Computer Architecture II

Author: Jose 胡冠洲 @ ShanghaiTech

Technical Paper Writing Streamline

Note Link

Weekly Reading Reports

See below.

Computer Architecture II: Reading Report 1

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the paper "Architecture of the IBM System/360" [1], which presents the groundbreaking contributions in general-purpose utility and inter-model compatibility of the IBM S/360* system.**

I. Introduction

Computer systems are designed to serve dedicated applications upon specific hardware models in 1950s. [2] As a big step of innovation, the birth of the IBM System/360 in 1964 marks the first appearance of a truly general-purpose computer system and the creation of *computer family* concept. The architectural structure of the IBM S/360 system is proposed in accordance with two major criteria:

- It must be **general-purpose**. It should be able to serve different scientific, real-time and commercial applications.
- It must achieve **compatibility** among the six sub-models with different scales in the same S/360 family.

Section below summarizes the key objectives of S/360 guided by the above two criteria. Detailed design principles and corresponding trade-offs encountered during its development are covered in the next section.

II. Key Objectives

This section extracts four key objectives of the IBM S/360 system, which lead the overall design and implementation of the system.

1) The system must be a versatile platform for users, in order to support different types of applications.
2) The system must separate its logical structure with the physical machine resources to achieve compatibility among machine models of different sizes.
3) The system should be designed as an *open-ended* framework, which allows future integration with new software and hardware technologies.
4) When the above three requirements are all satisfied, the most **efficient** design ("efficient" means higher performance with the same cost) should be adopted.

A visualization of conceptual structure of the IBM S/360 is given in figure 1, emphasizing its generalization both on application software interface and hardware model interface.

*S/360 is used here, and in following paragraphs, as an abbreviation for System/360.

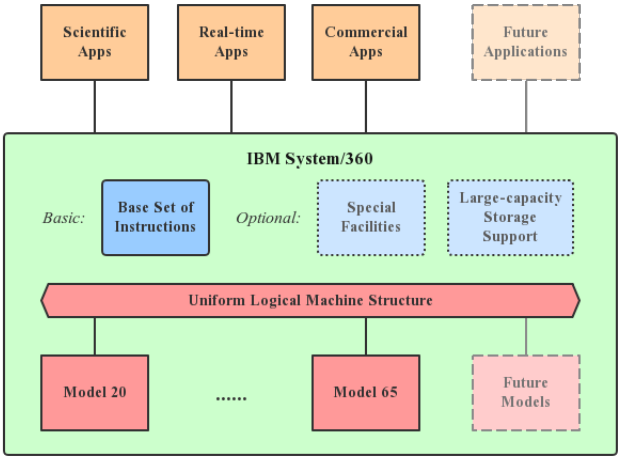

Fig. 1. Conceptual structure of the IBM System/360

III. Detailed Design

Detailed design principles of the IBM S/360 are summarized in this section. Each principle is followed by the most important decisions made when encountering corresponding trade-offs.

First of all, special facilities such as floating-point arithmetic are supported, though they are needed only in several specific applications. They are offered as options, but still integrated within the whole conceptual structure. Thus, the system is no longer optimized for certain usage, but obtains the ability to support multiple kinds of applications. Related decisions include using *addressed registers* instead of *pushdown stacks*, and supporting variable-length decimal fields.

Secondly, data format and granularity should be as flexible as possible both for user application and underlying machine instruction set architecture (ISA). As a result, it is possible to support different data formats and manipulations required by different applications, as well as being compatible among different machine models. Hence, they choose to adopt 4/8 character size, 32/64 floating point word length, and both ASCII and BCD codes as character code.

Thirdly, physical resources, such as registers, storage, and I/O devices, are virtualized as abstract components. Therefore a uniform logical structure of the system can be established, allowing the same program to execute on machines of different scales. This is called *strict program compatibility*. For

example, they decide to build channels that present a standard interface for the processor to communicate with various I/O devices.

Last but not least, special attention is paid towards the future trends that storage capacity is growing larger, and new I/O devices and CPU function schemes are emerging (which will possibly introduce brand-new instructions). With such anticipation, spare resources are reserved at all levels, such as spare bits in instructions, unused operation codes, and forward-looking support of larger memory space. [3] By this approach, S/360 makes it feasible to accept new devices, adopt new CPU instructions, and handle a bigger storage.*

IV. Conclusion

This report gives a brief summary of the key objectives and innovative design philosophies of the IBM System/360. The system is designed to offer general-purpose support for scientific, real-time and commercial applications. It provides a compatible logical structure for the six sub-models with different scales. Flexibility ranging from large-capacity storage support to instruction bits are also achieved. The IBM S/360 is a pioneer of applying abstraction in computer architectures, and significantly influences future computer systems design.

References

[1] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks Jr., "Architecture of the ibm system/360," *IBM Journal*, 1964.

[2] F. P. Brooks Jr., "Recent developments in computer organization," *Advances in Electronics*, 1963.

[3] F. Brooks, G. H. Mealy, B. I. Witt, and W. A. Clark, "The functional structure of os/360," *Pioneers and Their Contributions to Software Engineering*, 2001.

*Decisions which are discussed in the original paper but are left unmentioned in this section, are mainly the ones where both choices offer generalization and compatibility, but differs only in efficiency according to test statistics.

Computer Architecture II: Reading Report 2

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "The Future of Microprocessors" [1], which predicts that an inevitable paradigm shift towards *Chip Multiprocessors* (CMP) will be the future of microprocessors development*.**

I. Introduction

Since 1970s, the actual performance of microprocessors has increased tremendously, even faster than the prediction of Moore's law. [1] Such performance gain is achieved by applying user-transparent technologies. On the memory side, a larger cache hierarchy is introduced. On the processor side, longer pipelines and superscalar processors were developed, resulting in more powerful single processors.

Unfortunately, due to the limitations described in Section II, the attempt of enhancing individual processor performance has met a hard bottleneck. ***Chip Multiprocessor*** (**CMP**) architecture, which integrates multiple processor cores onto a single chip, will be the future direction instead.

II. Problem Background

Two traditional ways of enhancing the speed of processors, *pipelining* and *superscalar* technologies, both exploit *Instruction Level Parallelism* (ILP). Recently, the performance gain from ILP on a single processor is approaching its own limit. This bottleneck exists because of the following three reasons:

1) Degree of parallelism among instructions is typically no more than four, as observed in most application instruction streams. [2] Meanwhile, developing superscalar processors that can issue more instructions is extremely expensive. Thus, the superscalar technique can no longer bring obvious speedup.
2) Larger number of pipeline stages will make each separate stage too short to handle a reasonable task, and will bring huge complexity and overhead to the circuit. Thus, finer-grained pipelining is also not able to further increase single processor performance.
3) Pratical cooling systems will have difficulty serving even higher power consumption.

As a result, a higher level of parallelism should be introduced, which is *Thread Level Parallelism* (TLP) brought by the CMP architecture.

*Since this article is written in the year 2005, "future" here represents the trend after that timepoint.

III. Key Idea of CMP

The key idea of CMP is to gather multiple CPU cores onto one single chip. According to different caching and multithreading policies, CMP architecture can be roughly divided into the following three categories, as shown in b), c), and d), Figure 4 of the article. [1]

- **Simple chip multiprocessor**, where each core is single-threaded, and diffrent cores do not share on-chip cache.
- **Shared-cache** (often called *multicore processor* these days), where the cores share an on-chip *L2* cache.
- **Multithreaded, shared-cache**, where each core is furtherly *hyperthreaded*.

IV. Benefits of CMP

By integrating multiple cores onto one chip instead of barely putting several individual processors together, CMPs have the potential to make improvements for either *throughput-sensitive* or *latency-sensitive* conditions.

For throughput-sensitive workloads like web servers, tasks are relatively independent and possibly I/O-bound, and latency is not as critical. CMPs can bring significantly lower power consumption and better performance for such workloads. Firstly, for each core in the CMP, clock rate and superscalar level can be lowered, while the overall throughput is not influenced. This decreases the required voltage supply. Since

$$P \propto V^2,$$

where P is consumed power and V is supplied voltage, theoretically a processor only require $\frac{1}{k}$ of the original power to produce the same throughput, when k cores are integrated. Secondly, inter-processor communication will be much more efficient because of reduced distance and increased bandwidth between cores.

For latency-sensitive workloads like desktop applications, CMP is still the best goal to head for. Latency L can be modeled as:

$$L = \frac{\text{cycles}}{r \times s \times p},$$

where "cycles" is the fixed total number of clock cycles needed to complete the task, r is clock rate, s is superscalar level, and p is degree of multiprogramming. As r and s have both been pushed to the limit, multiprogramming (which increases p to more than 1) becomes the only solution. Compared to coding parallel programs across separate processors, multiprogramming on CMPs are much easier because on-chip cores share

the same memory. [3] Since that parallel programming is a necessary route, shifting to CMP architecture will reduce the cost and difficulty for users to handle it.

Apart from software's perspective, CMPs have advantages even for hardware developers. Without the need of frequently adjusting clock rates, they require less engineering effort during version updates. Similarly, the underlying system board does not require major modifications caused by core logic re-design.

V. Conclusion

In summary, future microprocessors will shift to Chip Multiprocessor architecture to overcome the performance bottleneck of single processors. CMPs can produce the same throughput with significantly lower power consumption. They can make multiprogramming much easier on shared memory, when user-defined parallelization is an inevitable trend. They can also simplify hardware updates between generations.

Such paradigm shift is a huge step in the history of microprocessors development, also a necessary one. Regarding software programmers, they also have to embrace multiprocessor programming techniques to get synchronized with this architectural transform, pushing code performance to a brand-new level.

VI. My Reflection

Looking back from 2019, this prediction made by Prof. Olukotun and Dr. Hammond at 2005 is quite forward-looking and accurate. Though such architectural design scheme is seldom mentioned as Chip Multiprocessor in recent years (instead, called *multicore processor* & *hyperthreading*), this paradigm actually gets widely accepted and implemented. Almost all modern processors are built upon this pattern.

However, fact is that we cannot keep plugging in more and more cores into a single chip, not to mention that the physical size of computer chips has been squeezed to quite a small level these days. Such fact indicates that performance gain achieved by CMPs is once again reaching a limitation. It is probably the time when we need new innovations in the field of processors, memory architecture, and I/O management, for example *in-memory / on-disk data pre-processing*, to fight for a higher peak of computation performance.

References

[1] K. Olukotun and L. Hammond, "The future of microprocessors," *ACM Queue*, 2005.

[2] D. W. Wall, "Limits of instruction-level parallelism," *WRL Research Report*, 1993.

[3] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun, "Transactional coherence and consistency: simplifying parallel hardware and software," *IEEE Micro*, 2004.

Computer Architecture II: Reading Report 3

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "An Overview of the Scala Programming Language". [1] Scala integrates object-oriented model with functional programming, and makes a step forward to building true component software systems.**

I. Introduction

One of the key points to industrialize software development is to provide a true component system, so that larger products can be easily assembled from pre-written libraries. This idea has been well established in hardware industry. On the contrary, a big proportion of software applications are written from scratch, partially due to lack of *component abstraction* in common programming languages.

The work of Scala aims at providing a *scalable* programming language which better supports component abstraction. This goal is achieved through two approaches:

- Integrate **object-oriented model** (OOP) and **functional programming** (FP).
- Provide **composition** and **decomposition** mechanisms, thus further supports implicit parameter inference.

The rest of this report will cover details of Scala's implementation platform and compatibility (see Section II), its multi-view abstraction design combining OOP and FP (see Section III), and its rich type system supporting component composition and decomposition (see Section IV).

II. Compatibility with Java

A pragmatic programming language should be easily applied to and vastly tested in real application development scenarios. In order to ease adoption by users, Scala is designed to execute upon Java Virtual Machine (JVM), thus has a great compatibility with Java and C#. However, Scala is not a superset of Java. It reinterprets Java's object hierarchy and type system, and adds in another view of functional programming abstraction.

III. Multi-view Abstractions

Designed as a statically typed language, Scala is the first one which integrates object-oriented programming and functional programming. Such integration brings two different perspectives of abstraction. First, OOP ensures "everything are objects", thus unifies numerical types, reference types, and user-defined types. Second, FP ensures "all operations are normal values", which brings great flexibility.

A. Object-oriented Abstraction

To follow the philosophy of object-oriented programming, Scala purifies Java's original object model. Every value, no matter it is a numerical value or a reference value (such as sequences), is an instance of an object. Consequently, every operation becomes a method call, even for variable access (for example, `x = e` is interpreted as `x_=(e)`). Users can take advantage of this design in two ways. Firstly, all user-defined classes have uniform formats and interfaces with embedded classes. Secondly, users can freely overload embedded operations to build fancy interfaces and implement class *properties*.

B. Functional Programming Abstraction

Guided by the idea of functional programming, every operation (function) is also a value, which can appear in formal parameters and wherever else a normal value variable can be. Furthermore, since every value is unified as an object through OOP abstraction, **every function is then an object** of certain type. This innovative design differentiates Scala with other programming languages. Functions can now have sub-classes extending their current abilities. *Generics* can be implemented in a consistent way with *abstract classes*.

A visual demonstration of objected-oriented and functional programming coupling is shown in Figure 1 below.

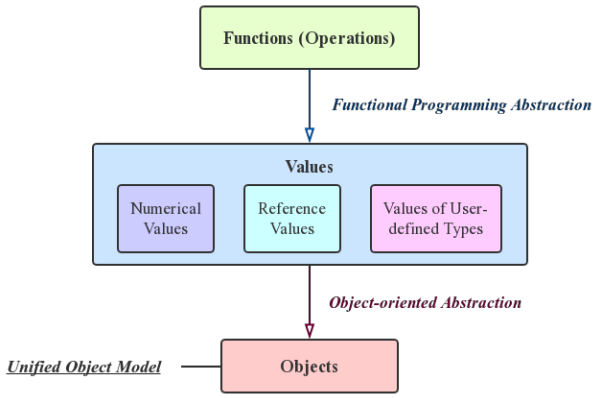

Fig. 1. Objected-oriented and Functional Programming Coupling

IV. Scala Type System

Based on the powerful abstractions, Scala provides a comprehensive type system that supports class composition and decomposition.

For class composition, Scala includes not only class *inheritance*, but also *traits*, *method shadowing*, *membership*, and *super calls*. The combination of these techniques is called *Mixin-class Composition*. With the help of mixin composition, Scala programmers can now do things like defining a base iterator class `BaseIter`, extending it to a generic `GenIter` and a specific `StrIter`, then combining their functionalities into class `Iter` by:

```
class Iter extends StrIter(args(0))
           with GenIter[char];
```

For class decomposition, Scala allows a *pattern matching* way of sub-typing. Instead of traditional object-oriented decomposition, where each sub-type must arrange their common methods in the same way as their parent, pattern matching decomposition uses a `match-case` expression on the input and maps different sub-types onto different implementations. This brand-new approach of decomposition is far less error-prone.

With the help of the above mechanisms, Scala successfully implements the `implicit` expression that automatically infers argument types.

V. Conclusion

To summarize, Scala is an innovative JVM programming language with multiple view of abstractions and a rich type system. Scala combines object-oriented programming model with functional programming, thus provides a highly uniform object model. Its type system provides comprehensive techniques for class composition and decomposition, therefore further supports implicit parameter inference. These characteristics of Scala make it possible to build true component systems.

Though Scala is a lab-born language, it has been put into practice in various projects, such as Facebook and Twitter. Its complicated characteristics put down more responsibility on library developers, meanwhile promote the idea of component abstraction and code reuse greatly.

References

[1] M. Odersky, P. Altherr, V. Cremet, and etc., "An overview of the scala programming language," *Technical Report LAMP REPORT*, 2006.

Computer Architecture II: Reading Report 4

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "Optimizing for Parallelism and Data Locality" [1], which presents a simple but accurate loop optimization algorithm to take advantage of both locality and parallelism at the same time.**

I. Introduction

Parallelism and *Data Locality* are the two major driving forces of compile-time loop optimization. Previous research used to exploit parallelism and data locality separately*. In contrast, this paper makes an effort to create a straight-forward loop optimization model which takes advantage of locality and parallelism simultaneously.

The rest of this report will cover the following aspects of the model:

- An overview of the optimization model (see Section II).
- Details about data locality optimization on the inner-most loop (see Section III).
- Details about parallelism scheduling on the outer-most loop (see Section IV).

Section V will summarize the experiment results and Section VI will conclude.

II. Model Overview

For simplicity, the model is designed to rearrange and adjust nested loops on shared-memory architecture for better performance. The model pursues two main objectives in concert. First, cache line reuse should be exploited efficiently. Second, parallelism should be introduced in large granularity, while not significantly lowering the degree of locality (for example, causing *false sharing*). Based on this guideline, the optimization algorithm is divided into two sequential phases, as demonstrated in Figure 1.

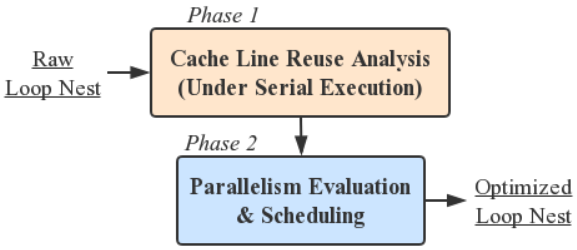

Fig. 1. Overall Algorithm Phases

*"*Previous research*" here refer to those before 1992.

Notice that parallelism is introduced and scheduled after inner loop locality optimization. This order is important when combining cache optimization with parallelization. If parallelism granularity and parallel loop order are fixed in the first stage, then we can hardly improve cache line reuse afterwards, since it mostly requires significant adjustment in loop permutation. On the opposite, introducing parallelism after locality optimization only requires slight modifications when the outer-most loop is non-parallelizable.

III. Data Locality Optimization

The first phase of the algorithm aims at optimizing data locality by improving the cache line reuse rate. Based on the observation that the inner-most loop often contains sufficient number of memory accesses which will completely flush the cache, the paper focuses on giving the lowest cost for the inner-most loop. This goal is achieved through the procedure shown in Figure 2.

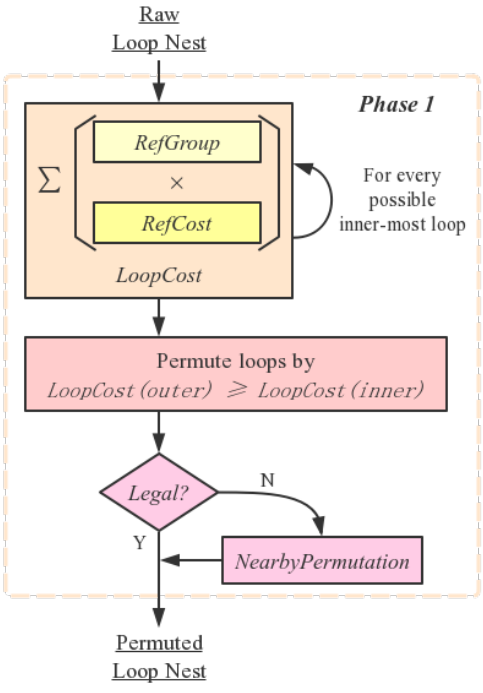

Fig. 2. Data Locality Optimization Phase Algorithm

For every loop l, all references to the same memory location captured when l is the innermost loop are grouped using

$RefGroup$. The groups are categorized into three different access patterns: *loop invariant*, *unit stride*, and *no reuse*, each contributing to a different amount of cost. After summing up the total cost for every possible inner-most loop, they are permuted in an order such that the inner loops cost less.

The permutation calculated above may not be legal. Under such circumstances, a nearby approximation needs to be generated.

IV. Parallelism Scheduling

The second phase of the algorithm evaluates the benefit of introducing parallelism on the outer-most loop, and schedules the parallelization while making as little effect as possible on the inner loop data locality. The algorithm logic is shown in Figure 3.

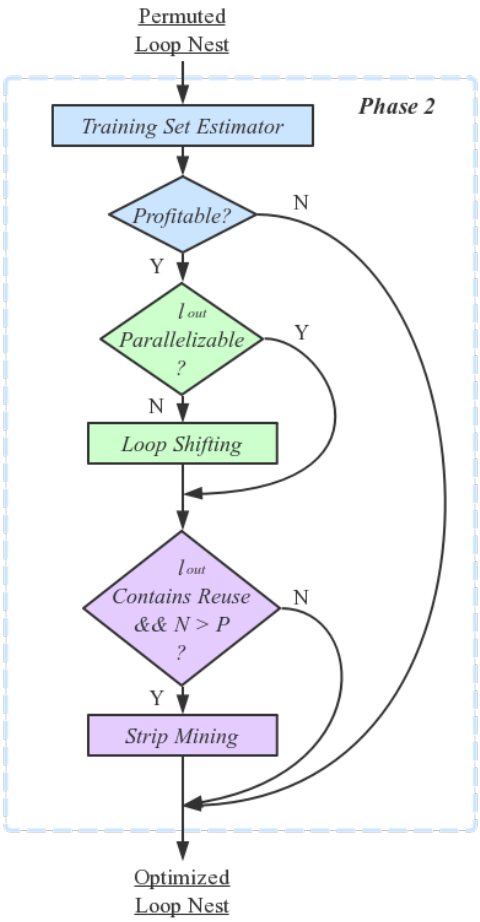

Fig. 3. Parallelization Phase Algorithm

One major innovation of this model is the *Training Set Estimator*. Whether or not to parallelize a certain loop significantly depends on the underlying machine hardware architecture. Thus, a set of sample training loops, whose performance has been pre-analyzed, will be executed on the target machine. Results will be used to estimate whether the current machine is profitable to parallelize the specific nested loops. The machine-independent estimator brings much better scalability to the model than other loop optimization algorithms. [2]

For loops that should benefit from parallelism, the model first checks if the outer-most loop is parallelizable. If not, a *Loop Shifting* will be conducted, moving a relatively inner loop to the surface. Afterwards, it *Strip Mines* the parallel loop to reduce the memory access range of each processor, thus making it more likely that the data accessed by one processor can wholy fit in its own cache.

V. Experiment Results

The model has been tested with matrix multiplication, *dmxpy* (a Linpack subroutine), and Erlebacher tri-diagonal solver applications. Under almost all these workloads, the loop permutation phase gives the desired permutations ranking. After the parallization phase, they all achieve a sub-linear speedup with the number of processors.

VI. Conclusion

The presented loop optimization model innovatively combines the strength of data locality and parallelism on shared memory multiprocessors. It improves cache line reuse on inner loops, and provides a high granularity of parallelism on outer loops. The model makes it possible to exploit data locality and parallelism at the same time, and illuminates the path that these two compile-time loop optimization technologies can be combined together to give existing code a higher performance.

References

[1] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *In Proceedings of the 1992 ACM International Conference on Supercomputing*, 1992, pp. 323–334.

[2] K. Kennedy, N. McIntosh, and K. S. McKinley, "Static performance estimation in a parallelizing compiler," *Technical Report, Dept. of Computer Science, Rice University*, pp. 91–174, 1991.

Computer Architecture II: Reading Report 5

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "GPFS: A Shared-Disk File System for Large Computing Clusters" [1], which describes the main features of *GPFS*, a massively scalable parallel cluster file system developed by IBM.**

I. Introduction

As modern computer clusters becoming larger and larger, a traditional file system can no longer manage the complicated storage issues effectively. Distributed local file systems can hardly cooperate with each other, while simple parallel file systems are not scalable enough.

GPFS (General Parallel File System) is a scalable parallel file system designed by IBM to serve large-scale **shared-disk** clusters. The rest of this report will cover the following essential features of GPFS:

- An overview of shared-disk architecture (see Section II).
- Exploiting **parallelism** in multi-aspects (see Section III).
- Details of **synchronization** strategies (see Section IV).
- Fault tolerance and **recovery** techniques (see Section V).

Then Section VI will conclude.

II. Shared-Disk Architecture

GPFS relies on shared-disk computer clusters. A typical shared-disk system deploying GPFS is shown in Figure 1.

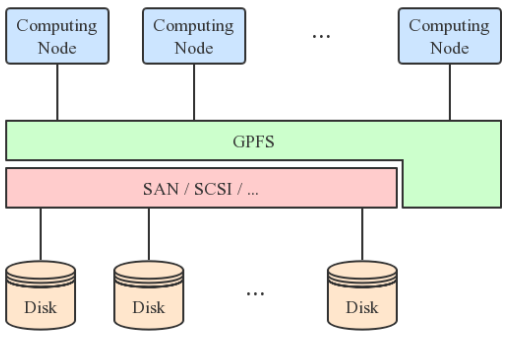

Fig. 1. Example of a Shared-Disk Architecture Deploying GPFS

The shared-disk approach brings two major advantages. First, GPFS can retain most of POSIX semantics and behave like a single node file system to users. Second, GPFS can achieve extreme scalability since all nodes have equal access to all disks.

III. Multi-Aspect Parallelism

GPFS exploits the power of I/O parallelism on large-scale clusters in the following four different aspects.

A. Strided Data Access

Data of a large file is stripped into equal sized strides and stored across multiple disks. Consequently, access to a single file can trigger simultaneous I/O pre-fetch on multiple back-end disks. Dirty buffers are also written back to disks in parallel. As long as the storage network bandwidth is sufficient, I/O throughput can be significantly improved.

B. Shared Write on Metadata

Concurrent accesses to a file can cause congestion on metadata modification. GPFS allows multiple nodes to write to its local cached copy of the file metadata. Updates are then forwarded to the dynamically chosen *metanode*, and merged into only one single modification which is actually written through to the disks.

C. Allocation Map Regions

Similarly, on file creation and deletion, the allocation map will become a conflicting resource. To optimize for this situation, the map is divided into k regions, each containing $1/k$ of the blocks on every disk. Regions are managed by an *allocation manager* node, who offers a non-locked region, instead of the entire map, to another node on request. This technology introduces parallelism into the allocation operation.

D. Parallel System Utilities

System-level utilities, such as disk reorganization and `fsck` consistency check, should also be scalable. For these tasks, GPFS assigns one of the nodes as *file system manager* who hands out tasks to other nodes on request, thus all nodes can work on subsets of files in parallel.

Some additional features, like *extensible hashing* directory entries and *online* disk reorganization, are also implemented in GPFS to support the large capacity and heavy workload brought by parallelism.

IV. Synchronization Strategies

Synchronization issues are handled in GPFS by combining two approaches together: *distributed locking* and *centralized management*.

A. Distributed Locking

File data access in GPFS is fundamentally based on byte-ranged distributed locking (in fact block-ranged in order to be compatible with allocations discussed in Section III-C). Each file is associated with a specific node m called *global lock manager*, who is responsible for handing out lock *tokens* to other nodes on request. A granted token represents the right of node n to read and modify a range of file blocks $[b_{start}, b_{end}]$, originally covering the whole file. If node n' requests a write to the locked region, meanwhile their destinations actually do not overlap, then the token splits into two, and the corresponding half is given to n'.

This locking scheme is called "distributed" since the locks are eventually held by each node distributively. Communication between the requester n' and the lock manager m will only involve one query and one lock information update per request. The actual token splitting task is done by the holder n. Thus, the global lock manager can afford a high degree of scalability and will not become a serious bottleneck when the system scales.

For small but frequently accessed files, such locking scheme can still result in catastrophic performance downgrade. *Data shipping* is used under these circumstances as a compensation.

B. Centralized Management

Since GPFS innovatively parallelizes metadata operations and disk block allocations, they also require proper synchronization. As stated in Section III-B and III-C, they are synchronized in a more centralized manner, using a metanode and an allocation manager. The reason is that these operations are small, frequent, and fragmented, so that distributed locking will cause many lock conflicts. A centralized manager, in contrast, will be able to schedule resources, avoid most of the conflicts, and gather and reduce individual results.

V. Fault Tolerance

In large-scale computer clusters, failures can happen frequently. For node failures, GPFS adopts write-ahead logs, which are widely used in journaling file systems like `ext` [2], to ensure the file system consistency. File operations from a node must first append a corresponding transaction in its log before the actual execution. Since disks are shared, a survival node can recover the log transactions on behalf of the failed node. Special roles played by the failed node, such as a lock manager, will also be taken over. For communication failures, the majority group will continue to function as a cluster, while the rest of the nodes wait until communication recovery. For disk failures, GPFS mainly utilizes RAID for error recovery. Manual replication is also supported but rarely enabled.

VI. Conclusion

GPFS is a collection of many brilliant ideas in file system design. It exploits parallelism not only in data accesses, but also in metadata operations, disk block allocations, and administrative utilities. It takes advantage of both distributed locking and centralized management to maintain synchronization correctness. It also adopts write-ahead logs to achieve fault tolerance.

GPFS has been installed and tested on the largest super computers in the world*. It makes a big step forward in parallel file system scalability and stability. Up to now (the year 2019), GPFS and its evolutionary version *IBM Specturm Scale* are still widely installed on high performance computer clusters.

References

[1] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *FAST*, 2002.

[2] Wikipedia, "Journaling file system — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Journaling\%20file\%20system&oldid=869377481, 2019, [Online; accessed 21-March-2019].

*In the year 2002 when this paper is published, the largest cluster deploying GPFS refers to *IBM ASCI White* [1].

Computer Architecture II: Reading Report 6

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "24/7 Characterization of Petascale IO Workloads" [1]. It presents a scalable I/O profiling tool, *Darshan*, which captures application-level I/O patterns meanwhile introducing negligible overhead to the system.**

I. Introduction

Existing I/O profiling tools go into two extreme categories. The first category logs every individual I/O operation and does not provide postprocessing. They introduce considerable overhead, thus are not scalable enough. The second category adopts sampling techniques to avoid huge overhead, however sacrifices the ability to capture user access patterns. With the scale of computation growing rapidly nowadays, an I/O characterization tool which **combines application-level reflection with scalability** is in need.

The paper presents a new I/O profiling tool, *Darshan*, which possesses the following three advantages. Firstly, it captures user-level file access patterns. Secondly, it is transparent to users, leaving no need for application API modifications. Thirdly, it achieves peta-scale scalability, making it possible to deploy Darshan on the world's largest HPC systems.

The rest of this report will cover:

- An overview of Darshan tool (see Section II).
- Implementation details of Darshan (see Section III).
- Test results on real machines (see Section IV).

Section V concludes, and Section VI contains my own reflection on this paper.

II. Overview of Darshan

Darshan is implemented as a set of user space libraries. It currently supports capturing MPI-IO routines and POSIX routines. MPI-IO routines are supervised through PMPI interface, while POSIX routines are traced by wrapping over original system call functions.

Upon successful linking, user space library organization ensures Darshan's user transparency. The logging is conducted by each process individually when it issues an I/O operation. Results will be gathered, reduced, and written to disk in a specified format at process termination.

III. Implementation Details

For every process, Darshan records the captured I/O information into *file record* structures, which reside in the process memory space. Each file record corresponds to one physical file that the process interacts with, and can be indexed through file name, file descriptor, or MPI file handle hashing.

Inside the file record, an array of counters are maintained. Users can specify what information should be tracked as counters, for example number of POSIX operations and total bytes written. Additionally, access sizes and stride sizes* are kept in two red-black trees respectively for pattern characterization analysis. A demonstration of Darshan file record organization is given in Figure 1.

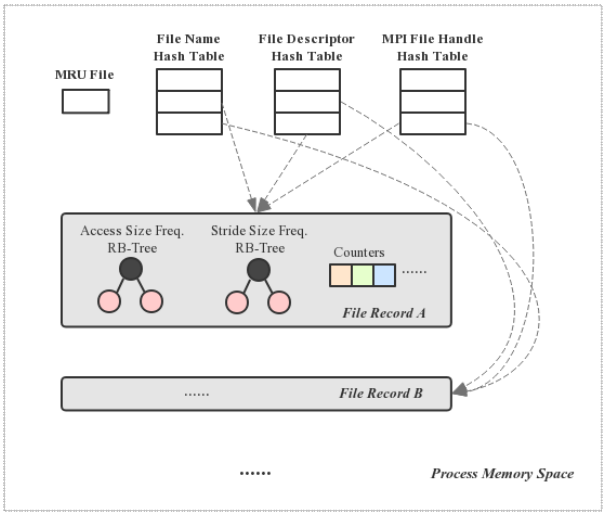

Fig. 1. Darshan File Record Organization

To achieve higher scalability, file records are not collected until process termination. Thus, extra I/O communications only occur when a process finishes, introducing much less overhead throughout most of the execution time. Reduction on shared file records across MPI processes, and output data compression, will both be conducted to minimize the write size. Furthermore, users are allowed to set the limit of number of files to trace by each process, and what counters to keep track of in a file record, to tune a better performance for user-specific workloads.

IV. Real Machine Testing

Darshan has been deployed on LANL IBM BlueGene/P supercomputing system, and tested on four different user

* *Stride* used here to indicate an interval between accesses. This is a totally independent concept with respect to *data striding*.

benchmarks or applications: *MADBench2*, *Combo I/O*, *S3D-IO*, and *HOMME*. Test results reveal that Darshan introduces little runtime overhead in either function wrapping execution time ($< 0.52\%$) and memory occupation ($< 0.2\%$). Also, log output optimizations suppress extra write-back time below 7 seconds for various job sizes. These advantages make Darshan capable of handling peta-scale workloads of over 65,536 processes.

V. Conclusion

In summary, Darshan is a highly scalable I/O characterization tool which tracks user application I/O access patterns, meanwhile introducing little overhead into the system. Darshan makes it possible to contiguously trace user-level I/O operations on large-scale supercomputing systems. These large-scale comprehensive traces will promote the storage community's understanding of how modern computational science applications interact with storage, therefore illuminates the future path of high-performance storage techniques.

VI. My Reflections

Though this paper asserts that Darshan is scalable towards peta-scale supercomputing clusters, the test result it provides is significantly insufficient. The four case studies conducted on IBM BG/P system only proved its availability and correctness, but did not give evidence of memory and latency overhead. The only scalability test is conducted with a specific 65,536 processes workload of *MADBench2*, which is not sufficient for inferring its capability of handling peta-scale supercomputing workloads. Making it worse, even this single test result (shown in Figure 2) is a bit suspicious.

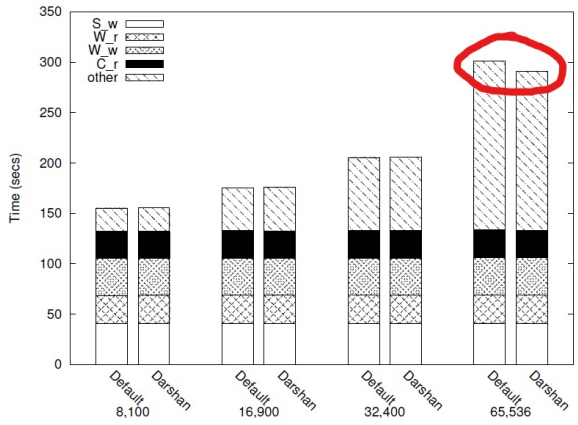

Fig. 2. Scalability Test Result from Philip H. Carns et al. [1]

The figure above shows that Darshan processing time is even shorter than the default program with 65,536 processes, with no reasonable explanation in the paper. I doubt the true scalability of Darshan tool, and hope for more detailed and comprehensive test results on various large-scale workloads.

References

[1] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, 2009.

Computer Architecture II: Reading Report 7

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "Adaptable, Metadata Rich IO Methods for Portable High Performance IO" [1]. It presents an adaptable high-performance I/O framework called *ADIOS*, which provides a portable and efficient interface between scientific computing applications and low-level I/O libraries.**

I. Introduction

Various I/O libraries, such as *HDF5* and *NetCDF*, have been developed to serve high-performance scientific computing. However, for different applications running on different hardware platforms, different I/O libraries show different performance. With the variety of HPC hardware resources increasing, an abstract layer that wraps over low-level I/O methods is in need.

This paper presents ADIOS (ADaptive I/O System), an I/O method abstraction layer developed by Oak Ridge National Lab that provides portable switching among I/O libraries and significantly improves the I/O performance of scientific computing applications. The rest of this report will cover the following aspects of ADIOS:

- An overview of ADIOS architecture (see Section II).
- Intermediate BP format representation (see Section III).
- Run-time selection of I/O methods (see Section IV).
- Data characteristics collection (see Section V).

Section VI will then conclude.

II. Overview of ADIOS

ADIOS is designed as a high-level I/O library that **redefines all the file interfaces** between user applications and lower-level I/O libraries. Demonstration of ADIOS architecture is given in Figure 1 below.

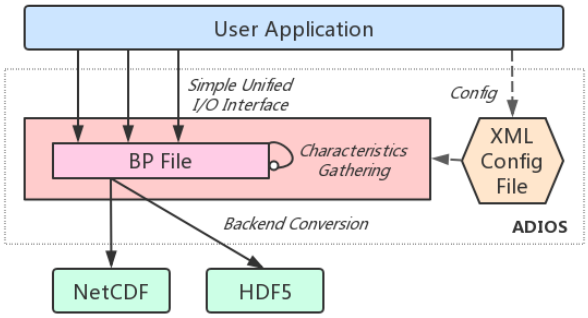

Fig. 1. Demonstration of ADIOS Architecture

Though such design makes ADIOS intransparent to users, it minimizes user code modifications by combining simple I/O methods API with an external *XML* configuration file. After specifying the grouping of data and which I/O approach to use for each group, users only need to invoke extremely simple methods for file operations. Actual I/O methods are generated automatically based on pre-defined configuration.

For a running application, all I/O operations are directly interfering with an intermediate *BP*-format file. With the help of carefully-designed BP format (discussed in details in Section III), ADIOS gets rid of redundant consistency checks and achieves seamless switching among backend I/O libraries.

III. BP-Format Representation

The BP intermediate representation format is organized as Figure 2. BP files act as a middle layer between user I/O operations and the actual files with chosen low-level format. Such organization brings three major advantages:

1) **High degree of parallelism**. All processes write to totally independent regions, therefore introduce no overhead for synchronizations.
2) **Convertible**. BP files can be easily converted to low-level formats such as `.nc` (of NetCDF).
3) **Avoids redundant consistency checks**. Original implementation of I/O libraries like HDF5 normally requires consistency checks on every write operation. By plugging in the BP intermediate layer, only a single consistency check is needed at file conversion. This feature significantly improves the I/O performance.

Process Group 1	Process Group 2	...	Process Group n	Process Group Index	Vars Index	Attributes Index	Index Offsets and Version #

Fig. 2. BP Format Given by the Paper [1]

This BP format organization is a major evolution from previous releases of ADIOS. [2] [3]

IV. Run-time Selection of I/O Methods

With the help of BP convertibility, run-time selection among different low-level I/O libraries becomes possible. Only a simple modification in the configuration file is need for changing the I/O methods used for a certain group of data. Thus, on HPC platforms with different hardware architectures, users can easily tune the I/O performance without touching application source code.

V. Data Characteristics

ADIOS also provides an additional functionality that collects data characteristics statistics, for example number of total variables or the maximum element in an array. Characteristics are maintained within the BP file at both process data segment and index segment, in order to ensure file consistency, and to ease metadata generation. Theses characteristics can simplify subsequent data analysis tasks.

VI. Conclusion

In summary, ADIOS is an adaptive high-level I/O framework that serves I/O method tuning for scientific computing applications. It wraps over common low-level I/O libraries and provides a brand-new unified interface for users. It introduces a BP file intermediate layer to parallelize I/O operations and avoid redundant consistency checks. Through simple modifications of the configuration file, users can easily switch among different I/O methods for better performance. Extra data characteristics can also be recorded for later analysis.

Tests on *Chimera supernova* and *GTS fusion* applications show surprising write performance gain (maximum over 1400/120) achieved by ADIOS. [1] The future releases of ADIOS will focus on read optimizations, a better BP format, and support for more low-level I/O libraries.

References

[1] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich io methods for portable high performance io," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10.

[2] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, ser. CLADE '08. New York, NY, USA: ACM, 2008, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1383529.1383533

[3] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, W. (georgia Tech, R. O. (s, and I. N. Laboratories, "Adaptive io system (adios)."

Computer Architecture II: Reading Report 8

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "Parallel netCDF: A High-Performance Scientific I/O Interface" [1], which presents a set of parallel I/O interfaces for netCDF scientific data library, called *PnetCDF*.**

I. Introduction

Portable I/O libraries, such as HDF and netCDF, have been widely used in modern scientific computing applications. Though HDF5 has already adopted parallel I/O access, netCDF is still stuck in serial I/O methods*, which becomes a bottleneck for large-scale datasets. Considering that netCDF is popular in the field of earth modeling and simulation, developing a parallel I/O interface for netCDF format is in urgence.

The paper presents **PnetCDF**, a set of parallel netCDF I/O interfaces built upon MPI-IO methods. It exploits parallelization in netCDF file operations, while only introducing minor modifications in user API. It brings tremendous performance gain for netCDF applications. The rest of this report will cover:

- Original netCDF design (see Section II).
- Overview of PnetCDF structure (see Section III).
- Implementation of PnetCDF (see section IV).
- Advantages and performance evaluation (see Section V).

Section VI will then conclude, and my own reflections will be given.

II. Original NetCDF

NetCDF is a widely-used I/O utility for self-describing, structured, array-based datasets. It is originally developed by Unidata community of National Center for Atmospheric Research (NCAR), and now becomes the most popular I/O library for earth and atmospheric science applications. [2] NetCDF consists of a portable data file format (`.nc`) and a set of programming APIs for corresponding file interactions.

Data is arranged as variables in pre-defined dimensions in netCDF. PnetCDF is based on netCDF-3 format (*classic netCDF*). File header contains file metadata, dimension declarations, and their attributes. Following are fixed-dimension variables, each occupying a determined amount of space. Variables with the same free (unlimited) dimension are appended to the end of file. No more free dimensions are allowed. Figure 1 gives a demonstration of netCDF-3 file construction.

*This comment is given from the perspective of year 2003, when the paper is published.

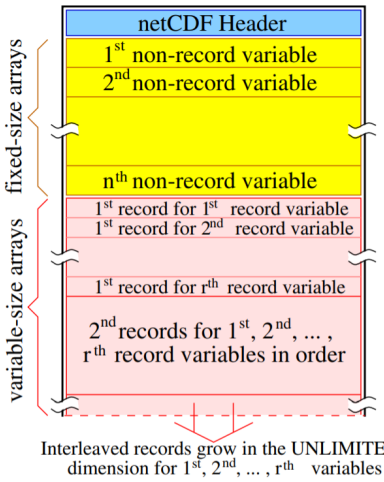

Fig. 1. NetCDF File Construction [1]

Original netCDF I/O interface only supports serial accesses through native I/O system calls. With data scale rapidly increasing, potential parallelism should be exploited to improve storage bandwidth and thus accelerate those scientific computing tasks over netCDF.

III. PnetCDF Structure

PnetCDF retains netCDF-3 file format and redesigns netCDF I/O interfaces on top of MPI-IO protocol. Such design not only extends netCDF to support parallel I/O among user processes, but also takes advantage of existing I/O optimizations from MPI-IO implementations. Parallel netCDF provides a unified intermediate layer for all user processes running the same application, as shown in Figure 2.

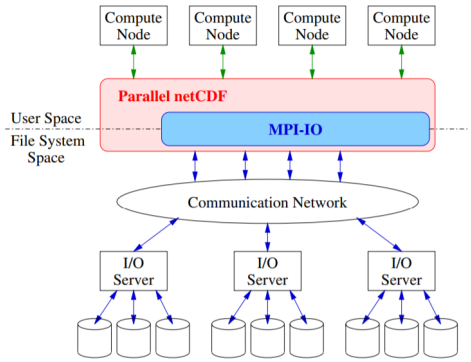

Fig. 2. Parallel NetCDF Architecture [1]

For netCDF file creation/open operations, the interface accepts a MPI communicator group as an additional argument. It specifies which processes are involved in interacting with this file. Data access operations, such as reading and writing a variable, are extended into two different sets of API. The *high-level* API follows original netCDF functions to serve easy accesses to contiguous regions, while the *flexible* API provides low-level MPI-style functions to handle accesses to non-contiguous regions.

IV. Parallel Implementation

On the top level, PnetCDF provides a new collection of netCDF user I/O interfaces, where each function name is prefixed with "`ncmpi_`". Under the bottom level, PnetCDF invokes the ROMIO implementation of MPI-IO. Features like collective I/O (I/O forwarding) used in ROMIO can thus bring further bandwidth improvement other than parallelization.

Every process maintains a cached copy of file header in its local memory during the I/O procedure, therefore metadataset operations no longer needs inter-process communication. Consistency is ensured by MPI convention that all processes should always call a MPI function with exactly the same arguments.

For actual data access, each process is assigned a different MPI file view corresponding to a different segment of the file body. Users can give extra MPI hints to guide the collective I/O, when they are accessing non-contiguous variable records.

V. Evaluation

PnetCDF offers the following two advantages compared to serial netCDF and HDF5:

1) Due to the straight-forward format of netCDF files, mature MPI-IO methods are easily applied without adding complicated transformation layers. Thus, only a minor amount of overhead is introduced.
2) File header caching removes most of the need of metadata synchronization, therefore metadata updates are less likely to be the I/O bottleneck.

Experiments have been conducted on IBM SP-2 machines. Scalability tests over a synthetic three-dimensional dataset show that PnetCDF significantly enlarges read/write bandwidth. Even when there is only one process, write performance is improved with the help of collective I/O. Comparison tests over FLASH I/O benchmark also indicate that PnetCDF brings less overhead than HDF5.

VI. Conclusion and Reflections

In summary, PnetCDF provides a new set of parallel I/O interfaces for the netCDF library. It perfectly couples netCDF file format with underlying MPI-IO methods to achieve significant performance optimizations. The successful implementation of PnetCDF makes netCDF a more powerful tool for large-scale, structured data storage on modern machines.

Looking back from 2019, there is a little pity that parallel netCDF still only supports netCDF-3 (classic) format [3], which is getting outdated. New netCDF-4 standard supports multiple unlimited dimensions, which is much more flexible. However, current netCDF-4 implementation is built upon a HDF5 intermediate layer. Possibilities exist that PnetCDF can be extended to support netCDF-4 to achieve both high performance and high flexibility.

References

[1] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Nov 2003, pp. 39–39.

[2] "Netcdf url," https://www.unidata.ucar.edu/software/netcdf/.

[3] "Netcdf architecture," https://www.unidata.ucar.edu/software/netcdf/docs/netcdf_introduction.html#architecture.

Computer Architecture II: Reading Report 9

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

Abstract*—This report is a brief summary of the article "zFS: A Scalable Distributed File System Using Object Disks" [1]. It introduces an IBM research project called *zFS*, an object disk-based file system that integrates a cooperative memory cache and is highly scalable.

I. Introduction

With commodity Object Store Devices (OSDs) entering the storage market, a reliable and scalable file system for object disks is in need. To dig the potential of decentralization in OSDs, zFS is proposed as a highly distributed file system for off-the-shelf commodity OSDs.

zFS integrates all individual machines' memory into a **global cache**. It also applies **distributed managers** for data and metadata accesses. The rest of this report covers the following aspects of zFS:

- Overview of zFS architecture (see Section II).
- Cooperative Global Cache (see Section III).
- Distributed Management Schemes (see Section IV).

Then Section V will conclude.

II. zFS Architecture

zFS separates high-level file management apart from low-level storage devices.

A. Low-level Disk Management

For every object disk in the storage system, zFS deploys a local *Object Store* (`OSD`) to interact with the disk. File data is indexed through `(obs, oid)` pairs, where `obs` is object disk id, and `oid` is object id on the corresponding disk. `OSD`s provide upper modules a uniform API to interact with file objects.

B. High-level File Modules

The essential part of zFS lies in its dedicated high-level file management modules. There are five main modules involved:

1) *Front End* (`FE`): An `FE` instance is running in every client node kernel. It presents a standard POSIX API for users to conduct I/O with zFS.
2) *Lease Manager* (`LMGR`): For every underlying OSD, there exists a corresponding `LMGR`. Holder of an OSD major lease acts as the `LMGR` for that disk, thus is responsible for sending out and revoking object leases regarding that disk. Leases (locks with certain expiration period) are used here to accommodate resource failures.
3) *File Manager* (`FMGR`): Each opened file is linked with a `FMGR`, held by the first machine to open the file. `FMGR`s are responsible for listening to user requests from `FE`, interact with corresponding `LMGR`s, and translate the operations to `OSD`s.
4) *Transaction Server* (`TSVR`): A `TSVR` resides in every client node, which listens to `FE` for directory operations (i.e. metadata operations like `rename()`), and conduct them as proper transactions.
5) *Cooperative Cache* (`Cache`): zFS innovatively combines all client memory as a global cooperative cache. Details are discussed in Section III.

Organization of zFS components and their intercommunications are demonstrated in Figure 1.

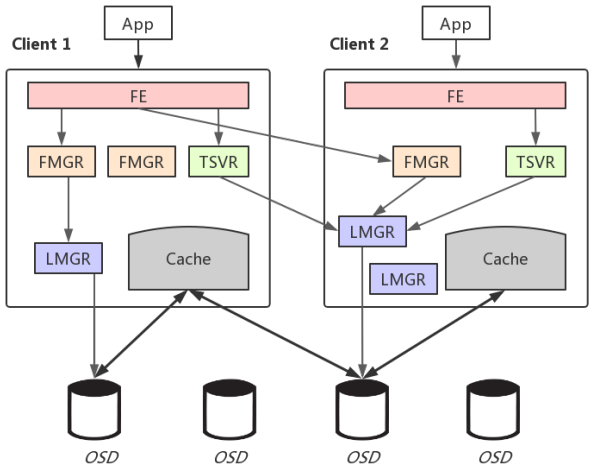

Fig. 1. zFS Architecture Design

III. Cooperative Cache

One major innovation of zFS project is the design of cooperative cache. By integrating all individual machines' memory into a global page cache, `read()` performance can get improved. If one client is reading a file object that is currently cached in another machine, it fetches data directly from the remote memory over RDMA, instead of going all the way through to disks.

However, since the paper does not include any test results, scalability of such cooperative cache design should be ques-

*Attention: This project is not related to ZFS (*Zettabyte File System*) developed by Sun Microsystems.

tioned. As the cluster scale going up (towards several thousand machines, which zFS claims to support), cache coherence overhead will significantly increase. Locating which memory does an object reside in, and dealing with write-backs at file `write()`, will both degrade the performance. Further explanation of the cooperative cache design is expected.

IV. Distributed Managers

Another main advantage of zFS is the pure distributed management scheme for files and objects. Distributed file managers, lease managers, and transaction servers make every I/O operation in zFS a decentralized procedure. Under real-world balanced workloads, such scheme can bring high performance together with high scalability. IBM GPFS adopts a similar design of distributed lock managers, but with the help of OSDs, zFS extends it to an extreme.

V. Conclusion and Reflections

To summarize, zFS is a distributed file system designed for commodity object storage devices. It integrates client memory as a global cooperative cache. It decentralizes data accesses, metadata operations, and synchronizations by using distributed file managers, transaction servers, and lease managers. zFS opens the way of designing large-scale file systems for object disks.

This paper focuses on theoretical design of zFS, instead of actual implementation and tests. No experiments are conducted to support the strength and robustness of zFS design, especially for scalability of the cooperative cache system. Further test results are expected to prove the value of zFS and its advantages over other existing file systems.

References

[1] O. Rodeh and A. Teperman, "zfs - a scalable distributed file system using object disks," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, April 2003, pp. 207–218.

Computer Architecture II: Reading Report 10

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the short paper "Recent Progress in Tuning Performance of Large-scale I/O with Parallel HDF5" [1], which summarizes current work on tuning HDF5 performance for scientific computing applications MOAB and VPIC.**

I. Introduction

Large-scale scientific applications put tremendous pressure on the underlying storage system. In simulation phases, massive amount of data needs to be stored in files. In analysis phases, data files are read through different access patterns. Parallel I/O optimizations thus become essential to modern high-performance computing.

HDF5 is a widely-used I/O optimization library, which supports mainstream programming languages and is portable on various kinds of hardware platforms [2]. Parallel I/O in HDF5 is achieved through collective MPI-IO. This short paper presents recent progress on optimizing HDF5 performance for two scientific computing applications:

- **M**esh-**O**riented Dat**AB**ase (MOAB, see Section II).
- **V**ector **P**article-**I**n-**C**ell (VPIC, see Section III).

II. MOAB on Mira

Mesh-Oriented Database (MOAB) is package for operating mesh data [3]. It originally uses HDF5 to store and represent mesh data structures.

Problem. MOAB assigns each process to read from a different coordinate in the mesh. This results in poor locality in HDF5 file accesses, thus cannot benefit from collective I/O.

Tuning. In HDF5, subset of data is accessed through *hyperslabs*. An improved hyperslab selection algorithm that merges multiple non-contiguous hyperslabs into a *hyperslam* can significantly reduce the number of read calls. On Mira system of Argonne National Lab (ANL), this approach achieves 10x performance gain.

III. VPIC on Blue Waters

Vector Particle-in-Cell (VPIC) is an application for simulating plasma physics phenomenon. Its VPIC-IO kernel is also built upon HDF5.

Problem. As the number of processes increase, more particles are written simultaneously into the same HDF5 data file. This introduces a potential I/O bottleneck.

Tuning. Two techniques are proposed to solve improve VPIC-IO performance. Firstly, write load among processes and among I/O servers are re-distributed. Secondly, a new feature called *multi-dataset writes* is introduced into HDF5, so that different datasets can be stored in one HDF5 file without serial collective operations. On Blue Waters system of the National Center for Supercomputing Applications (NCSA), such approach achieves at least 2x speedup.

IV. Conclusion and Reflections

Modern scientific computing applications are becoming more complicated and more personalized. Though parallel I/O libraries like HDF5 can provide a uniform storage optimization scheme, we can never ignore the importance of tuning I/O operations for individual applications. In order to push the throughput of scientific computing to an extreme, user-specific I/O pattern analysis and performance tuning should always play an important role.

References

[1] M. S. Breitenfeld, K. Chadalavada, R. Sisneros, S. Byna, Q. Koziol, N. Fortner, M. Prabhat, and V. Vishwanath, "Recent progress in tuning performance of large-scale i/o with parallel hdf5," in *9th Parallel Data Storage Workshop (PDSW 2014)*, 2014.

[2] (2017) Hdf5 introduction. [Online]. Available: https://support.hdfgroup.org/HDF5/whatishdf5.html

[3] (2019) Moab:mesh-oriented database. [Online]. Available: https://bitbucket.org/fathomteam/moab

Computer Architecture II: Reading Report 11

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—This report is a brief summary of the article "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" [1]. It presents a distributed memory abstraction model called *Resilient Distributed Datasets* (RDDs). It implements RDDs into a framework called *Spark* that enables fault-tolerant, efficient in-memory cluster computing.**

I. Introduction

Existing high-level cluster computing frameworks (like Hadoop MapReduce [2]) are inefficient in reusing intermediate results. At the end of a computation stage, all generated results need to be written back to external stable storage such as disks. To overcome this problem, many domain-specific in-memory frameworks have been developed. They pursue high performance by sacrificing generality.

This paper presents *Resilient Distributed Datasets* (RDDs) that abstract distributed memory in a **coarse-grained** way. RDDs allow **lazy computation** and **efficient fault recovery** with the help of *lineage* logging. RDDs naturally improve the performance of iterative algorithms over batch operations. The rest of this report will cover:

- RDD model of data abstraction (see Section II).
- The *Spark* system that implements RDD with Scala programming language (see Section III).
- Performance evaluation and results (see Section IV).

Section V will then conclude and contain my own reflections.

II. RDD Abstraction

An RDD is a **read-only**, **partitioned** collection of data records. Different partitions may reside in memory of different nodes. RDDs only support the following two sets of **coarse-grained** operations:

1) *Transformation*: generate a new RDD from a data file in external storage, or from an existing RDD. Examples include `map`, `filter`, and `flatMap`.
2) *Action*: reduce an RDD into a return value, or export it to external storage. Examples include `count`, `collect`, and `save`.

Its read-only and coarse-grained property ensures that any algorithm based on RDD model should follow the pattern: 1. create initial RDDs from existing data $\rightarrow$ 2. do a chain of transformations to get a target RDD $\rightarrow$ 3. reduce/collect/save the target RDD to get the result. A classic example is the PageRank algorithm demonstrated in Figure 1. Every rectangle represents a different RDD, and every edge indicates a transformation.

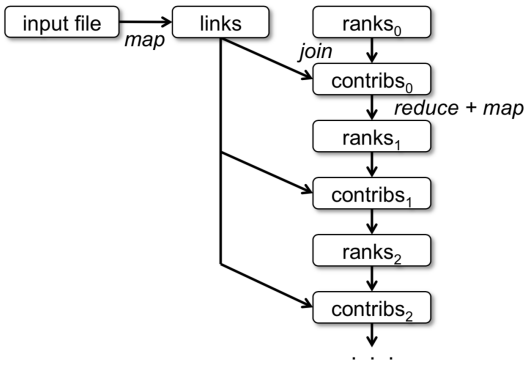

Fig. 1. PageRank Algorithm using RDDs (from the paper [1])

Since RDDs are obtained from transformation chains, their corresponding dependencies form a directed acylic graph (DAG) as shown in Figure 1. This graph is called *lineage* graph, and plays an important role in fault recovery which will be discussed in Section III.

For algorithms based on bulk operations (applying the same operation to every piece of data), RDDs can provide a straightforward computation logic. However, RDDs are not suitable for fine-grained algorithms, which frequently updates small proportions of data.

III. Spark Framework Implementation

Based on the aforementioned RDD model, they implemented an **in-memory** cluster computing framework called *Spark* (open-sourced at [3]) using the *Scala* programming language. Spark can be deployed on a distributed computer cluster. Users issue a Spark program from a driver node. A global scheduler arranges tasks into stages and is responsible for data distribution across worker nodes. When the computation finishes, distributed RDDs are reduced into a return value/stored back to external storage. Spark runtime architecture is shown in Figure 2.

Three aspects of the Spark system are worth discussing: job scheduling, memory management, and fault tolerance.

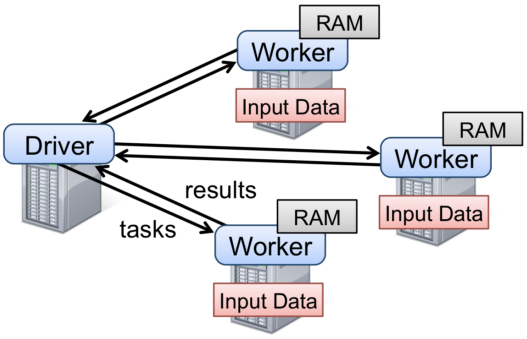

Fig. 2. Spark Runtime Architecture (from the paper [1])

A. Job Scheduling

As mentioned in Section II, an RDD is partitioned across nodes, and transformations result in lineage. We can classify transformations into two categories according to how different partitions of parent and child RDD interact with each other. In *narrow* transformations, one parent partition is required by only one child partition (for example `map`). In *wide* transformations, one parent partition may be accessed by multiple child partitions. Spark job scheduler divides a program into stages based on theses two different kinds of dependencies. Firstly, it groups as many narrow transformations as possible into one stage, where different partitions can be pipelined and processed efficiently. Secondly, wide transformations only occur across stages.

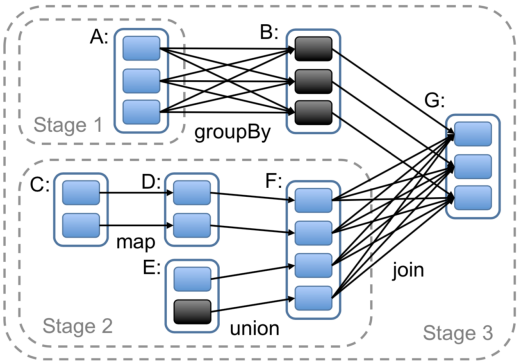

Fig. 3. Spark Job Scheduling as Stages (from the paper [1])

B. Memory Management

Since memory size is limited, it is possible that we run out of memory for large tasks. Spark adopts an LRU policy on evicting partitions to external storage. Though evicting part of the data out of memory will affect performance, it is still no worse than traditional frameworks, as they store all intermediate results in external storage.

C. Efficient Fault Tolerance

RDDs are computed lazily until users invoke actions. During the computation, if a certain partition fails, we can simply trace back through the lineage graph and recompute that failed partition. Compared to traditional fault recovery mechanisms, such as copying data replicas from other nodes, this approach significantly lowers the overhead brought by fault recovery. This property of RDD makes Spark an extremely robust cluster computing system.

IV. Performance Evaluation

From the design of RDD model, we can see that it is specifically optimized for **iterative** algorithms over distributed memory clusters. Test results show that Spark outperforms Hadoop MapReduce by 20x in iterative machine learning algorithms and graph applications. Recovery of a node failure in K-Means algorithm only introduces less than 25 seconds of overhead in the failed iteration. They also integrated Spark into the interactive Scala command line, and achieved only 5-7 seconds of latency to query a 1 TB dataset.

V. Conclusion & Reflections

This paper presents Resilient Distributed Datasets (RDDs), a general-purpose distributed memory abstraction for cluster computing. RDDs enable in-memory computation on bulks of data by applying coarse-grained transformations. RDDs also allow efficient fault recovery in the middle of a task. They implemented RDDs in a system called Spark, and achieved tremendous speedup in iterative big-data applications, compared to traditional cluster computing frameworks like Hadoop MapReduce.

The Spark framework is a remarkable milestone in the field of distributed cluster computing. It is the first work to address both in-memory cluster computing and generality. Spark is now integrated into the Hadoop family, and widely deployed by big companies due to its simplicity. However, it also suffers from a major drawback that its memory occupation is extremely high. With the size of data scaling up nowadays, Spark can no longer satisfy cutting-edge requirements in big-data analysis. Domain-specific systems are still the first choice to achieve acceptable performance, meanwhile using a limited amount of hardware resource.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[2] (2019) Apache hadoop release 3.1.2. [Online]. Available: https://hadoop.apache.org/

[3] (2019) Apache spark release 2.4.2. [Online]. Available: http://spark.apache.org/