# Power and Performance in I/O for Scientific Applications

K. Coloma A. Choudhary A. Ching W.K. Liao
*Center for Ultrascale Computing and Information Security*
*Northwestern University*
{kcoloma, choudhar, aching, wkliao}@ece.northwestern.edu

S. W. Son M. Kandemir
*CSE Department*
*Pennsylvania State University*
{sson, kandemir}@cse.psu.edu

L. Ward
*Scalable Computing Systems Department*
*Sandia National Laboratories*
lee@sandia.gov

## Abstract

*The I/O patterns of large scale scientific applications can often be characterized as small, non-contiguous, and regular. From a performance and power perspective, this is perhaps the worse kind of I/O for a disk. Two approaches to mitigating the mechanical limitations of disks are write-back caches and software-directed power management. Previous distributed caches are plagued by synchronization and scalability issues. The Direct Access Cache: DAChe system is a user-level distributed cached that addresses both these problems. Past work on managing disk power during run time were effective, one should be able to improve on those results by adopting a proactive scheme.*

## 1. Introduction

Both transparent client-side caching and software-directed disk power management have an affect on I/O performance and power. While power consumption is not the primary concern of write-back caching, write-back caching nevertheless aggregates man smaller writes into larger ones improving both performance and power. The primary issue with client-side caching in large-scale systems is scalably enforcing coherency. Coherency is of particular concern in scientific applications, since many processes may be accessing a single file. Because of this, the more relaxed semantics of filesystems like NFS cannot be tolerated because of the counter-intuitive results they might produce. DAChe uses remote memory access and a scalable lock manager to maintain coherency and address applications that must use independent I/O.

Software-directed power managment is, of course, more concerned with power, but extra performance is a welcomed side-effect. Prior research and present practices include spinning up and spinning down hard drives based on load at run time. One traditional way is to consider only two speeds (TPM), while another, Dynamic RPM (DRPM) considers several incremental speeds. A more proactive way of efficiently managing disk power consumption is to analyze code and insert explicit disk speed calls.

In section 2 we describe the potential benefits and challenges of file caching in large scale systems. Then in section 3, we describe its architecture and some initial performance results. Following in section 4, we discuss compiler directed disk power management, before moving on to our conlcuding remarks in section 5.

## 2. Client-side Caching Issues

The core of the cache coherency problem is ensuring globally accessible data is up-to-date when used. Two processes, *p0* and *p1*, may read and cache some shared piece of data. Subsequently, *p0* may write to the data. When the *p1* rereads the data, it will read directly from cache and not the new data written by *p0*. This general issue arises at many levels of the memory hierarchy, and is attacked from different angles depending sprecific features at each level.

In the context of I/O, caching can mask the latency of accessing a local or even remote disk by allowing for quicker

reads and writes. Another optimization made possible by client-side caching is write-behind in which small write data is buffered in the cache while computation continues. The aggregated writes to one page are finally written to disk, can reduce the number of I/O calls.

In a parallel environment, there are often many more clients than there are I/O servers. Client-side caching can not only provide cached data quicker, but it also reduces the load on and contention for I/O server resources. Parallel environments are similar to distributed ones in that there are multiple clients accessing a single file system. In a parallel environment however, many processes often work on a single problem and concurrently access an output or input file. Allowing a single client at a time to access and cache a file is unreasonable. Either a multiple client-side caches must be carefully maintained, or there should be know caching at all. Maintaining a client-side cache in a parallel environment is far more challenging than doing it in a distributed one.

Whenever considering client-side caching, it is necessary to also consider what kind of semantics to follow. The strict nature of POSIX consistency semantics make them ill-suited for parallel and distributed computing environments. Rigid enforcement of POSIX consistency in these environments is usually at the expense of performance by either disallowing efficient caching or centralizing cache management. MPI semantics are, by default, slightly more relaxed. After write completion, data must be visible to all processes in the same communicator. Unless atomic mode is explicitly set, concurrent access to conflicting parts of a file are undefined. The tricky thing is that in atomic mode, an atomic access may be noncontiguous. If not for the last bit about atomic noncontiguous accesses, a POSIX compliant filesystem could provide all that MPI requires semantically.

Cache coherency provides fairly intuitive results. With sequential consistency so hard to provide at the library or file system level, this onus is better left to the application developer who is much more well equiped to handle ordering issues than any library could possibly be.

# 3. DAChe

DAChe uses one-sided communication, and can be architecturally divided into 3 primary subsystems: cache metadata, locking, and cache management. Figure 1 illustrates the basic interactions between these subsystems. Its modular implementation makes it quite easy to port and experiment with given that some basic RMA requirements are met. One over-arching theme always under consideration during the design of DAChe is to keep all aspects of cache management as decentralized as possible. A secondary theme is minimization communication where possible. The main tenet of DAChe is that only a single copy of any file page can reside on at most one process. This single-copy rule ensures cache coherency and removes the task of maintaining state for replicated data. Another way to think of it is that there is never more than one usable copy of any given file page. The use of RMA keeps I/O operations in DAChe passive, but requires that `dache_open` and `dache_close` be collective in order to set things up and break things down safely. After `dache_open` completes, all communication is one-sided unless it is with a mutex server described in more detail in the Mutual Exclusion 3.3 subsection.

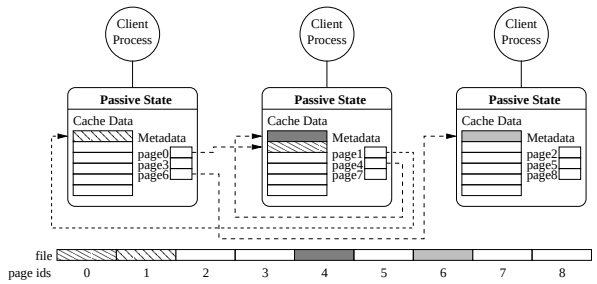


**Figure 1. DAChe architecture with passive metadata and cache servers on each client. Metadata is striped across clients, but a file page can be cached on any client.**

## 3.1. Remote Memory Access

Remote Memory Access (RMA) interfaces provide what can be thought of as shared-memory emulation for a distributed memory environment. RMA in its truest form allows for the movement of data to or from a remote process without its active participation. The primary functions of any RMA interface mirrors shared-memory should consist of a get, put, and some sort of atomic test & set or swap. DAChe uses RMA to remotely access cache metadata and cached data.

DAChe currently uses Portals [1] for RMA. While portability is definitely a con, the Portals implementation is fairly mature and exists on specific large-scale platforms of interest. At the time of development, MPI-2's one-sided communication was not yet ready in MPICH2.

## 3.2. Cache Metadata

Cache metadata maintains basic state for each page in the file. Most importantly, this metadata provides the whereabouts of any given file page. File page refers to the logical partitioning of the entire file into blocks of a size matching the page size of the cache. If a page is cached on any

process the metadata reflects the caching process as well as an index location into that process's cache. cache metadata is remotely accessible through RMA and distributed across the application nodes in a deterministic fashion; in this case a basic striping algorithm. By striping the metadata array, or table as it will be called, across nodes deterministically, not only is a potential bottleneck avoided, but there is also no communication required to find the metadata associated with any file page.

Creating metadata for each logical file page brings up the issue of metadata allocation. The allocation process requires explicit coordination amongst the application processes, and this is only available at the collective `dache_open` and `dache_close` functions. Ideally, one would want the size of the metadata table to be directly related to the size of the file. What this basically entails is some level of cooperation among processes for growing or shrinking the table size during run time. Since the write operations are independent, however, there is no opportunity to coordinate all the processes in order to modify the size of the table. Without this coordination, the last resort would be the ability to remotely allocate globally accessible memory. Needless to say, remote memory allocation brings its own set of challenges.

Given the distributed and passive nature of cache metadata, one crucial element is enforcing mutually exclusive access to it.

### 3.3. Mutual Exclusion

The purpose of the mutual exclusion subsystem is to ensure safe access to read and modify cache metadata. Ideally, mutual exclusion is directly supported in the RMA interface. With proper RMA support for mutual exclusion, the locking subsystem can be also distributed across the application nodes as passive remote accessible state.

MPI-2 explicitly provides the `MPI_Win_Lock` and corresponding unlock functions. Since the current test platform for DAChe is a threadless environment, however, this rules out the use of the MPICH2 RMA interface. MPICH2 is focused on portability, and threads are more commonly available than hardware supported one-sided communication. At the same time, the atomic swap operation in the Portals library has yet to be implemented.

In the absence of an RMA solution to mutual exclusion, several processes from the allocated user processes are siphoned off from the main group to act as mutex servers. They are spun off during `dache_open` and returned during `dache_close`. During this time, the mutex servers cannot execute any user application code. So while passive RMA mutual exclusion is preferred, DAChe can still be evaluated using dedicated mutex servers. Later, these mutex servers will become passive elements on the application processes.

Since mutex responsibilities will eventually be moved to the client, the mutex servers are intentionally kept quite simple. Lock responsibility for each file page is spread across the mutex servers in the same way cache metadata is spread across application processes. The mutex servers service locks in the order they come, queueing requests to the same cache metadata. Polling and simple queueing are both implementable when the mutex servers become passive state on remote processes. A process must block until its lock request is fulfilled by the mutex server.

Typically, cache metadata is not "held" for extended periods of time. It is locked only briefly for modification. It is not held for the duration of access to the actual cached data. Minimizing the time that the metadata is locked, should reduce the amount of simultaneous lock requests to the same metadata. The exception to short lock times is when a particular file page is being brought into the cache or a cache page is being evicted. In either case, the cache metadata must be locked for entire I/O phase in order to prevent early or late cache accesses, respectively.

### 3.4. Cache Management

Pages cached on one process are globally accessible to any other process through RMA. Although access to metadata is carefully mediated, remote access to cache data is basically a free-for-all. Since any file page can be cached in at most one cache, all accesses to that page are coherent.

Cache management and eviction is handled locally with one exception to be discussed a little further along. What data to cache is determined by the local procces's I/O accesses. If a process accesses a file page that is not yet cached on any of the other processes, it caches it itself and updates the corresponding cache metadata to reflect this change. Should a process run out of cache space locally, it must evict a page based on some local policy such as a least recently used (LRU) policy. Remember that during eviction, a page's metadata cannot be accessed at all. Another precaution alluded to earlier is that processes accessing a remotely cached page must "pin" the page in cache so the page cannot be evicted while being accessed. This pin is a semaphore contained in the cache metadata along with location information. While a page is pinned, the process on which the cache page resides cannot evict the page, and must either wait until the page is "un-pinned" or try to evict a different page. New data is not written to disk until it is either evicted or written out at `dache_close`.

### 3.5. Initial Performance Evaluation

A preliminary analysis of DAChe is done using a synthetic I/O access pattern that should, by design, benefit from

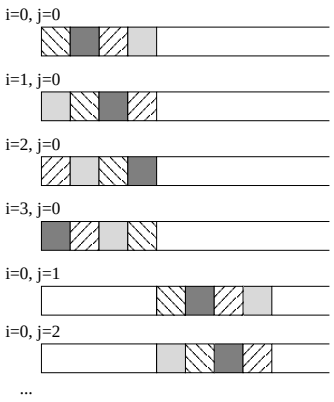


**Figure 2. Several iterations of the sliding window I/O pattern for 4 processes.**

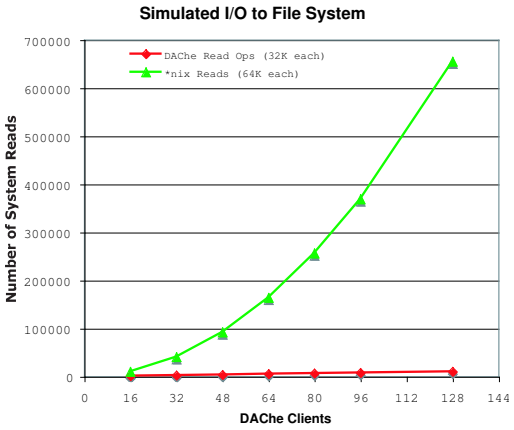


**Figure 3. The number of I/O calls that would have been made to the filesystem is significantly reduced by the presence of DAChe.**

cached data. It is roughly based on the regular noncontiguous access patterns often found in scientific applications and is meant to test the performance of DAChe. The benefits a real application may derive from DAChe are also clearly of interest. The basic labeling convention is as follows:

- dache-n where n is the number of mutex servers
- Nio is the number of clients actually caching data (non-mutex servers)
- 50:50 refers to an equal mix of clients and mutex servers

The sliding window application uses a repetitive I/O access pattern, and the underlying caching library uses the lock service to gain exclusive access to cache page metadata. 2 describes the access pattern of the synthetic benchmark. Each process accesses a contiguous chunk of data. In each subsequent iteration, processes circular shift their accesses until all chunks are read before sliding to the next set of four chunks. Since there is one lock per page, and the benchmark accesses are page-aligned, the number of meta data locks is tied directly to the number I/O operations. The cyclic access pattern makes the total amount of I/O done increase along a power function rather than linearly.

It is important to note that since the DAChe system is the primary subject of evaluation, actual I/O is removed from DAChe to prevent interference from any specific filesystem. The sliding window access pattern is such that once a file page is brought into cache, it remains there throughout the execution and is only remotely accessed. Though actual I/O would have been performed to bring in the file data, all subsequent accesses are really accessing cached data, possibly on remote nodes. Figure 3 illustrates the infeasibility of running this benchmark with actual I/O and without any caching. The resulting amount of aggregate I/O would be from 1 GB to 80 GB. There is little doubt this reduction in I/O calls would also reduce the power consumption of the disks. Bandwidth is calculated based on the amount of I/O the sliding window application thinks it has performed.

Since scalability is the primary goal of the DAChe, it was tested with various numbers of mutex servers running. Intuitively, the heavier the lock system is taxed, the more mutex servers are needed to accomodate the increased load. This is illustrated clearly in figure 4 where a larger number of mutex servers allows a larger number of clients without severely hindering performance when fewer than the maximum clients is used. From a cost efficiency perspective, one would like to stay on the outside curve using the minimum number of processes at each point. This client to mutex server ratio is highly dependent on the properties of the specific machine. A 50:50 mix where there are an equal number of clients and mutex servers allows the mutex service to scale with the application size. As expected from the growth rate of I/O amount, this outer bandwidth curve is roughly an inverse power function. This 50:50 mix is in anticipation of each client doubling as a passive lock server. The most important point from Figure 4 is the number of mutex servers determines at which number of clients performance will plateau.

# 4. Compiler Directed Disk Power Management

Our overall approach is depicted in Figure 5. A unique characteristic of our approach is that it *exposes* the disk pa-

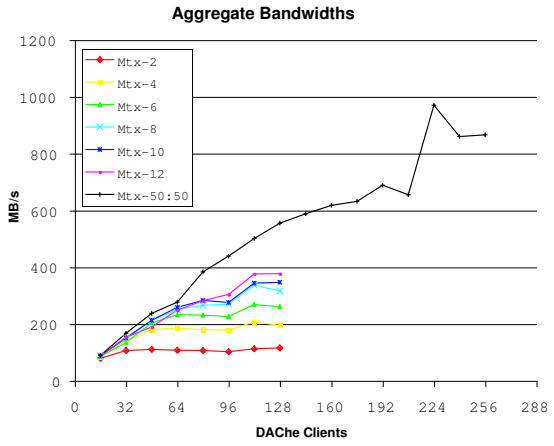


**Figure 4. The number of clients at which bandwidth knees over is dependent on how many lock servers there are.**

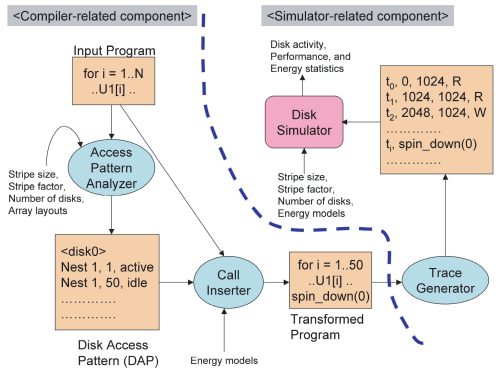


**Figure 5. High-level view of our compiler driven approach to power reduction.**

rameters and the disk layout of array data to the compiler. The goal of doing so is to allow the compiler to determine (estimate) the disk active and idle times. The parameters used by the compiler in this approach include the number of ids of the disks over which each array is striped (i.e., the stripe factor), the stripe size used, and the id of the disk that contains the first stripe of data. Using this information and the data access pattern extracted by analyzing the application source code, the compiler determimnes what we refer to as the *disk access pattern* (DAP). Basically, a DAP indicates how the disks in the I/O subsystem are accessed and

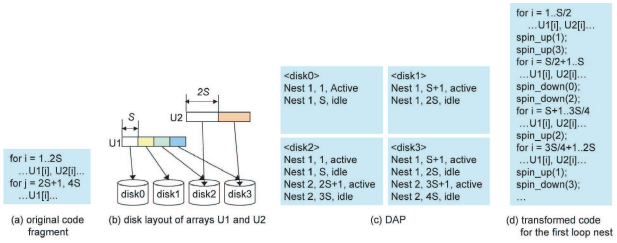


**Figure 6. An example of application of our approach.**

reveals information that is vital for disk power management: *disk inter-access times*, i.e., the gap between the two successive accesses to a given disk. This information can be used in two different ways to do proactive power management: (1) placing a disk into a suitable low-power mode after its current access is complete, and (2) pre-activating a disk to eliminate the potential performance penalty due to power management. Then, based on this information, the application code is modified to insert explicit power management calls. The nature of these calls depends on the underlying method used (e.g., TPM versus DRPM), and will be explained shortly. This compiler transformed code can execute on a system that uses TPM or DRPM disks to do proactive power management. In the rest of this section, we explain the three important components of our compiler-driven approach: the compiler-analysis to identify disk accesses, the DAP, and the insertion of explicit power management calls in the code.

In order to determine the disk access pattern, we need two types of information: data access pattern and disk layout of array data. The data access pattern indicates the order in which the different array elements are accessed, and is extracted by the compiler by analyzing the source code of the application. To determine which particular disks are being accessed, the compiler also needs the layout of array data (i.e., the file that holds the array elements) on the disk subsystem. In this context, the disk layout of an array (which is stored in a file) is specified using a 3-tuple:

$$(\text{starting\_disk}, \text{stripe\_factor}, \text{stripe\_size}).$$

The first element in this 3-tuple indicates the disk from which the array is started to get striped. The second element gives the number of disks used to stripe the data, and the third element gives the stripe (unit) size. As an example, in Figure 6(b), array `U1` is striped over all four disks in the figure. 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, $S$). Now, let us consider the other array (`U2`) in Figure 6(b), assuming

its size and stripe size are $4S$ and $2S$, respectively. Consequently, its disk layout can be expressed as (2, 2, $2S$). To illustrate the process of identifying the disk accesses, let us consider the code fragment in Figure 6(a). During the execution of the first loop nest, this code fragment accesses the array elements `U1[1]`, `U1[2]`, . . . , `U1[2S]` and `U2[1]`, `U2[2]`, . . . , `U2[2S]`. Consequently, for array `U1`, we access the first two disks (disk0 and disk1); and for array `U2`, we 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 [6], we can 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. When creating a file from within the application, this layout information can be made available to the compiler as well, and, as explained above, the compiler uses this information in conjunction with the data access pattern it extracts to determine the disk access pattern. On the other hand, if the file is already created on the disk subsystem, the layout information can be passed to the compiler as a command line parameter.

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

$$\begin{array}{lll} < \text{Nest 1}, & \text{iteration 1}, & \text{Idle} > \\ < \text{Nest 2}, & \text{iteration 50}, & \text{Active} > \\ < \text{Nest 2}, & \text{iteration 100}, & \text{Idle} > \end{array}$$

We see from this example DAP that, 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 6(a) and the disk layouts illustrated in Figure 6(b), Figure 6(c) gives the DAPs for each of the four disks in the system. The last component of our compiler-driven strategy is responsible from inserting explicit disk power management calls in the code. Let us first focus on the TPM based disks. It is important to note that a DAP is given in terms of loop iterations. In order to determine the appropriate places in the code to insert explicit power management calls, we need to interpret the loop iterations in terms of cycles, which can be achieved as follows. The cycle estimates for the loop iterations are obtained from the actual measurement of the program by using a high-quality timer called *gethrtime*, which is available on the UltraSPARC-based systems. Since the measured execution time is given in nanoseconds scale and we are given the machine's clock rate, we can estimate the number of cycles per each loop iteration.

Once we determine that the estimated disk idleness (in terms of cycles), if this idleness is larger than the *breakeven threshold*, i.e., the minimum amount of idle time required to compensate the cost of spinning down in a TPM disk, the compiler inserts a spin down call in the code. The format of this call is as follows:

$$\texttt{spin\_down}(\texttt{disk}_i),$$

where $\texttt{disk}_i$ is the disk id. Since a DAP indicates not only idle times but also active times anticipated in the future, we can use this information to preactivate disks that have been spun down by a `spin_down` call. To determine the appropriate point in the code to spin up the disk, we take into account the spin-up time (delay) of the disk. Specifically, the number of loop iterations before which we need to insert the spin-up (pre-activation) call can be calculated as:

$$d = \lceil \frac{T_{su}}{s + T_m} \rceil \quad (1)$$

where $d$ is the pre-activation distance (in terms of loop iterations), $T_{su}$ is the expected spin-up time, $T_m$ is the overhead of a `spin_up` call, and $s$ is the number of cycles in the shortest path through the loop body. Note that, $T_{su}$ is typically much larger than $s$. We also stripe-mine the loop, because it is unreasonable to unroll the loop to make explicit the point at which the spin-up call is to be inserted. The format of the call that is used to pre-activate (spin up) a disk is as follows:

$$\texttt{spin\_up}(\texttt{disk}_i),$$

where as before $\texttt{disk}_i$ is the disk id. For our running example, Figure 6(d) shows the compiler-modified code with the `spin_down` and `spin_up` calls. In this transformed code, the calculated pre-activation distance, $d$, is assumed to $S/2$. Let us assume that, at the beginning of the first loop, disk0 and disk2 are active, and disk1 and disk3 are in the low-power mode. Since the analyzed DAP indicates that disk1 and disk3 will be accessed after $S$ iterations, we insert a `spin_up` call to pre-activate disk1 and disk3 between the first and the second loop nests in Figure 6(d). After executing $S$ iterations, we insert a `spin_down` call for disk0 and disk2 because they are not accessed for the remaining execution of the code. Note that, if we do not use pre-activation, the disk is automatically spun up when an access (request) comes; but, in this case, we incur the associated spin-up delay fully. The purpose of the disk pre-activation is to eliminate this performance penalty.

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

TPM case. The main difference is how the information recorded in the DAP is used (by taking into account the times to change RPM) and in the calls inserted in the code. In this case, we employ the following call:

$$\texttt{set\_RPM}(\texttt{rpm\_level}_j, \texttt{disk}_i),$$

where $\texttt{disk}_i$ is the disk id, and $\texttt{rpm\_level}_j$ 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 [4], we need to consider the detected idle time in order to determine the target RPM step. Consequently, we select an RPM level if and only if it is the slowest RPM level whose transition latency can be captured by the estimated idleness. To evaluate our proposed compiler-directed proactive approach to disk power management, we wrote a trace generator and a disk power simulator (see Figure 5).

### 4.1. Brief Results

To compare different approaches to disk power management, we implemented and performed experimentswith different schemes:

- Base: This is the base version that does not employ any power management strategy. All the reported disk energy and performance numbers are given as values normalized with respect to this version.
- TPM: This is the traditional disk power management strategy used in studies such as [3] and [2], using two modes, normal and low-power.
- Ideal TPM (ITPM): This is the ideal version of the TPM strategy. In this scheme, we assume the existence of an oracle predictor for detecting idle periods. Consequently, the spin-up and spin-down activities are performed in an optimal manner.
- DRPM: This is the dynamic RPM strategy proposed in [4], which steps the RPM at several different increments.
- Ideal DRPM (IDRPM): This is the ideal version of the previous strategy. In this scheme, we assume the existence of an oracle predictor for detecting idle periods, as in the ITPM case. Consequently, the disk speed to be used is determined optimally.
- Compiler-Managed TPM (CMTPM): This corresponds to our compiler-driven approach when it is used with TPM. The compiler estimates idle periods by analyzing code and considering disk layouts, and then makes spin-down/up decisions based on this information.
- Compiler-Managed DRPM (CMDRPM): This corresponds to our compiler-driven approach when it is used with DRPM. The compiler estimates idle periods by analyzing code and considering disk layouts, and then selects the best disk speed to be used based on this information.

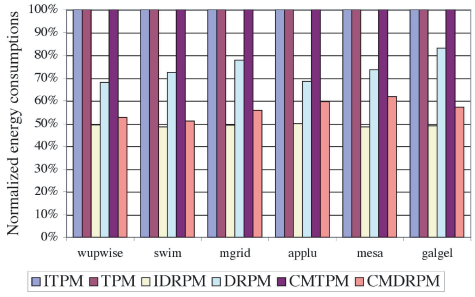


**Figure 7. An example of application of our approach.**

It must be emphasized that, the ITPM and IDRPM schemes are not implementable. The reason that we make experiments with them is that we want to see how close our compiler-based schemes come close to the optimal. All necessary code modifications are automated using the SUIF infrastructure [5].

The graph in Figure 7 gives the energy consumption of our benchmarks under the different schemes described earlier. One can make several observations from these results. First, the TPM version (ideal or otherwise) does not achieve any energy savings. Second, while the DRPM version generates savings (26% on the average), the difference between it and the ideal DRPM (IDRPM) is very large; the latter reduces the energy consumption by 51% when averaged over all benchmarks in our suite. This shows that a reactive strategy is unable to extract the potential benefits from the DRPM scheme. Our next observation is that the CM-DRPM scheme brings significant benefits over the DRPM scheme, and improves the energy consumption of the base scheme by 46%. In other words, it achieves energy savings that are very close to those obtained by the IDRPM strategy. These results demonstrate the benefits of the compiler-directed proactive strategy.

It is to be noted, however, that the energy consumption is just one part of the big picture. In order to have a fair comparison between the different schemes that target disk power reduction, we need to consider their performances (i.e., execution times/cycles) as well. The bar-chart in Figure 8 gives the normalized execution times (with re-

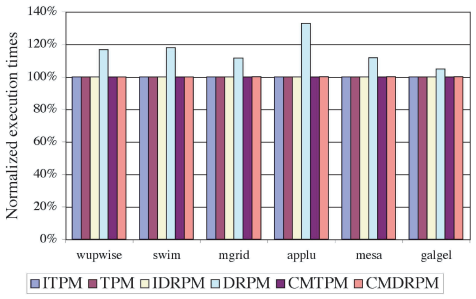


**Figure 8. An example of application of our approach.**

spect to the base version) for the different schemes evaluated. The reason that the TPM-based schemes do not incur any performance penalty is because they are not applicable, given the short disk idle times discussed earlier. When we look at the DRPM-based schemes, we see that the conventional DRPM incurs a performance penalty of 15.9%, when averaged over six benchmarks. We also see that the CM-DRPM scheme incurs almost no performance penalty. The main reason for this is that this scheme starts to bring the disk to the desired RPM level before it is actually needed, and the disk becomes ready when the access takes place. This is achieved by accurate prediction of disk idle periods. These results along with those presented in Figure 7 indicate that the compiler-directed disk power management can be very useful in practice, in terms of both energy consumption and execution time penalty. Specifically, as compared to the reactive DRPM implementation, this scheme reduces the disk power consumption and eliminates the performance penalty.

# 5. Conclusions and Future Directions

While preliminary results for DACche are promising, there is still much to be investigated. In the current implementation, the LRU queue is stored locally, and is only affected by local accesses to cache pages. Not only should the basic eviction performance of DACche be evaluated, but further research is planned into how taking remote accesses into account in the eviction policy may affect overall cache performance. An extension of that would be migrating cache pages frequently accessed by a particular process to that node.

Tighter integration with MPI is also planned. By moving away from the Portals interface, DACche will benefit from the portability of MPICH2.

Early performance results for DACche suggest it scales reasonably well. The extremely distributed characteristics of DACche get around a number of potential bottlenecks. While there were no explicit power studies done on the effects of client-side caching, presumably DACche can offer a reduction in the number of disk accesses an application may make. The two most interesting features of DACche are its efficient use of the one-sided communication architecture and its scalable lock manager.

For array-intensive scientific applications, the compiler can extract a disk access pattern and use it for placing disks into the most suitable low-power modes. In principle, this approach can be used with both TPM and DRPM disks. The compiler-directed proactive approach to disk power-managements is successful in imporoving the behavior of the DRPM based scheme. On average, it brings an additional 18% energy savings over the hardware-based DRPM.

# Acknowledgements

This work was supported in Part by NSF Grants #0444158, #0406340, #0093082, CNS-0406341, CCF-0444405, Sandia National Laboratory contract 28264, and DOE award DE-FC02-01ER25485.

# References

[1] R. Brightwell, B. Lawry, A. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication.

[2] F. Douglis, P. Krishnan, and B. Bershad. Thwarting the power-hungry disk. In *Proceedings of the USENIX Winter Conference*, pages 292–306, 1994.

[3] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[4] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–179, June 2003.

[5] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[6] R. Ross, P. Carns, W. L. III, and R. Latham. Using the parallel virtual file system. July 2002.