the compiler can restructure the source code to reduce the power consumption by inserting clear disk power management calls in suitable place of the source code.

4.3 Disk Power Management

4.3.1 Traditional Disk Power Management (TPM)

In traditional disk power management techniques (TPM), if the detected disk idle period is longer than a certain amount of time, called the *idleness threshold*, the disk is spun down to the low-power mode. The disk remains in the low-power mode until it receives the next request. Note that this strategy typically incurs performance slowdown because the disk should first spin up to service the upcoming request. The time it takes to spin up/down a disk is called the *spin-up/down time*. Therefore, in TPM, choosing the idleness threshold by making use of either fixed or adaptive threshold based strategies is crucial in managing both disk energy and performance. While TPM is an effective approach in the domain of laptop/desktop systems, but it is not an appropriate choice for large servers and high-performance parallel systems available in the data centre.

4.3.2 Dynamic RPM (DRPM)

Since exploiting idle time is hardly a viable option in the server class disks, Dynamic RPM (DRPM) is used in which the disk hardware/controller provides several RPM steps. Note that, the higher RPM a disk spins at, the faster it services
the I/O requests, and the higher power it consumes. An application that executes on a platform with DRPM capability can select disk speed dynamically at runtime to achieve the optimal balance point between energy consumption and execution time. Note that DRPM also incurs performance penalty because a lower RPM can potentially degrade response time. This can occur because a hardware-based DRPM strategy (like TPM) works with an estimation of disk idle times. If the estimation is not accurate, DRPM can select a wrong disk speed.

4.4 Related Works

4.4.1 Disk Power Management States

It is assumed that each disk is equipped with a timer-based power management capability. Figure 4.2 depicts the transitions between the different states supported by the disks considered in this thesis. The labels attached to the arcs in Figure 4.2 indicate how the transitions are triggered.

![Diagram of disk power management states](image)

**Figure 4.2 Different disk states and transitions of computing energy consumption.**

In this mechanism, when the current access to a disk is completed, the disk transits to the idle state. If it remains in the idle state for a certain amount of time, it is spun down. Thus the disk is placed into the low-power operation mode. Then the disk transitions back to the active mode by spinning up when a new request to it is
made. Note that this model represents one of the simplest mechanisms that can be supported by a server disk that allows power management. The important point to emphasize here is that, since spinning-down and spinning-up take both extra time and energy, they need to be minimized. Therefore, one would prefer, from both performance and power consumption angles, a few long idle periods over numerous short idle periods. A profile-driven approach is used here to increase the duration of idle periods, thereby increasing the chances for this power management scheme to be applicable and successful. It is necessary to set the layout parameters for each disk-resident array manipulated by the application code.

4.4.2 Disk Layout Extraction

File striping is a technique that divides a large data into small portions and stores these portions on separate disks in a round-robin fashion. The striping of array data structure is as shown in Figure 4.3. This permits multiple processes to access different portions of the data concurrently without much disk contention.

![Stripe Array Diagram](image)

**Figure 4.3 Striping of Array Data Structure**
While striping can be performed manually, many file systems today provide automatic support for it. In this work, the disk layout of an array is represented using a triplet of the form

(start disk, stripe factor, stripe size).

The first component, start disk, in this triplet indicates the disk from which the array is started to get striped. The second component, stripe factor, gives the number of disks used to stripe the data, and the third component, stripe size, gives the stripe (unit) size used. Several current file systems and I/O libraries for high performance computing provide APIs to convey them the disk layout information when the file is created.

![Diagram of disk layouts](image)

**Figure 4.4 Two different example disk layouts**

*Left (d0,6,S) and Right (d4, 3, 2S)*

For example, in Parallel Virtual File System (PVFS), one can change the default striping parameters by setting base (the first I/O node to be used), pcount
(stripe factor), and ssize (stripe size) fields of the pvfs filestat structure. Then, the
striping information defined by the user via this pvfs filestat structure is passed to
the pvfs open() call’s parameter. Two example disk layouts for two-dimensional
disk-resident arrays are depicted in Figure 4.4. The first layout (i.e., the one for
array U) is (d0, 6, S), whereas the second layout (i.e., the one for array V) is
(d4, 3, 2S).

The three layout parameters is to be determined for each disk resident array
that needs to be created by a given application program. It needs to be noted
however that this has to be done in a coordinated fashion by considering all the
disk-resident arrays in the application. This is because the different disk-resident
arrays can potentially share the same set of disks and determining their layouts in an
independent fashion that can lead to unpredictable results (e.g., due to irregular disk
access patterns) at runtime as far as saving disk power is concerned.

4.4.3 Data Access Pattern

The user request along with the application program is analyzed to get the
data access pattern. The data access pattern is of the form (A1, A2, ..., An). Each
array access Ai in this sequence has the form (X, a, t), where X is the id of the disk-
resident array, a is the offset of the accessed array element within the array, and t is
the time stamp. The time stamp of an array access is the time since the start of the
program after deducting the I/O time spent in the disk accesses. The proposed
approach determines a single component of a disk layout at a time. More specifically, it
first determines the stripe factor for all arrays and then the stripe size for all arrays.
Then, based on these, it determines the start disk for all arrays.

4.4.4 Disk Access Pattern (DAP) Construction

One of the requirements for being able to use a compiler in reducing disk
power consumption is to capture how parallel disks are accessed at a high level (i.e.,
source code level). The term disk access pattern is the high-level information on the
order in which parallel disks are accessed by a given application code. This order is important since it determines, for each disk in the system, active and idle periods, which is the primary information used for power management.

From the selected disk seek algorithm, it is easy to construct the disk access pattern. The data access pattern indicates the order in which the different array elements are accessed. Disk access patterns can be constructed at the loop iteration, loop nest, procedure, or even larger granularities.

As an example, in Figure 4.5, array U1 is striped over all four disks. Assuming that the stripe size is S and the total array size is 4S (for illustrative purposes), the disk layout of this array can be expressed as \((0, 4, S)\)

![Figure 4.5 Disk layout of arrays U1 and U2](image)

For \(i = 1, 2S\)

\[\ldots U1[i], U2[i] \ldots\]

For \(j = 2S + 1, 4S\)

\[\ldots U1[i], U2[i] \ldots\]

Figure 4.6 Original code fragment
<table>
<thead>
<tr>
<th>Disk</th>
<th>Nest</th>
<th>Iteration</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk0</td>
<td>Nest 1</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk0</td>
<td>Nest 1</td>
<td>S</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk2</td>
<td>Nest 1</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk2</td>
<td>Nest 1</td>
<td>S</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk2</td>
<td>Nest 2</td>
<td>2S+1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk2</td>
<td>Nest 2</td>
<td>3S</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk1</td>
<td>Nest 1</td>
<td>S+1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk1</td>
<td>Nest 1</td>
<td>2S</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk3</td>
<td>Nest 1</td>
<td>S+1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk3</td>
<td>Nest 1</td>
<td>2S</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk3</td>
<td>Nest 2</td>
<td>3S+1</td>
<td>Active</td>
</tr>
<tr>
<td>Disk3</td>
<td>Nest 2</td>
<td>4S</td>
<td>Idle</td>
</tr>
</tbody>
</table>

**Figure 4.7 Disk Access Pattern**

To illustrate the process of identifying the disk access pattern, let us consider the code fragment in Figure 4.6. During the execution of the first loop nest, this code fragment accesses the array elements U1[1], U1[2], \ldots, U1[2S] and U2[1], U2[2], \ldots, U2[2S]. Consequently, for array U1, access the first two disks (disk0 and disk1); and for array U2, access only the third disk (disk2). Note that, the several current file systems and I/O libraries for high-performance computing support calls available to convey them the disk layout information when the file is created. For example, in PVFS, change the default striping parameter by setting base (the first I/O node to be used), pcount (stripe factor), and ssize (stripe size) fields of the pvfs filestat structure. Then, the striping information defined by the user via this pvfs filestat structure is passed to the pvfs open() call’s parameter.

The DAP lists, for each disk, the idle and active times in a compact form. An entry for a given disk looks like

\[
\text{< Nest 1, iteration 1, idle >}
\]

\[
\text{< Nest 2, iteration 50, active >}
\]

\[
\text{< Nest 2, iteration 100, idle >}
\]
From this example DAP, the disk in question remains in the idle state (not accessed) until the 50th iteration of the second nest. It is active (used) between the 50th iteration and the 100th iteration of the second nest, following which it becomes idle again, and remains so for the rest of execution. For the example code fragment in Figure 4.6 and the disk layouts illustrated in Figure 4.5, Figure 4.7 gives the DAPs for each of the four disks in the system.

4.5 Overview of Back Propagation Neural Network

A popular type of neural network is the multilayer perceptron, in which neurons are organized into at least three layers as shown in figure 4.8. Each layer is (usually) fully interconnected to its adjacent layers. Multilayer perceptrons can be viewed as performing a functional mapping from an input space to an output space. One type of multilayer perceptron is the back propagation neural network. A back propagation network with a single hidden layer of processing elements can model any continuous function to any degree of accuracy (given enough processing elements in the hidden layer).

![Figure 4.8 Back Propagation Network](image)
The first—forward/activation—pass presents an input vector to the first layer of the network, which then propagates through the network one layer at a time. For a single non-input layer, this propagation requires each neuron in that layer to compute a weighted sum of its incoming signals to yield a net input and apply a continuous non-linear (usually sigmoid in shape) activation function to determine its output value. The output vector of the network is the activation vector of the final layer of neurons.

The second—backward/weight adjusting—pass attempts to correct any error made with respect to the desired mapping. The overall network error for the given input vector is computed by comparing the output vector to a given teacher. Each hidden layer must then compute the contribution of its neurons to this error. This is achieved by each neuron in a layer calculating its error from a weighted sum of the error of the succeeding layer. The weights into each neuron are adjusted in proportion to that neuron’s error.

During the training session of the network, a pair of patterns is presented \((X_k, T_k)\), where \(X_k\) is the input pattern and \(T_k\) is the target or desired pattern. The \(X_k\) pattern causes output responses at teach neurons in each layer and, hence, an output \(O_k\) at the output layer. At the output layer, the difference between the actual and target outputs yields an error signal. This error signal depends on the values of the weights of the neurons in each layer. This error is minimized, and during this process new values for the weights are obtained. The speed and accuracy of the learning process—that is, the process of updating the weights—also depends on a factor, known as the learning rate.

Before starting the back propagation learning process, the following are needed

- The set of training patterns, input, and target
- A value for the learning rate
- A criterion that terminates the algorithm
- A methodology for updating weights
- The nonlinearity function (usually the sigmoid)
- Initial weight values (typically small random values)

The process then starts by applying the first input pattern \( X_k \) and the corresponding target output \( T_k \). The input causes a response to the neurons of the first layer, which in turn cause a response to the neurons of the next layer, and so on, until a response is obtained at the output layer. That response is then compared with the target response, and the difference (the error signal) is calculated. From the error difference at the output neurons, the algorithm computes the rate at which the error changes as the activity level of the neuron changes. So far, the calculations were computed forward (i.e., from the input layer to the output layer). Now, the algorithm steps back one layer before that output layer and recalculate the weights of the output layer (the weights between the last hidden layer and the neurons of the output layer) so that the output error is minimized. The algorithm next calculates the error output at the last hidden layer and computes new values for its weights (the weights between the last and next-to-last hidden layers). The algorithm continues calculating the error and computing new weight values, moving layer by layer backward, toward the input.

When the input is reached and the weights do not change, (i.e., when they have reached a steady state), then the algorithm selects the next pair of input-target patterns and repeats the process. Although responses move in a forward direction, weights are calculated by moving backward, hence the name back propagation.

### 4.6 Implementation of BPN

The hard disk model is implemented by a fast, reliable, and general mathematical model based on back propagation network. For all hard disks, a 3 : \( h(f1); (f0) \) BPN network is proposed as shown in Figure 4.9. The start position, end position, and movement distance of disk head are the inputs of neural network and
the seek time is its output. However, using start and end positions of head are sufficient as network inputs but to increase the learning speed of proposed BPN network, the third input has been also considered.

Figure 4.9 Network topology with 3 inputs, n hidden neurons in one hidden layer and one output neuron

The Matlab neural network toolbox (Demuth and Beale 1993) has been used in implementing the proposed BPN network. Moreover, to find the best and effective number of hidden neurons and the type of transfer functions, several networks were tested with variable number of hidden neurons and transfer functions on a hard disk whose parameters has been shown in Table 4.1.
A data set contained 60% of all possible input vectors is provided which were randomly generated (each example includes a random start position, a random stop position and their movement distance), applied them on the mentioned hard disk and provided their seek times (the time that the hard disk needs to move its head from the start position to the stop position). On the other hand, mappings of each Logical Block Numbers (LBN) to its physical location is obtained, given by the (cylinder, head, sector) tuple and then to measure seek time, a pair of cylinders separated by the desired distance is chosen, issue a pair of read commands to sectors in those cylinders and measure the time between their completions. These data have fed to BPN network for training. To find the best topology for the network, several networks are considered with following structures:

$$\text{net} = \{3 : h(f1) : l(fo) | h \in \{1, 11, 21, \ldots, 91\} \land f1, fo \in \{\text{LogSig}, \text{TanSig}, \text{PureLin}\} \}$$

This equation shows a set of networks with different number of neurons in hidden layer and different transfer functions for hidden and output layers. In this equation, $h$ is the number of neurons and $f1$ is the used transfer function in hidden layer and $fo$ is the used transfer function in output layer where

<table>
<thead>
<tr>
<th></th>
<th>Cylinders per disk</th>
<th>1449</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>Tracks per cylinder</td>
<td>19</td>
</tr>
<tr>
<td>3.</td>
<td>Sectors per track</td>
<td>72</td>
</tr>
<tr>
<td>4.</td>
<td>Sector size</td>
<td>512 Bytes</td>
</tr>
<tr>
<td>5.</td>
<td>Revolution speed</td>
<td>7200 RPM</td>
</tr>
<tr>
<td>6.</td>
<td>Transfer rate</td>
<td>10 MBps</td>
</tr>
</tbody>
</table>
PureLin \( (n) = n \)  

\[
\text{LogSig} \ (n) = \frac{1}{1 + e^{-n}} 
\]  

\[
\text{TanSig} \ (n) = \frac{e^n - e^{-n}}{e^n + e^{-n}} 
\]

(4.1)  

(4.2)  

(4.3)

It is worth mentioning that the networks were trained using the Modified Levenberg-Marquardt (MLM) algorithm and considered 20% of the data for test, 20% for validation and 60% for training. Note that the validation vectors were used to stop training early if the network performance on the validation vectors fails to improve or remains the same for predefined number of epochs (100 in this test). Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training. The training phase was terminated when at least one of the following conditions emerged

- \( Mu > 1e10 \)
- 1000 epochs
- 5 successive iterations the validation data failures
- \( Gradient < 1e-10 \)

The value of \( Mu \) has been elaborated in Hagan et al. (1996). Each network is trained 10 times independently and an error was calculated after training phase. The error was calculated as follows: all possible head seeks were generated and provided their corresponding output (seek times of the hard disk). After the training phase was completed in each run, these data are applied and calculated the error of the network using the following formula

\[
E = \frac{\sum_{i=1}^{n} |net(testcase_i) - 0(testcase_i)|}{n} 
\]

(4.4)

The expression \( E \) shows the error and \( n \) is the number of all possible head seeks. \( Testcase_i \) shows the \( i \)th test example contained three fields: head start position,
head stop position and the distance. net (testcasei ) calculates the output of the network with the input Testcasei . Furthermore, O is the real (expected) output value for Testcasei . Figure 4.10 exhibits the mean errors in hidden layers.

**Figure 4.10 : The mean of errors in hidden layers**

The x-axis of the graph shows number of neurons in the hidden layer of the network while the y-axis shows the mean of error over 60% randomly generated test cases for the hard disk in Table 4.1. Each curve in the figure exhibits the error of a network whose transfer functions have been denoted in the legend of the figure. It is obvious that the network approximated the working function of the hard disk with mean of error about 0.005 (ms) when the transfer function LogSig and PureLin for hidden and output layers were used respectively. Moreover, the best performance has appeared when the number of neurons in the hidden layer is 61 in Figure 4.10.

The average of error for a network with 61 neurons in its hidden layer is very small and the network is satisfactory. Therefore, a 3 : 61(LogSig) : 1(PureLin) network structure to model disks seeks time is used.

Form the above BPN network, the optimized seek time for the given input data request is calculated and compared with the average seek time of the FCFS.
SSTF and SCAN disk seek algorithm. Then the appropriate disk seek algorithm which matches with the optimized seek time is selected for a particular disk. Therefore each disk will be accessed by different disk seek algorithm.

4.7 Analysis and Design of the System

4.7.1 Proactive Disk Power Management

After extracting disk access patterns and the selected disk seek algorithm, the compiler can insert explicit disk power management calls (instructions) in appropriate places in the source code. For TPM disks, use spin up() and spin down() calls. The format of the spin down() call is as follows:

Spin_down(di)

where diski is the disk id. Since a disk access pattern indicates not only idle times but also active times anticipated in the future, this information can be used to preactivate disks that have been spun down by a spin down() call. To determine the appropriate point in the code to start spinning up the disk (that is, preactivation point), the spin-up time (delay) of the disk (i.e., the time it takes for the disk to reach its full speed where it can perform read/write activity) is taken into consideration. Specifically, the number of loop iterations before inserting the spin-up (preactivation) call can be calculated as:

\[ Q_{su} = \frac{T_{su}}{S + T_m} \]

(4.5)

where \( Q_{su} \) is the preactivation distance (in terms of loop iterations), \( T_{su} \) is the expected spin-up time, \( T_m \) is the overhead incurred by a spin up call, and \( S \) is the number of cycles in the shortest path through the loop body. It is to be noted that \( T_{su} \) is typically much larger than \( S \). The format of the call that is used to preactivate (spin up) a disk is as follows:
spin_up(d_i)

Note that, if preactivation is not used, a TPM disk is automatically spun up when an access (request) comes, but, in this case, the associated spin-up delay is incurred fully. The purpose of the disk preactivation is to eliminate this performance penalty.

The discussion so far has focused on the TPM disks as the underlying mechanism to save power, this compiler-driven proactive strategy can also be used with DRPM disks. The necessary compiler analysis and the disk access pattern construction process in this case are the same as in the TPM case.

The main difference is how the disk access pattern collected is used (by taking the times to change disk speed into account) and the calls inserted in the code. In this case, the following call is employed:

\[\text{set RPM(rpm level}_{j}, d_{i})\]

where \(d_{i}\) is the disk id, and \(rpm \text{ level}_{j}\) is the \(j\)th RPM level (i.e., disk speed) available. When executed, this call brings the disk in question to the speed specified. The selection of the appropriate disk speed is made as follows. Since the transition time from one RPM step (level) to another is proportional to the difference between the two RPM steps involved, the detected idle time is considered to determine the target RPM step. Consequently, an RPM level is selected if and only if it is the slowest available RPM level that does not degrade the original performance. It must be mentioned that a wrong placement of the spin up(), spin down(), and set RPM() calls in the code does not create a correctness issue. In the worst case scenario, they increase execution cycles and/or energy consumption. For example, prematurely spinning down a disk (in the TPM-based architecture) delays the time to service the next request, and leads to some extra energy consumption. Similarly, selecting a wrong RPM level to use (in the DRPM-based architecture) can increase disk energy consumption (if the selected level is faster than the optimal
one) or execution time (if the selected level is slower than the optimal one). In either case, however, this is not a correctness issue. Notice however that the compiler places these power management calls into the code based on the disk access pattern it constructs for each disk. Since the compiler is conservative in handling the control flow within the loop bodies (i.e., it assumes that all branches of a conditional construct can be taken at runtime with an equal probability), the information it extracts (regarding disk idle/active times) may not be hundred percent accurate. Also Notice that while this compiler-directed proactive management can be very effective in reducing disk power, restructuring the source code can be done so that disk reuse can be increased significantly.

4.7.2 Comparison of TPM techniques

![Diagram showing comparison between hardware-based TPM and compiler directed TPM](image)

**Figure 4.11 Comparison of the hardware-based TPM and the compiler directed TPM**

Figure 4.11 illustrates the difference between the hardware-based TPM and the compiler-directed TPM. Compared to the hardware based TPM, compiler-directed TPM approach has two advantages. First, the compiler directed TPM can put idle disks in low-power mode earlier than the hardware-based TPM can do. Second, the compiler-directed TPM can avoid the performance overhead, using preactivation, due to the spin up latency when an idle disk is accessed.
4.7.3 Compiler Inserting Disk Power Management calls

Here presents the compiler algorithm for disk energy optimization. This algorithm works in two steps. In the first step, a Loop Transition Graph (LTG) is built for a given procedure. Each node $L_i$ in the LTG corresponds to a loop nest in the procedure. A loop nest whose execution time is longer than a given threshold $Q$ is recursively broken down into smaller loop nests until no loop nest contains any internal loop, or the execution time of the loop is shorter than $Q$. Each edge (from $L_i$ to $L_j$) in LTG has a tag $C_{ij}$, indicating the condition under which the flow of execution transitions from loop nest $L_i$ to $L_j$.

Procedure loop Transformation()
{
    buildLG();
    transform();
}

Procedure buildLTG()
{
    for each outermost loop $L_i$
        addNode($L_i$);
    for each node () in the LTG
        determine disk access pattern $D_i$;
    for each pair of nodes ($L_i$ and $L_j$) in the LTG
        determine transition condition $C_{i,j}$;
}

Procedure addNode($L_i$)
{
    If (execTime($) $Q$ and $L_i$ contains inner loops)
    {
        for each outermost loop $L_j$ in $L_i$
            addNode($L_j$);
else
    { Add node L_i to the LTG; }
    }
    }
Procedure transform()
{
    for each node L_i in the LTG
    {
        if (execTime(L_i) > Q)
        {
            for each disk d_x
                if (D[x][i] = 0)
                    insert before the entry of loop nest L_i:
                        spin_dwn(d_x);
            if (exists L_i \rightarrow L_j such that d[i] \& d[j] != d[j])
            {
                split L_i into two consecutive loop nests : L_i and L_i''
                such that execTime(L_i'') = Q_{su}:
                for each disk d_k such that D[k][x] = 0
                for each loop nest L_j such that L_i \rightarrow L_j
                    if (D[j][x] = 1)
                        insert before the entry of L_i'':
                if (C_{i,j} \& spin_up(d_k))'';
            }
        }
    }

Compiler algorithm for inserting disk power management calls in
a given code fragment
for \( l_1 = 0 \) to \( N_1 \)
{
    for \( l_2 = 0 \) to \( N_2 \)
    {
        \( L_1 \): for \( l_3 = 0 \) to \( N_3 \)
        access : \( d_3 \), \( d_4 \),
        \( L_2 \): for \( l_4 = 0 \) to \( N_4 \)
        access : \( d_5 \), \( d_6 \),
    }
    \( L_3 \): for \( l_5 = 0 \) to \( N_5 \)
    access : \( d_7 \),
}
\( L_4 \): for \( l_6 = 0 \) to \( N_6 \)
access : \( d_8 \),

An example that illustrates proactive disk power management

![Loop Transition Graph (LTG) for the code fragment](image_url)
for \( I_1 = 0 \) to \( N_1 \)
{
for \( I_2 = 0 \) to \( N_2 \)
{
    \text{Spin\_down}(d_2, d_3)
    \text{L}_{1'}: \text{for } I_3 = 0 \text{ to } N_3
    \text{access: } d_0, d_1 \)
    \text{if(true) spin\_up}(d_i);
    \text{L}_{4''}: \text{for } I_3 = N_3 - Q_{su} \text{ to } N_3
    \text{access: } d_0, d_1 \)
    \text{L}_{2'}: \text{spin\_down}(d_1, d_4)
    \text{for } I_4 = 0 \text{ to } N_4 - Q_{su} - 1
    \text{access: } d_0, d_3 \)
    \text{if } (I_2 < N_2) \text{ spin\_up}(d_1)
    \text{if } (I_2 = N_2) \text{ spin\_up}(d_1)
    \text{L}_{2'}': \text{for } I_4 = N_4 - Q_{su} \text{ to } N_4
    \text{access: } d_0, d_3 \)
    \text{spin\_down}(d_1, d_2, d_3, d_4)
    \text{L}_{5'^*}: \text{for } I_5 = 0 \text{ to } N_5 - Q_{su} - 1
    \text{access: } d_4 \)
    \text{if } (I_1 < N_1) \text{ spin\_up}(d_3, d_1)
    \text{if } (I_1 = N_1) \text{ spin\_up}(d_3)
    \text{L}_{5'}: \text{for } I_5 = N_5 - Q_{su} \text{ to } N_5
    \text{access: } d_4 \)
    \text{spin\_down}(d_1, d_2, d_3, d_4, d_5)
    \text{L}_{4}: \text{for } I_6 = 0 \text{ to } N_5
    \text{access: } d_3 \)
}

An example that illustrates proactive disk power management.
Above algorithm shows an LTG for the code fragment in Figure 4.12. In second step, the algorithm inserts code to the program to spin up/down the disks. Specifically, for each node Li in LTG, the algorithm inserts, before the entry of Li, the spin down calls for the disks that are not accessed in Li. Further, if node Li has a successor Lj that accesses a disk that has been spun down in Li, split Li into two consecutive loop nests, L’i and L”i, such that the execution time of L”i is equal to Qsu, the time required to spin up a disk. Before L”i, our algorithm inserts the spin up calls for the disks that will be used in Lj. By doing this transformation, the performance overhead due to disk spin up is hidden. That is, as explained earlier, this preactivation eliminates potential performance penalty. Figure 4.12 shows an example code fragment, and gives the corresponding LTG. The above algorithm is the transformed code fragment after applying the compiler algorithm.

4.8 Experimental Setup

The DiskSim simulator is used in this research work. DiskSim is driven by externally-provided disk I/O request traces, which are generated, by the trace generator. Each I/O request is composed of the following five parameters:

- The id of the processor that issues the request.
- Request arrival time: Time in milliseconds specifying the time at which the disk request arrives.
- Start block number: An integer specifying a logical disk block striped over several I/O nodes.
- Request size: An integer in bytes specifying the size of a request.
- Request type: A character specifying whether the request is a read (R) or a write (W).

The simulator generates statistical data for performance and energy consumption. Both performance and energy statistics were calculated based on the
figures extracted from the data sheet of the IBM Ultrastar 36Z15 [109], and are given in Table 4.2. The values for power mode transitions are also included in Table 4.2.

**Table 4.2 Default simulation parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Clock Frequency</td>
<td>1.5GHz</td>
</tr>
<tr>
<td>Disk Model</td>
<td>IBM Ultrastar 36Z15</td>
</tr>
<tr>
<td>Interface</td>
<td>SCSI</td>
</tr>
<tr>
<td>Storage Capacity</td>
<td>18 GB</td>
</tr>
<tr>
<td>Power (active)</td>
<td>13.5 W</td>
</tr>
<tr>
<td>Power (idle)</td>
<td>10.2 W</td>
</tr>
<tr>
<td>Power (standby)</td>
<td>2.5 W</td>
</tr>
<tr>
<td>Energy (spin down: idle!standby)</td>
<td>13 J</td>
</tr>
<tr>
<td>Time (spin down: idle!standby)</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>Energy (spin up: standby!active)</td>
<td>135 J</td>
</tr>
<tr>
<td>Time (spin up: standby!active)</td>
<td>10.9 sec</td>
</tr>
<tr>
<td>Stripe unit (stripe size)</td>
<td>64 KB</td>
</tr>
<tr>
<td>Stripe factor (number of disks)</td>
<td>8</td>
</tr>
<tr>
<td>Starting iodevice (starting disk)</td>
<td>0</td>
</tr>
</tbody>
</table>

The disk energy consumption includes the energy consumptions in both active and idle periods, taking into account all the states that the disks experience during the entire execution. Also, the performance numbers include all conflicts in accessing the parallel disk system.

**Table 4.3 Benchmarks and their characteristics**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Size (MB)</th>
<th>Number of Disk Reqs</th>
<th>Base Energy (J)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>168 wupwise</td>
<td>176.7</td>
<td>24718</td>
<td>20835.96</td>
<td>248790.00</td>
</tr>
<tr>
<td>171 swim</td>
<td>96.0</td>
<td>3159</td>
<td>2686.79</td>
<td>32088.98</td>
</tr>
<tr>
<td>172 mgrid</td>
<td>384.0</td>
<td>49152</td>
<td>32759.71</td>
<td>388436.95</td>
</tr>
<tr>
<td>173 applu</td>
<td>256.0</td>
<td>32768</td>
<td>22763.58</td>
<td>270278.68</td>
</tr>
</tbody>
</table>

Table 4.3 gives the set of array-based benchmark codes used in this research work. These benchmarks were randomly chosen from the Spec2000 floating-point
benchmark suite in IBM (1999). To complete the simulations within a reasonable amount of time, only on time-consuming loop nests were considered from these applications. The second column in Table 4.3 gives the total disk resident data size manipulated by the selected loop nests, and the third column shows the number of total disk requests made by each application. The last two columns, on the other hand, give the disk energy consumption and execution time, respectively, for each application when no disk power management is employed.

4.9 Results and Discussion

To compare different approaches to disk power management, three different schemes for each benchmark code is simulated in the experimental suite:

TPM: This is the traditional disk power management strategy used in previous studies (Douglos .F, Krishnan .P and Bershad .B).

DRPM: This is the dynamic RPM strategy proposed by Gurumurthi .S, Sivasubramaniam .A, Kandemir .M and Franke .H.

BPN with MLM: This is the compiler directed disk power management using back propagation network trained with modified levenberg marquardt algorithm.

Here considered two types of distributions for the inter-arrival times, namely, exponential and Pareto. Exponential arrivals model a purely random Poisson process, and to large extent model a regular traffic arrival behavior (without burstiness). On the other hand, the Pareto distribution introduces burstiness in arrivals, which can be controlled.

The term workload to define the combination of the distribution that is being used and the mean inter-arrival time for this distribution. For example, the workload <Exp, 100> denotes an exponential traffic with a mean inter-arrival time of 100ms.

The performance evaluation of the proposed BPN with MLM power management approach is compared with the techniques TPM and DRPM. The energy saving comparison for various power loads are evaluated.
Figure 4.13 Comparison of Energy Savings for Exponential and Pareto Methods

Figure 4.14 shows the comparison of the energy savings of the proposed BPN with MLM power management approach with the approaches such as TPM and DRPM for the Exponential and Pareto Methods. The result reveals that the energy savings obtained by the proposed BPN with MLM power management approach is very high when compared with the approaches like TPM and DRPM.

Figure 4.14 Impact of stripe size on energy consumption

Figure 4.15 Impact of Stripe Factor on energy consumption
Figure 4.16 Impact of starting disk on energy consumption

The disk energy consumption values in Figure 4.14, 4.15 and 4.16 are given as normalized values with respect to the system that does not employ any power management strategy.

Figures 4.14, 4.15 and 4.16 show the impact of the stripe size, stripe factor and starting disk on energy consumption. It is observed that the proposed BPN with MLM power management approach reduces energy consumption in all the three cases. Thus, the proposed BPN with MLM power management approach outperforms the other three approaches in terms of reducing energy consumption.

4.10 Summary

This chapter shows the software based approach for disk power management by utilizing back propagation network trained with modified levenberg-marquardt algorithm for selecting appropriate disk-seek algorithm. The disk layout information of the disk resident data and the data access pattern are used to construct the disk access pattern. The compiler uses the disk access pattern and the selected disk seek algorithm to restructure the application code to reduce the power consumption by inserting clear disk power management calls in suitable place of the source code. The experimental results show that BPN with MLM power management approach reduces energy consumption.