Dynamic Compilation for Reducing Energy Consumption of I/O-Intensive Applications*

Seung Woo Son[1], Guangyu Chen[1], Mahmut Kandemir[1], and Alok Choudhary[2]

[1] Pennsylvania State University, University Park PA 16802, USA,
{sson,gchen,kandemir}@cse.psu.edu
[2] Northwestern University, Evanston IL 60208, USA,
choudhar@ece.northwestern.edu

Abstract. Tera-scale high-performance computing has enabled scientists to tackle very large and computationally challenging scientific problems, making the advancement of scientific discovery at a faster pace. However, as computing scales to levels never seen before, it also becomes extremely data intensive, I/O intensive, and energy consuming. Amongst these, I/O is becoming a major bottleneck, impeding the expected pace of scientific discovery and analysis of data. Furthermore, the applications are becoming increasingly dynamic in terms of their computation patterns as well as data access patterns to cope with larger problems and data sizes. Due to the complexities of systems and applications and their high energy consumptions, it is, therefore, very important to address research issues and develop dynamic techniques at the level of run-time systems and compilers to scale I/O in the right proportions. This paper presents the details of a dynamic compilation framework developed specifically for I/O-intensive large-scale applications. Our dynamic compilation framework includes a set of powerful I/O optimizations designed to minimize execution cycles and energy consumption, and generates results that are competitive with hand-optimized codes in terms of energy consumption.

1 Introduction and Motivation

Tera-scale high-performance computing has enabled scientists to tackle very large and computationally challenging problems, such as those found in the scientific computing domain. This in turn helps advancement of scientific discovery at a faster pace. However, as computing scales to levels never seen before, it also becomes extremely data intensive, I/O intensive, and energy consuming. Thus, I/O is becoming a major bottleneck, slowing the expected pace of scientific discovery and analysis of data. This high I/O intensiveness also means that a significant portion of the energy consumption during the execution of high-performance applications occurs in the I/O systems. Furthermore, to cope with larger problems and data sizes, models and applications are being designed to be dynamic in nature. That is, the applications are becoming increasingly dynamic [8, 9] in terms of their computation patterns and data access patterns (e.g., changing smaller

* This work is supported by NSF grants #0444158, #0406340, #0093082 and a grant from GSRC.

structured mesh based designs to dynamic adaptive mesh refinement techniques for algorithm scalability, or dynamically analyzing the data to determine interesting features or events to steer computation, etc.). Due to the complexities of systems and applications, it is, therefore, very important to address research issues and develop techniques at the level of run-time systems and compilers to scale I/O in the right proportions. If such techniques are not developed, users will be overwhelmed with I/O bottlenecks since the complexities of large-scale systems do not lend to manual optimizations.

Consider a typical scientific exploration process that involves large-scale simulations. It usually has several phases including simulation runs, post-processing, and analysis. The simulation phase consists of intensive computations that generate large quantities of data. The data need to be saved quickly as they are generated so that the computation is not slowed down because of I/O bottlenecks. In some cases, the simulation can benefit from dynamic steering (which means dynamically changing I/O access patterns), by quickly analyzing intermediate results. A subsequent phase usually requires the post-processing of the simulation data. This may include transformation of the data from one format (storage layout) to another, summarization of the data, reorganization of the data at run-time to facilitate future use efficiently and most effectively. In this phase, a large volume of data has to be read efficiently, and a large volume of data may be generated as well. In the next phase, the analysis phase, relevant subsets of the data need to be selected and analyzed based on the properties of the data (that is, the processing and access patterns are data dependent and dynamic). The analysis phase may require methods that discover specific patterns and relationships in the data as well as capturing inter-relationships between the different datasets. Clearly, the complexities of various phases and steps are tremendous, and all these phases involve energy-consuming operations. Data read/write, processing, organization and flow are major components and represent a major bottleneck today and for the future. An important impact of this I/O intensiveness of large-scale applications is the increased energy consumption on the I/O system. Frequent accesses to parallel disks, for example, can be responsible from a significant fraction of overall power budget, as noted by several prior studies such as [1, 2].

As a result of high-level dynamic changes in the application behavior and/or data layout, two important entities also change: *data access pattern* (i.e., how the datasets are accessed – direction of access, volume of access, frequency of access, etc) and *I/O performance* (e.g., the time spent in I/O activities and energy consumption on the disk subsystem). A data-intensive application can benefit a lot if these changes in its I/O access pattern and I/O performance can be captured and feedback to a *dynamic compiler* that can re-compile the application to take the best advantage of the changing behavior and to improve the time/energy spent in I/O.

This paper explores dynamic compilation for I/O-intensive applications. Specifically, we present an infrastructure that contains a *dynamic optimizing compiler/linker*, a *high-level I/O library* (called HLL), a *mini database system* (a *metadata manager*), and a *layout manager* that together manage a parallel, hierarchical storage system The framework provides I/O-optimized access to datasets regardless of the type of the media they currently reside on, what their storage layouts are, or where the media is located. Where/how the datasets are stored and in what type of media they are stored are hid-

den from the user. This allows the user applications to access a dataset the same way regardless of its current location and storage layout. The compiler, the HLL, the mini database, and the layout manager cooperate to maintain this uniform storage system view. While the dynamic compilation framework discussed in this paper can be used for optimizing both performance and power/energy, in this paper we focus exclusively on energy reduction on the I/O system.

The rest of this paper is organized as follows. Section 2 discusses the major components of our system at a high-level. Since the main focus of this paper is the dynamic compiler, Section 3 focuses on the compiler alone and discusses the suite of I/O optimizations it employs. Finally, Section 4 concludes the paper with a summary of our major contributions.

2 Dynamic Compilation Infrastructure

Figure 1 illustrates the major components of our dynamic compilation framework for I/O-intensive parallel applications. The storage system is assumed to be a parallel, hierarchical storage architecture that has typically a disk-based layer such as NAS (Network Attached Storage) [10] or SAN (Storage Area Network). We also assume that there is a tertiary storage (tape system) that serves as the next level in the storage hierarchy. In this storage architecture, the most critical issue is to schedule and coordinate accesses to data, and manage the data-flow between the different components. We assume that this storage system is used by parallel applications.

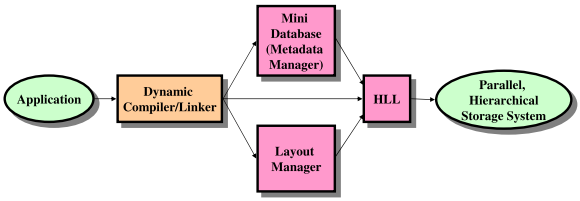

Fig. 1. High-level view of the dynamic compilation approach.

The main goal of the dynamic compilation support discussed in this paper is to identify and implement various I/O optimizations dynamically using the features provided in the HLL. The HLL's capabilities include an interface that facilities the propagation of I/O access patterns and hints for run-time optimizations. Furthermore, to take advantage of the past access patterns from the application, the HLL makes use of a mini database (called the metadata manager) that maintains information about the I/O access patterns as well as relationships among datasets. This is akin to the locality concept in memories. For example, spatial locality says that data items that are close in data space tend to be accessed together and this locality is determined using the addresses of data items. Our approach identifies and takes advantage of so-called dataset locality, which indicates which datasets tend to be accessed together. The metadata stored in the mini database contains such information, and is periodically updated during the course of execution. The goal of the mini database is to learn and store access patterns at various

levels and maintain I/O performance statistics. It does not perform I/O in our implementation. Since the proposed analyses for dynamic compilation are oriented towards exploiting the I/O optimizations supported by the HLL, we first explain the HLL and briefly discuss its functionality and user interface.

The HLL allows an application to access data located in the storage hierarchy via a simple interface expressed in terms of datasets (and arbitrary rectilinear regions of datasets). The main difference between the HLL and the previous array-oriented runtime I/O libraries (e.g., Passion [5, 6] and Panda [12]) is that the HLL maintains the same abstraction (dataset name) across an entire storage hierarchy, and that it accommodates storage hierarchy-specific dynamic I/O optimizations.

The routines in the HLL can be divided into four major groups based on their functionality: Initialization/Finalization Routines, Data Access Routines, Data Movement Routines, and Hint-Related Routines/Queries. Each routine takes a processor id as one of its input parameters, and is invoked by each participating processor. This enables the HLL to see the global picture (which includes the I/O access pattern of each processor) in its entirety. Initialization/finalization routines are used to initialize the library buffers and metadata structures (in the mini database), and finalize them when all the work is done. Data access routines manage the data flow between storage devices and memory. An arbitrary rectilinear portion of a dataset can be read or written using these routines. Using a read routine, for example, the HLL can bring a rectangular portion of a dataset from tape (or disk) to memory. Data movement routines are used to transfer data between storage devices other than memory. These provide a powerful abstraction by expressing the data movement between any storage device pair as a simple copy operation; moreover, these routines work on arbitrary rectilinear portions of datasets. All these routines also have their asynchronous counterparts that return the control to the application code immediately (but perform the specified operation at the background). Hint-related routines are used to pass specific hints on a given dataset to the HLL (hints and queries are not discussed in this paper). Queries, on the other hand, are used by the HLL to extract specific information from the mini database about the datasets such as their current locations in the storage hierarchy, the sizes of their subfiles, etc.

The HLL contains a large set of I/O optimizations (implemented as library routines) that can be incorporated into the application in an on-demand fashion using dynamic linking. However, if a desired I/O optimization (for the best I/O performance and energy savings) is not available in the HLL, the proposed dynamic compiler (that will be described shortly) generates the optimized version by making use of the already available routines (in the HLL).

3 Details of the Dynamic Compilation Framework

Our dynamic compiler has four major components as depicted in Figure 2: (1) dynamic compiler; (2) dynamic linker; (3) performance tracer; and (4) steering unit. The performance tracer is responsible from collecting both I/O access pattern information and performance/energy statistics. The I/O access pattern information includes access directions for data arrays (e.g., row-wise vs. column-wise accesses), whether the dataset is accessed in the read-write mode or mostly in the read-only mode, which datasets are

Table 1. An illustration of performance optimization rules incorporated for data access strategies for efficient I/O. The "Invoked if" column lists the conditions under which the corresponding optimization is invoked by the dynamic compiler.

Optimization	Brief Explanation	Invoked if
Collective I/O (CIO)	Distributing the I/O requests of different processors processor among them so that each accesses as many consecutive data as possible it involves some extra communication between processors.	Access pattern of the data is different from its storage pattern, and multiple processors are use to access the data.
Subfiling (SUB)	Dividing large array into subarrays to reduce transfer latency between different levels of the storage hierarchy	A small subregion of a file is accessed. with high temporal locality.

accessed with temporal affinity, how frequently the datasets are accessed, and similar information that indicates how different datasets are manipulated by the application. The performance statistics include the number of accesses to different storage units (e.g., tapes, disks), misses in disk/file caches, and the time spent in I/O and the energy consumption in different storage elements.

After collecting this information from the metadata manager, the performance tracer passes it to the steering unit (note that the performance analyzer collects only application-specific data from the metadata manager, which keeps metadata for different entities and applications). The main responsibility of the steering unit is to decide whether any dynamic linking and/or compilation needs to be performed, and if so, select the most appropriate libraries and/or optimizations to be invoked . While different triggering criteria can be used for determining whether dynamic compilation/linking is necessary at a particular point during execution, in this work we use a data structure centered approach as explained in rest of this section. As shown in Figure 2, our dynamic compiler and linker are invoked by the steering unit.

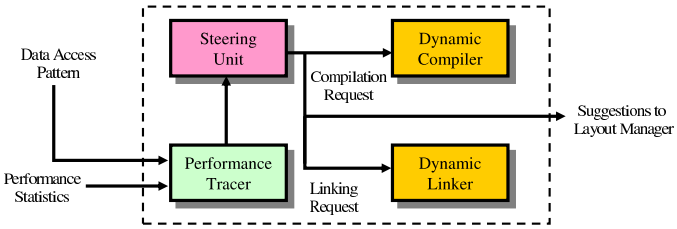

Fig. 2. Components of the dynamic compilation framework.

Table 1 lists the I/O optimizations currently supported by our dynamic compilation framework. The second column briefly describes each optimization, and the third column gives the condition(s) under which each optimization is to be invoked dynamically at run-time.

In collective I/O, small disk requests are merged into fewer larger requests to minimize the number of times the disks are accessed. While it can be used for both read and write operations, we describe it here only for the read operations. In two-phase I/O [6], a client-side collective I/O implementation, the processors first communicate with each other so that each processor knows the total data that need to be read from the disk system. In the second step, they decide what data each processor needs to read so that the number of disk accesses is minimized. In the next step, the processors perform disk

accesses (in parallel). In the last step, they engage in interprocessor communication so that each data item is transferred to its original requester. It needs to be noted that collective I/O, where applicable, can be beneficial from the energy consumption viewpoint since it can reduce the number of disk accesses. While it is true that it also causes some extra interprocessor data communication, the energy incurred by these communications is normally very small compared to the energy gains achieved on the disk system.

Our dynamic compilation analysis for collective I/O has four components: (1) Determining I/O access pattern to the data; (2) Determining storage pattern (layout) of the data; (3) Comparing access and storage patterns to decide whether to apply collective I/O or not; and (4) Modifying the code dynamically if necessary. The access pattern information is obtained from the performance tracer, which keeps track of the dynamic I/O access patterns. The storage pattern indicates how the data is stored in the storage system, and is maintained by our metadata manager. If these two patterns do not match collective I/O is expected to be useful and can reduce energy consumption, and the steering unit either links the appropriate library routine (in the HLL) that implements collective I/O (if such a library routine is available), or dynamically recompiles the application code (that is, the application code is compiled to implement collective I/O using the existing I/O support provided by the HLL). This dynamic compilation is confined to the relevant part(s) of the code, that is, typically the loop nest (or a set of related loop nests) that accesses the data in question. Therefore, the energy spent during dynamic compilation is not expected to be excessive.

It is also possible that the steering unit may decide a "storage layout (pattern) change" for the dataset in question. This may be required in cases where the desired modification to the application code may not preserve the original semantics of the application (hence, it is not legal). In such cases, the steering unit advises the layout manager (see Figures 1 and Figure 2) to change the storage layout of the data. It should be noted that the layout manager can receive such requests from multiple applications running concurrently on the same storage system, and since a given dataset can be accessed by multiple applications, its layout should be modified only if it is going to be beneficial globally (i.e., from multiple applications' perspective). In other words, the steering unit of our framework just makes a suggestion (considering only one application), and the layout manager is free to obey it or not. In this paper, however, we do not evaluate the behavior of layout optimizer.

It should be emphasized that applying I/O optimizations such as collective I/O in a dynamic compilation/linking based setting brings some unique benefits. For example, in many cases, the data access patterns cannot be extracted statically. Consequently, a static compiler either cannot apply collective I/O (as it does not know the access pattern) or can apply it conservatively, which means reduced energy savings. Also, in some cases, the same data can be shared by multiple applications. It is possible that, between two successive accesses by the same application to the same dataset, the layout of data could be modified. In such a case, we need to change the I/O access strategy of the application on-the-fly to take advantage of the new storage layout. Dynamic compilation and linking allow us adapt the I/O access behavior to the current status (layout, location) of the data.

The second optimization for which we discuss the necessary dynamic compilation support in this paper is subfiling [11]. In many I/O-intensive applications such as terrain imaging, document imaging, and visualization, although the datasets manipulated are very large, at a given time, only small portions (regions of interest) of the datasets are used. Unfortunately, most current solutions to large-scale data movement across the storage hierarchies proposed by hierarchical storage management systems [7, 3, 4] retrieve the entire file that contains the dataset in question. This increases latency enormously, and also wastes significant bandwidth. In addition, this also increases the energy consumption significantly. For example, to satisfy a program request of 50 KB of data, they retrieve, say, an entire 8 GB file from tape to disk. In fact, this limitation forces the application programmers/users to break their datasets into small, individually addressable objects, thereby cluttering the storage space and making file management very difficult. In addition, this process is very time consuming and error-prone. Instead, subfiling moves a minimum amount of data between storage devices when satisfying a given program's I/O requirements. This is achieved by breaking up the large datasets into uniform, small-sized chunks, each of which is stored as a subfile in the storage hierarchy. As mentioned above, if we do not employ any subfiling, a large file needs to be transferred from tape to disk. This increases both access latency and energy consumption. Therefore, subfiling is expected to bring energy benefits in both tape and disk accesses (though in this paper we focus only on the disk energy benefits). Then, an important job of the dynamic compilation framework is to determine the optimal chunk size and restructure the code on-the-fly based on it. Our approach achieves this by exploiting the data access pattern information. Specifically, the data access pattern information gives us the type and volume of data reuse. For example, if the accesses are localized (clustered) in small regions of the dataset, the chunk size should be kept small; otherwise, we can use a large chunk size. It should also be observed that using subfiling in conjunction with dynamic compilation brings an important advantage over the static compilation-directed subfiling. If we do not use dynamic compilation, then we are forced to select a specific chunk size (most probably based on the profile data), generate code customized for that size, and use that size throughout the execution. In comparison, with the dynamic compilation support, we can change the chunk size during the course of execution, thus better adapting to the dynamic changes in the I/O access patterns.

While dynamic compilation has the potential for improving the performance of I/O-intensive applications and reducing their energy consumptions, it also comes with its own costs that need to be accounted for. Therefore, our dynamic compilation framework should be selective in applying I/O optimizations. However, an overly selective compiler will not work well either as it can miss lots of optimization opportunities. Our approach maintains cost information within the metadata manager. This cost information consists of the time/energy overhead incurred for each I/O optimization for the last couple of invocations. When the next time the same I/O optimization is needed, the steering unit obtains this cost information from the metadata manager (through the performance tracer) quickly, and uses it in deciding whether the optimization in question should really be applied. A similar cost-benefit tradeoff is also carried out by the layout manager with one major difference. Unlike the dynamic compiler (which modifies the

generated code), the layout manager modifies the storage layout of the data. And, since a given dataset can be manipulated by different applications in different fashions, the changes to its layout should be performed with extreme care. A further argument for this is the fact that a typical layout change in the storage system can be much more expensive (in terms of the number of execution cycles it takes and energy consumption) than a typical dynamic code restructuring at run-time.

4 Concluding Remarks

This paper has presented the structure and operation of a dynamic compilation infrastructure that specifically targets I/O-intensive scientific applications. Focusing on the energy benefits of dynamic compilation in this application domain, we have described dynamic compilation framework that employs a suite of I/O optimizations, so that it allows I/O-intensive applications to optimize energy savings.

References

1. J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle, "Managing Energy and Server Resources in Hosting Centers," In *Proc. of the 18th Symposium on Operating Systems Principles*, pages 103-116, October 2001.
2. J. Chase and R. Doyle, "Balance of Power: Energy Management for Server Clusters," In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, page 165, May 2001.
3. L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani, "Efficient Organization and Access of Multi-Dimensional Datasets on Tertiary Storage Systems," *Information Systems Journal* 20(2): 155–183, 1995.
4. L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani, "Optimizing Tertiary Storage Organization and Access for Spatio-Temporal Datasets," In *Proc. of the NASA Goddard Conference on Mass Storage Systems*, 1995.
5. A. Choudhary, R. Thakur, R. Bordawekar, S. More, and S. Kutipidi, "PASSION: Optimized Parallel I/O," *IEEE Computer*, June 1996.
6. A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur, "PASSION: Parallel and Scalable Software for Input-Output," *NPAC Technical Report SCCS-636*, Syracuse, NY, September 1994.
7. R. A. Coyne, H. Hulen, and R. Watson, "The High-Performance Storage System," In *Proc. of Supercomputing*, Portland, OR, November 1993.
8. F. Darema, "Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements," In *International Conference on Computational Science*, pages 662–669, 2004.
9. F. Darema, "Dynamic Data Driven Applications Systems: New Capabilities for Application Simulations and Measurements," In *International Conference on Computational Science*, pages 610–615, 2005.
10. G. Gibson and R. Van Meter, "Network Attached Storage Architecture," *Communications of the ACM*, 43(11). November 2000.
11. G. Memik, M. Kandemir, A. Choudhary, "APRIL: A Run-Time Library for Tape Resident Data," In *Proc. of the NASA Goddard Conference on Mass Storage Systems and Technologies*, Baltimore, MD, April 2000.
12. K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," In *Proc. of Supercomputing*, San Diego, CA, December 1995.