Opportunistic data locality for end user data analysis

Opportunistic data locality for end user data analysis

M Fischer, C Heidecker, E Kuehn, G Quast, M Giffels, M Schnepf, A Heiss, A Petzold

Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, DE

E-mail: {max.fischer, eileen.kuehn, manuel.giffels, guenter.quast, andreas.heiss, andreas.petzold}@kit.edu, {christoph.heidecker, matthias.jochen.schnepf}@cern.ch

Abstract. With the increasing data volume of LHC Run2, user analyses are evolving towards increasing data throughput. This evolution translates to higher requirements for efficiency and scalability of the underlying analysis infrastructure. We approach this issue with a new middleware to optimise data access: a layer of coordinated caches transparently provides data locality for high-throughput analyses. We demonstrated the feasibility of this approach with a prototype used for analyses of the CMS working groups at KIT. In this paper, we present our experience both with the approach in general, and our prototype in specific.

1. Introduction

With the LHC Run2, end user analyses are increasingly challenging for both users and resource providers. On the one hand, boosted data rates and more sophisticated analyses favour and require larger data volumes to be processed. On the other hand, efficient analyses and resource provisioning require fast turnaround cycles. These requirements put the scalability of analysis infrastructures to new limits. Existing approaches to this issue, such as data locality based processing [1], are difficult to adapt to High Energy Physics (HEP) workflows [2].

For the first data taking period of Run2, the KIT CMS group deployed a prototype enabling data locality via coordinated caching. This systems augments the common HEP infrastructure of batch system and file servers connected via local network. The underlying middleware successfully solves critical issues of data locality for HEP:

(i) Caching joins local high-performance devices with large background storage.
(ii) Data selection for user workflows uses cache storage only for data optimising throughput.
(iii) Transparent integration into the batch system and operating system avoid compatibility issues with user software itself.

So far, our practical work focused on local user analyses. This was primarily driven by the scope of our demonstrative system. At its core, coordinated caching is applicable at a broader scope, however.

2. Coordinated Caching for Batch Processing

Our concept of coordinated caching aims at distributed data consumers, such as worker nodes in a batch system, and remote data providers, such as file servers. The feasibility of caching in such scenarios has been widely demonstrated [3, 4, 5, 6]. To the best of our knowledge, all existing approaches fall into one of two categories: Consumer focused caching, targeting data shared by many consumers at once, and provider focused caching, targeting data requested repeatedly.

In contrast, our concept is focused on consumers, but targets recurring requests. Specifically, we aim at optimising local data availability for few, recurring workflows processing the same data.

2.1. User Workflows as Caching Targets

Our approach is analogous to regular caches targeting applications, which use read system calls to process blocks of data. We target user workflows, which use batch jobs to process groups of files. This granularity allows us to intercept automatic data access, without interfering with manually defined tasks of workflows.

We specifically target workflows which split and assign their data statically: the portion of data processed by each job is decided at submission time. Therefore, caching must work with the predetermined data splitting.

In principle, it is possible for jobs to pick data at runtime to match cache content. However, the benefit of this micro-optimisation is low, while the cost is high: jobs must be synchronised, and perform load balancing to avoid underutilising processing resources. In effect, this needlessly replicates scheduling responsibilities already handled by the batch system.

From our experience, best results are achieved if workflows use comparable splittings on repeated invocations. That is, the chance for any two files to be accessed together should be constant. Ideally, the splitting is stable, and the same files are always accessed together. This allows the infrastructure to arrange data with regards to load balancing.

2.2. Caching for Data Locality

The goal of coordinated caching is to pre-place data on processing nodes before job execution. In essence, data locality of input is optimised, removing the need to access data over network. As shown in past work, using local storage devices solves scalability issues of network storage access [8].

However, the scale of batch processing makes caching not inherently advantageous. Performance of local storage devices is comparable to network: Throughput of each SSD drive used in our tests is only half of a 10 Gbit/s network. Since data is streamed over minutes or hours, differences in latency are negligible.

As a result, caching should be designed to *complement* existing network capacities. This has two major implications:

(i) Caching should be selective only for workflows exhausting network capacity.
(ii) Caching should provide only the fraction of data exhausting network capacity.

Both features make caching easy to adopt in batch systems: It is sufficient to provide a fraction of data for a fraction of workflows. Even small caches are beneficial, and selection may be based on simple, statistical rules.

However, the challenge for distributed processing is coherence between data and job placement. This makes independent, local caches on hosts unsuitable even on small setups. In a batch system of four nodes with independent caches, 75 % of jobs have access to 25 % or less of their cached data. Thus, it is critical to improve the hit rate by coordinating the individual caches.

2.3. Scheduling and Coordination

A distributed infrastructure adds a layer of complexity to cache access. This can be expressed by splitting the cache hit rate into two components:

(i) The *global cache hit rate* is the fraction of job data available anywhere on a cache in the infrastructure.

(ii) The *local cache hit rate* is the fraction of cached job data available on a cache accessible to the job.

The global cache hit rate is equivalent to the hit rate of regular caches. Algorithms optimising this have been studied extensively in literature. For our research, we picked an existing algorithm, based on a simple model of data access. In contrast, the local cache hit rate requires an active scheduling effort.

On the one hand, jobs must be scheduled to data. Without such an intervention, the local cache hit rate is inversely proportional to the number of processing nodes. To coordinate job scheduling to match data placement, data locality information must be published to any scheduling mechanism.

On the other hand, data must be scheduled to match job demand. If data for a single job is evenly distributed, the local cache hit rate is again penalised by cluster size. Worse, this forms a hard limit on the local cache hit rate: regardless of job scheduling, a job cannot locally access more data than what is available on a single host. The inaccessible data on other hosts also degrades overall cache performance as it reduces effective cache capacity.

As such, individual caches must be coordinated to provide data distribution coherent with job demand. This requires an explicit scheduling of data to hosts to match the demand by jobs.

3. The HTDA Middleware

We have developed and deployed a prototype to demonstrate and study coordinated caching: the *High Throughput Data Analysis* (HTDA) middleware [7, 8, 9]. It is designed to integrate the batch system scheduling jobs, and the data access protocols used by jobs.

To study different contexts for coordinated caching, the middleware is highly modular: at the moment, we are using backends to interface with the HTCondor [10] batch system and POSIX data access. This matches the analysis environment used by our local physics analysis groups. In fact, the prototype is in active use on a portion of our production batch system.

As outlined before (see Section 2.2), we see caching as a means to complement existing resources. Thus, it is a key design goal that our prototype is transparent to users. There is no difference in user workflows whether jobs are running on hosts with or without a cache, and whether data is cached at all. This means that jobs are submitted naively, with data being split to jobs without regards to cache status. Furthermore, it means that the middleware interacts only with the batch system and data access methods[1]

3.1. Prototype Architecture

Since coordinated caching is mainly a scheduling challenge (see Section 2.3), our implementation is focused on scheduling data. Loosely put, we have scaled up an operating system cache to the scale of a batch system, with new components to handle the distributed elements. The design architecture closely follows that of a batch system, which simplifies deployment alongside an existing batch system.

The middleware is made up of distinct nodes (see Figure 1), each implementing one aspect of coordinated caching:

(i) *Provider* nodes handle data provisioning. Each provider node creates and maintains local copies of data on a batch system worker node.

[1] It is technically possible for jobs to query cache content, adjust themselves at runtime, and propagate this to the middleware. Since caching only selects a fraction of data and workflows, this is a waste of effort for most workflows.

(ii) *Locator* nodes handle meta-data of job input data. For every job, they locate hosts providing data, and store the context of data access. Usually, one locator node is deployed per batch system submission node.

(iii) *Coordinator* nodes handle data scheduling. They select job input data viable for caching, and assign it to hosts where it should be provided. While not necessarily co-located, this corresponds to a batch system scheduler node.

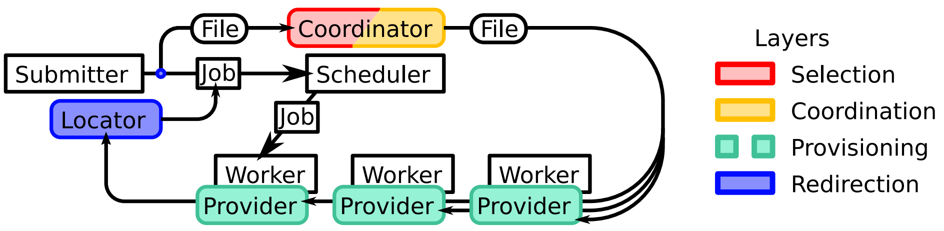

Figure 1. The HTDA middleware is made of several components, following the layout of batch systems. While the underlying batch system schedules jobs, HTDA schedules data. Components correspond both to batch system nodes and layers of regular caches.

In addition, there is a thin compatibility layer to hook into the existing infrastructure. Provider nodes interface with the data access protocols to redirect job data access. Locator nodes interface with the batch system to influence scheduling.

We have found this architecture to scale appropriately with batch system size, as expected from the similar design. This is enabled by distributing responsibilities over many nodes, and asynchronously performing individual tasks. We achieve this via distinct steps, passing on meta-data to decide which data to provide.

3.2. Meta-Data Processing

Handling meta-data in our middleware is divided into multiple steps, which form a logical sequence: Meta-data on jobs and their input data is collected. Data critical for throughput is selected based on this meta-data. An allocation of data to hosts is formulated for selected data. Finally, data is actually provisioned on hosts.

Successive steps are not synchronised - each step is regularly performed with the information currently available for it. Since job runtimes and data access are on the order of hours, the resulting asynchronicity at the order of minutes is negligible. In turn, this makes it trivial perform each step on different nodes. Furthermore, it allows distributing steps - for example, each provisioning node decides on its own which data to drop when cache devices fill up.

The overall goal is to keep coupling between nodes and tasks to a minimum. For example, centrally allocating data to specific devices on hosts is not performant even on small clusters. Instead, each node has a fixed set of unique responsibilities, which do not depend on the implementation of other components. This allows us to test different approaches for the core of our research, namely coordination.

3.3. Coordination Strategies

Similar to the rejection of synchronicity, strict coordination has proven to provide unsatisfactory performance. Attempts to enforce optimal placement of files at any time have led to unnecessary migration of data. Correlating the selection of data to cache and the allocation of selected data to hosts results in high overhead due to the complexity of each decision.

Instead, we have found heuristic approaches for each individual task to provide considerably better results. By distributing tasks over nodes, this results in a multi-agent environment with emergent behaviour. In short, having each element avoid a small set of negative conditions has proven more feasible than trying to optimise a few global, beneficial conditions at once.

To select data, we have adopted the existing LRFU algorithm [11]. This directly reflects our two major goals for selecting data: Selecting frequently used data makes it likely that data on caches is requested. Selecting recently used data avoids obsolete data. Indeed, in 90 % of cases the algorithm selects data of workflows generating the highest data throughput in our cluster.

Allocation of data to hosts is handled by an incremental group placement algorithm (see Figure 2). This takes into account the popularity of data, and the grouping of data as input for jobs. The algorithm start by randomly allocating the groups of data most often requested by jobs. Subsequently, less popular groups are either placed to groups with high overlap, or randomly if there is no overlap. While this can create duplicates on multiple hosts, we have found it to work reliably if data grouping by jobs is stable.

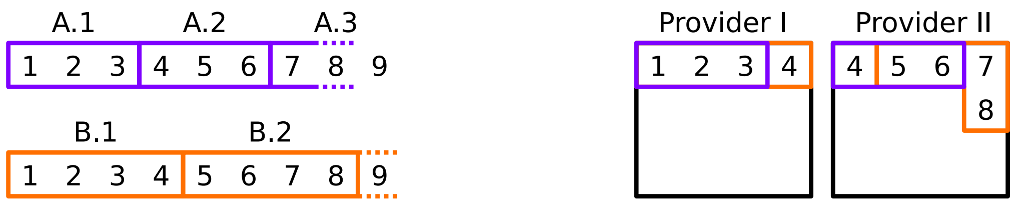

Figure 2. Files are allocated to provider nodes with the grouping observed in jobs. The example shows two jobs A and B, processing the same data with jobs A.1, A.2, and so on. Groups are first randomly allocated for the more often occuring jobs A. The similar groups of B are allocated to hosts where most files are already present.

The scheduling of jobs to their data uses a best-effort approach. For each job, we rank worker nodes by the fraction of data available locally. This prefers optimal job and data placement, but also handles misalignment.

Furthermore, job scheduling has to strike a middle ground between efficiency and latency. After job submission, there is an optimal scheduling window in which only hosts providing data can be selected. After this window, the job may run on any compatible host to avoid high latency. This mechanism is skipped if there are no hosts providing data for a job. We use a window corresponding to half the expected job wall time, which is sufficient for optimally scheduled jobs to complete and release their slot. On average, jobs access about 75 % of the data stored on caches.

It is important to reiterate that we do not aim for a perfect overall cache hit rate (see Section 2.2). Our algorithm accurately selects repeating, high throughput workflows, but ignores non-recurring workflows entirely. While scheduling of jobs increases hit rates drastically, we purposely trade ideal hit rates and efficiency for improved latency. In contrast to classic caching, our goal is not to improve all data accesses, but to improve capacities used during peak load. This makes our approach an enabler for data locality complementing existing analysis infrastructure.

3.4. Deployment via Docker

To implement data locality transparently for users requires tight integration with the underlying infrastructure. For our local physics analysis users, we need to provide data for POSIX access. In specific, this means we must hook into the virtual file system layer of the operating system. To implement this, our demonstration systems use SSDs as local cache, on which our middleware copies a fraction of data from file servers mounted via NFS. We use an AUFS [12] overlay file system to provide a single, unified view of both cache and storage to users.

This approach is not suitable in any standard processing setup currently used for HEP. The 2.X linux kernel used by RHEL6 and RHEL7 operating systems is unsuitable to deploy modern components such as AUFS. While older versions exist, these are not stable enough for high throughput data processing. Our tests consistently result in deadlocks in the kernel after several minutes of peak load.

To address this issue, we aggressively separate the middleware and job runtime via the use of containers. The host operating system is a recent CentOS 7, but runs the most up to date 4.X kernel. This allows the deployment of our middleware with dependencies suitable for high performance processing. Jobs run inside docker containers, which provide a runtime environment comparable to Scientific Linux 6.

This separation enables the use of current technologies, while ensuring compatibility with existing workflows. It underlines our approach of providing coordinated caching as part of the infrastructure. The entire setup is entirely transparent for users and their jobs - the only noticeable effect is an increased in performance.

4. Further Applicability

The choice of a local batch system environment is largely motivated by the high level of control and ease of modifications. Similarly, some design choices such as full transparency are the result of scope and user group. The underlying concept of coordinated caching is applicable beyond this, however.

4.1. Cross Site Caching for Independent Processing Resources

To fully utilise processing resources, it can be advantageous to read data from external resources. This allows the use of processing resources without any data available on a local storage element. In specific, this is the default situation for opportunistic resources.

Related work [3] has shown the viability of using the XRootD [13] protocol to enhance data locality in local clusters. However, XRootD is also commonly used for remote data access, and is a key component in data federations. We have exploited both aspects in a demonstrative setup of independent processing nodes.

XRootD allows the use of *forwarding proxy servers* as caches agnostic to remote storage. In this setup, the proxy server is actively requested to serve a file from remote storage. This allows the proxy server use a local cache. Since the request explicitly includes the remote storage, the proxy does not require knowledge of all possible remote data sources.

Our demonstration setup has any provider node launch a local forwarding proxy server. Jobs perform their data requests via this local proxy, which serves data provided by us. This allows to locally provide files from arbitrary remote data providers. This setup can be deployed without root privileges, making it suitable for dynamically acquired pilot resources.

The advantage of this approach is that it has no dependency on any dedicated storage element, nor any other supporting infrastructure. The presence of local scratch space is sufficient, since our cache expects data to be volatile anyway. Coordination of caches allows the coherent usage of processing resources even if storage space is not shared.

4.2. Locality Aware Jobs

The model of locality unaware jobs is a result of our prototype serving complementary to regular resources. Efficiency of this setup relies on network being performant enough to absorb occasional cache misses. This may not be the case in all situations, and in fact may be the reason to desire data locality in the first place.

Our approach works well if locality awareness is used prior to submission. It is easy for submission tools to query data placement, and split data to jobs accordingly. For users, this has the advantage that jobs do not need to be synchronised at runtime. At the same time, the batch system and caching middleware remain their autonomy on scheduling decisions.

Our prototype is technically capable of handling jobs that adjust their input based on locality information at runtime. However, this makes several features obsolete or counterproductive, breaking assumptions on exclusive responsibilities. Most prominently, scheduling of jobs to data conflicts with internal synchronisation of jobs. In principle, similar algorithms apply, but responsibilities between infrastructure and workflows should be strictly divided. We strongly recommend that services focus on jobs which are either aware or unaware of locality at runtime.

5. Summary and Conclusions

To improve performance for data intensive workflows, modern infrastructure strives to optimise data locality. We have developed a concept to transparently enable this in existing HEP processing environments: Data is provided locally on processing nodes by caches, which are coordinated to stay coherent with distributed workflows. While we have tested our approach in a limited environment, the approach is applicable in general.

The key point of our approach is the extension of scheduling of data to match jobs. This is a requirement for adequate cache hit rates in distributed environments. Our prototype implementation demonstrates that even simple, heuristic scheduling boosts hit rates sufficiently.

In our experience, heuristic approaches are not just sufficient but superior at the scale of even simple batch systems. Asynchronous, loosely coupled steps for data provisioning are required to avoid scalability issues. Simple rules have been sufficient to create a provisioning that is robust against cache trashing and similar effects.

While our work so far is largely focused on local processing resources, we have experimented with other use cases. We can unconditionally recommend exposing location data to allow adjustments to data splitting before job submission. Furthermore, we consider caching for cross-site data access easily viable. Demonstrative setups show that this enables data processing with little if any infrastructure, as is the case for opportunistic computing.

References

[1] The Hadoop project homepage **URL** http://hadoop.apache.org/
[2] Lehrack S, Duckeck G and Ebke J 2014 Evaluation of Apache Hadoop for parallel data analysis with ROOT *J. Phys.: Conf. Ser.* **513** 032054
[3] Yang W, Hanushevsky A, Mount R and the Atlas Collaboration 2014 Running a typical ROOT HEP analysis on Hadoop MapReduce *J. Phys.: Conf. Ser.* **513** 042035
[4] Weitzel D, Bockelman B and Swanson D 2015 Distributed caching using the HTCondor CacheD *Proc. for Conf. Parallel and Distrib. Process. Techn. and Appl.*
[5] Blome, J and Fuhrmann T 2010 A fully decentralized file system cache for the CernVM-FS *2010 Proc. of 19th Int. Conf. Comput. Commun. and Netw. (ICCCN)* 1-6
[6] Paul S and Fei Z 2001 Distributed caching with centralized control *Comput. Commun.* **24** 256-68
[7] The HTDA project repository **URL** https://bitbucket.org/kitcmscomputing/hpda
[8] Fischer M, Giffels M, Jung C, Kuehn E, and Quast G 2015 Tier 3 batch system data locality via managed caches *J. Phys.: Conf. Ser.* **608** 012018
[9] Fischer M, Metzlaff C, Giffels M, Jung C, Kuehn E, Hauth T and Quast G 2015 High performance data analysis via coordinated caches *CJ. Phys.: Conf. Ser.* **664** 092008
[10] Thain D, Tannenbaum T and Livny M 2005 Distributed computing in practice: the Condor experience *Concurrency Computat.: Pract. Exper.* **17** 32356 (doi: 10.1002/cpe.938)
[11] Lee D, Choi J, Kim J, Noh, S.H., Min S L, Cho Y and Kim C S 2001 LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies *IEEE Transactions on Computers 12* **50** 1352-61
[12] The AUFS project homepage **URL** http://aufs.sourceforge.net
[13] The XRootD project homepage **URL** http://xrootd.org