Distributed data processing for High Energy Physics

Dzmitry Makatun

4th year of PGS, email: dzmitry.makatun@fjfi.cvut.cz
Department of Mathematics
Faculty of Nuclear Sciences and Physical Engineering, CTU in Prague

advisors:

Jérôme Lauret, STAR, Brookhaven National Laboratory, USA

Hana Rudová, Faculty of Informatics, Masaryk University, Czech Republic

Michal Šumbera, Nuclear Physics Institute, Academy of Sciences, Czech Republic

Contents

1 Introduction

The STAR experiment at the Relativistic Heavy Ion Collider (RHIC) studies a primordial form of matter that existed in the universe shortly after the Big Bang. Collisions of heavy ions occur millions of times per second inside the detector, producing tens of petabytes of raw data each year. All the raw detector data has to be processed in order to reconstruct physical events which are further analyzed by scientists. This process is called *data production*.

Like any other modern experiment in High Energy and Nuclear Physics (HENP), STAR intends to rely on distributed data processing, making use of several remote computational sites (for some experiments this number can scale up to several hundreds) (see Figure 1).

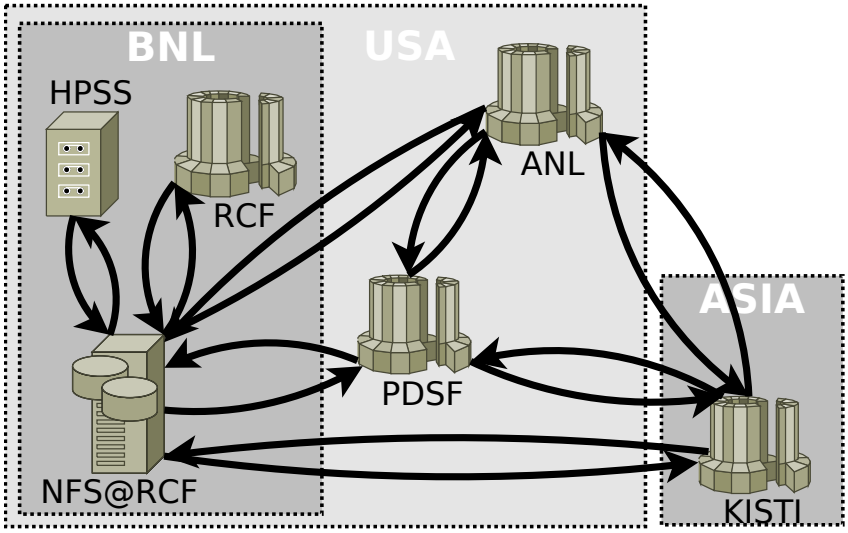

Figure 1: Data production Grid of the experiment STAR. Raw data is recovered from central HPSS to NFS storage at RCF and then it is distributed between local and remote sites. Arrows represent network links.

When running data intensive applications on distributed computational resources long I/O overheads may be observed as access to remotely stored data is performed. Latency and bandwidth can become the major limiting factors for the overall computation performance and can reduce the *CPU time* to *wall time* ratio due to excessive I/O wait [1, 2]. Widely used data management systems in HENP community (Xrootd [3], DPM [4]) are focused on providing heterogeneous access to distributed storage and do not consider data pre-placement with respect to available CPUs, job durations or network performance. At the same time job scheduling systems (PBS [5], Torque [6], Condor [7]) do not reason about transfer overheads when accessing data at distributed storage. For this reason, an optimization of data transferring and distribution across multiple sites is often done manually, using a custom setup for each particular infrastructure [8].

Previous collaborative work between BNL (Brookhaven National Laboratory) and NPI/ASCR (Nuclear Physics Institute, Academy of Sciences of the Czech Republic) showed that the global planning of data transfers within the Grid can outperform widely

used heuristics such as Peer-to-Peer and Fastest link (used in Xrootd)[9]. This concept was later complemented with usage of caching [10]. The comparison and selection of the most appropriate caching policies for the data access pattern in HEP is described in Section 4.

Those results became the ground for continuation of research and extension of global planning to the entire data processing workflow, i.e., scheduling of CPU allocation, data transferring and placement at storage [11, 12]. In Section 3 we propose a constraint programming planner that schedules computational jobs and data transfers in a distributed environment in order to reduce the overall completion time. Since such global scheduling is computationally demanding it should be divided into several stages in order to improve scheduler performance and scalability. A planing of resource load can be accomplished in the first stage before scheduling particular file transfers and jobs. In Section 2 we address the problem of data production planning, answering the question how the load should be distributed over resources and how the data should be transferred given the network structure, bandwidth, storage and CPU slots available. This will allow each computational site to load the CPUs with jobs while not exceeding disk and network capacities.

1.1 Workload types

Due to a data level of parallelism a typical workflow of HENP computation consists of independent jobs requiring one CPU, using one input and producing one output file each. This workload can be further divided into two classes of jobs: data production and user analysis.

Data production jobs are processing raw detector data into reconstructed physical events. This jobs are usually submitted centrally. Input files for these jobs are initially placed at the central data storage of an experiment (e.g. tape storage at Tier-0 site). Each raw file has to be processed once. An output file is of comparable size with the input file. After processing, the output file has to be copied back to the central storage. The typical task of a data production campaign would be to process a given dataset (i.e. several Petabytes of similar files) as soon as possible (e.g. next several month after data taking). Under such scenario the particular order of jobs is unimportant, while the makespan becomes the main value for an optimization. Since all the jobs processing input files form the same dataset are similar, parameters of upcoming jobs can be estimated knowing the statistics of completed jobs.

User analysis consists of jobs submitted by many different researches. These jobs are processing the reconstructed data which are the result of data production. Input files can have several copies within Grid. In this case the same input file can be processed several times by the same or different users. Output size of user analysis is usually negligible compared to the size of input. Each researcher uses its own code for analysis, for that reason the job parameters are far less predictable then for the data production. Moreover, when users submit their jobs to the batch system they often overestimate the runtime to prevent their jobs being killed due to overrun. Another contrast to data production is that due to a multi-user workload such characteristic as *fairness to users* becomes important. There is no strict definition of fairness in scheduling, but the general

principle is that all the users should receive a fair share of resources. There also exist other commonly accepted metrics to measure the quality of a schedule in a multi-user environment: response time, wait time, slowdown, fragmentation and *etc.* [13].

Sections 2 and 3 are focused on scheduling for data production, while Section 4 compares caching strategies for a user access pattern.

1.2 Use cases

Long I/O overheads when accessing data from remote site can significantly reduce the application's CPUtime/WallTime ratio [2, 1]. For this reason, when setting up a data production at remote sites one has to consider the network throughput, available storage and CPU slots. When there are few remote sites involved in the data processing, the load can be tuned manually and simple heuristic may work, but, as the number of sites grows and the environment is constantly changing (site outage, fluctuations of network throughput and CPU availability), an automated planning of workflows becomes necessary.

As an intuitive example of optimization let us consider a situation when a given dataset can be either processed locally, or can be sent to a remote site. Depending on transfer overhead it may appear to be optimal to wait for free CPU slots at the local site and process all the data there, or send a smaller fraction of the dataset for remote processing. Commonly used heuristics such as "*Pull a job when a CPU slot is free*" will not provide an optimization with respect to an overall processing makespan.

Another example arises from a workflow optimization which was done for inclusion of the ANL (Argonne National Laboratory) computational facility into the Cloud based data production of the STAR experiment [8]. In this case, and due to the lack of local storage at the site for buffering, the throughput of a needed direct on-demand network connection between BNL (New York) and ANL (Illinois) was not sufficient to saturate all the available CPUs at the remote site. An optimization was achieved by feeding CPUs at ANL from two sources: directly from BNL and through an intermediate site NERSC (National Energy Research Scientific Computing Center, Oakland, California) having large local caching and with better connectivity to ANL (see Figure 1). This example illustrates an efficient use of indirect data transfers which cannot be guessed by simple heuristics. A general illustration of distributed resources used for data production and their interconnection is given at Figure 2.

Scheduling of computational jobs submitted by users (user analysis) has even more degrees of possible optimization: selection between multiple data sources, grouping of jobs that use the same input files. This case becomes even more complex due to a poor predictability of the user analysis jobs. However, the main question for optimization remains the same as for the examples above: How to distribute a given set of tasks over the available set of resources in order to complete all the tasks within minimal time?

1.3 Related work

Most of the widely used job scheduling policies such as First Come First Served (FCFS), conservative backfilling (CONS), aggressive backfilling (EASY), selective backfilling and

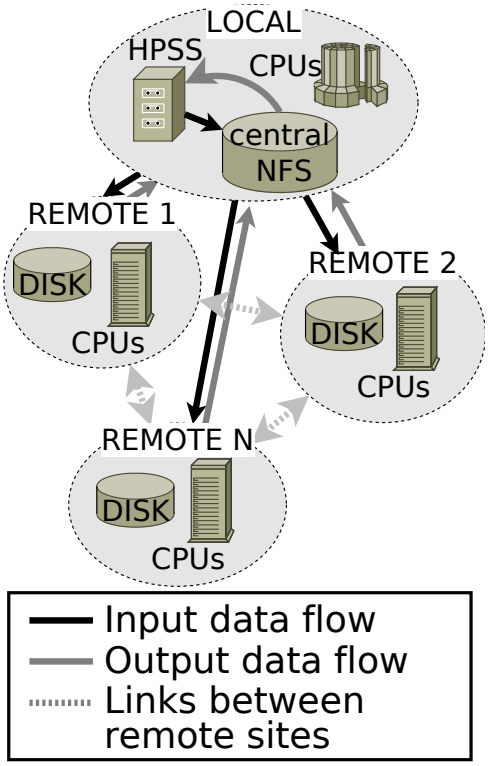

Figure 2: Schema of data production in the Cloud.

etc. [14] are focused on maximizing CPU throughput and fairness to users. In work [15] the backfilling is performed with the help of Tabu search, which helps to find a schedule closer to the global optimum. In work [13] authors proposed a multi-resource aware user prioritization mechanism, where fairness to users is improved by taking into account the usage of several resources (such as CPU, RAM, HDD, GPU) and heterogeneity of the system. The above strategies are dedicated to manage a single computational cluster where data transfer overhead is not a significant factor. Also, for the data production workflow such metrics as fairness to users is irrelevant.

In [16] authors have modified existing heuristics to schedule parameter sweep applications with file I/O requirements (e.g. Monte Carlo simulations), and studied an impact of inaccurate performance prediction on scheduling. Authors consider job scheduling on heterogeneous resources (Grid) taking data transfer overhead for each job into account. While the input transfer overhead was estimated knowing an end-to-end connection speed, the file transfers itself were not scheduled at network links, neither actual network topology was taken into account. In case of data production in HENP, uncoordinated data transfers may oversaturate the network capacity which leads to overall degraded performance. For this reason in our research we consider network load planning.

Optimization of data intensive applications in Grid was studied in [17]. In this work an optimization was achieved by replication of frequently used files to more sites while the jobs were executed where their input data is located. However, this is not the case for data production, when each file has to be processed once.

Explicit model distributing jobs over a Grid with respect to the network bandwidth

was proposed in [18]. The network structure of the Grid was modeled as a tree and all the files were assumed to be of the same size and processing time. In our study we do not limit the network topology to trees, and assume fluctuations of job parameters.

2 Resource load planning

Due to a data level of parallelism a typical workflow of HENP computation consists of independent jobs using one CPU, one input and one output file. For the following model we assume there is a local scheduler running at each site, which picks a new input file to process from the local storage of that site each time when a CPU becomes free. Input data must be transferred from the central storage to each site in such a manner that at the every moment of time there is enough input files at each site to keep all the available CPUs busy while not exceeding the local storage and network throughput. Another task is to transfer the output files back to central storage, cleaning each local storage for the new input.

Let us consider a scheduling time interval ΔT. We assume that at the starting moment all the CPUs in the Grid are busy, and there is some amount of input data already placed at each site. We need to transfer the next portion of data to each site during time interval ΔT in order to avoid draining of the local queue by the end of this interval.

The computational Grid is represented by a directed weighted graph where vertexes $c_i \in C$ are computational nodes and edges $l_j \in L$ are network links. Weight of each link b_j is the amount of data that can be transferred over the link per unit of time (i.e. bandwidth). One of the nodes, c_0, is the central storage where all the input files for the further processing are initially placed. All the output files have to be transferred back to c_0 from the computational nodes. We will give two separate problem formulations: for an input and output transfer planning.

In order to formulate a network flow maximization problem [19] for input/output file transferring we have to define a capacitated $\{s, t\}$ network, which is a set of vertexes V including a source s and a sink t; and a set of edges $e \in E$ with their capacities $cap(e)$. A solution that assigns a nonnegative integer number $f(e)$ to each edge $e \in E$ can be found in polynomial time with known algorithms.

2.1 Input flow planning

In order to transform a given graph of a Grid into a capacitated $\{s, t\}$ network for an input transfer problem we add two dummy vertexes: a source s and a sink t. Next we add dummy edges $d_i \in D$ from each computational node i to the sink, and a dummy edge q_0 from the source s to the central storage c_0. For example, the transformation of the Grid from Figure 1 is illustrated at Figure 3 (HPSS node was excluded from the consideration).

These dummy edges allow us to introduce constraints on the storage capacity of the nodes. The set of vertexes V consists of computational nodes C and dummy vertexes: $V = C \cup \{s, t\}$. The final set of edges consists of real network links L, dummy edges D from computational nodes to the sink and from the source to the central storage q_0: $E = L \cup D \cup \{q_0\}$. Capacity of each edge defines the maximal amount of data that can be transferred over an edge within time interval ΔT:

$$cap(e) = \begin{cases} b_j \cdot \Delta T & \text{if } e = l_j \in L \\ w_i & \text{if } e = d_i \in D \\ k_0 & \text{if } e = q_0 \end{cases} \quad (1)$$

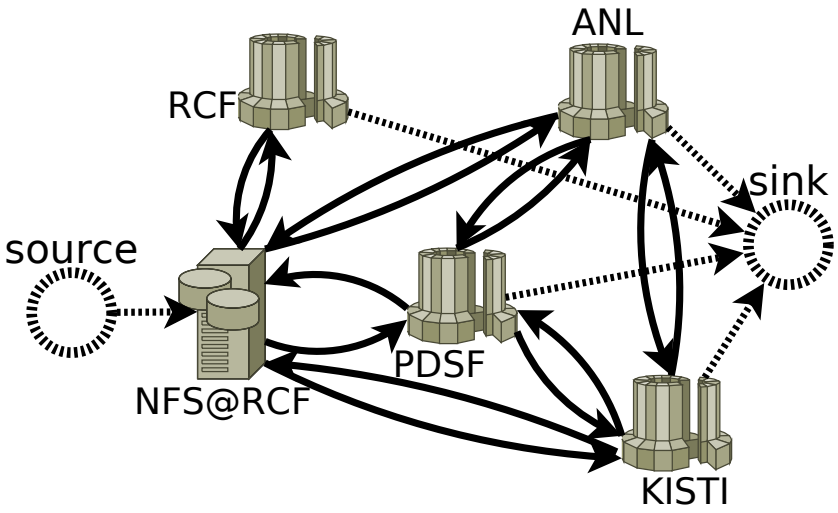

Figure 3: Input transfer planning: transformation of Data production Grid into capacitated $\{s,t\}$ network for a flow maximization problem. Dummy vertexes (a source and a sink) are added. Each computational site is connected to the sink by a dummy edge. The source is connected to the NFS at RCF via dummy edge. The dummy edges are used in order to express constraints on network flow imposed by storage capacity and number of CPUs at each site. Central HPSS is removed from consideration.

where w_i is the maximal amount of data that can be transferred to the node i without exceeding its storage capacity $Disk_i$ and k_0 is the total size of available input files at c_0. A general case of a resulting network is represented at Figure 4. We denote the solution for the input transfer problem as $f^{in}(e)$.

2.2 Output flow planning

For transfer of output files we use a similar transformation, but swap the source s and the sink t, change the direction of dummy edges and redefine capacities of dummy edges (see Figure 5). In this case the capacity $\overline{k}_0$ of the dummy edge $\overline{q}_0$ leading from the central storage c_0 to the sink s is equal to the amount of data which can be transferred to c_0 within time interval ΔT (it is limited by the available disc space at the central storage). The capacity $\overline{w}_i$ of dummy edges $\overline{d}_i$ leading from the source t to computational nodes c_i is equal to the maximum amount of output data which can be transferred from the node c_i.

$$cap(e) = \begin{cases} b_j \cdot \Delta T & \text{if } e = l_j \in L \\ \overline{w}_i & \text{if } e = \overline{d}_i \in \overline{D} \\ \overline{k}_0 & \text{if } e = \overline{q}_0 \end{cases} \quad (2)$$

We denote the solution for the output transfer problem as $f^{out}(e)$.

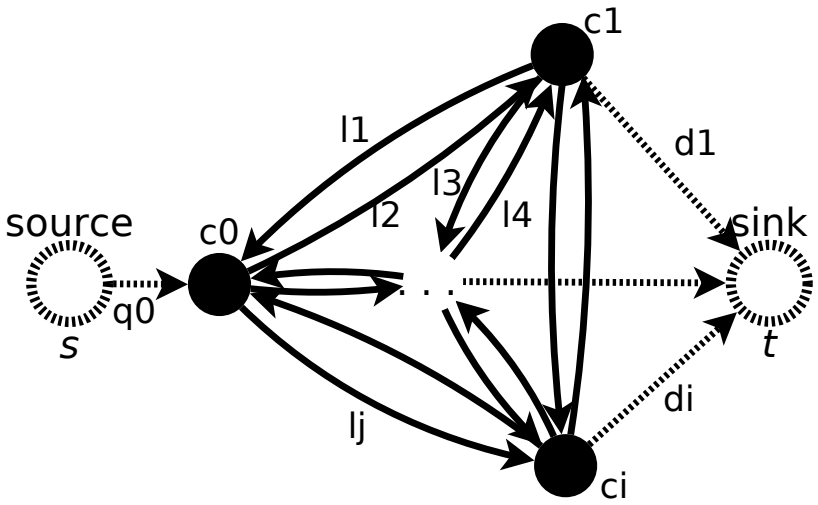

Figure 4: Data production Grid represented as an capacitated $\{s, t\}$ network for input transfer planning (general case). c_0 is a central NFS storage (Tier-0), c_i are computational nodes (where $i > 0$), l_j are network links, d_i are dummy edges from computational nodes to the sink t, q_0 is a dummy edge leading from the source s to the c_0.

2.3 Calculating capacity of dummy edges

Let us consider data production jobs which perform the same type of processing on the same type of files. Duration p_j of job j processing input file of size $InSize_j$ at node i is

$$p_j = \alpha_i \cdot InSize_j \tag{3}$$

Figure 6 shows a distribution of parameter α for 7000 of data production jobs executed at the same computational site (KISTI). It can be observed that the narrow peaks correspond to jobs using input files of the different type (`"st_physics"`, `"st_physics"`, `"st_physics_adc"`, `"st_jet"`, etc.). For this reason, parameter α can be considered constant for a given type of data processing at a given site.

The ratio of size of input $InSize_j$ and output $OutSize_j$ files of each job j is considered to be constant for the same type of data processing:

$$OutSize_j = \beta \cdot InSize_j \tag{4}$$

Figure 7 shows a distribution of parameter β for 7000 of data production jobs executed at the same computational site (KISTI). Similarly to written above, for a given type of data processing this parameter can be considered constant.

During time interval ΔT (which should be long enough) a node i with number of CPUs $NCPU_i$ will process $\frac{1}{\alpha_i} \cdot NCPU_i \cdot \Delta T$ of input data and will produce $\frac{\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T$ of output data.

Let us consider a storage at a processing node i during time interval ΔT. $Disk_i$ is available disk space at node i. The I_i^{in} is the initial size of input data stored at the local storage; I_i^{out} is the initial size of output data at the storage; New_i^{in} is the amount of input data that will be transferred to i during ΔT; Del_i^{in} is the amount of input data that will

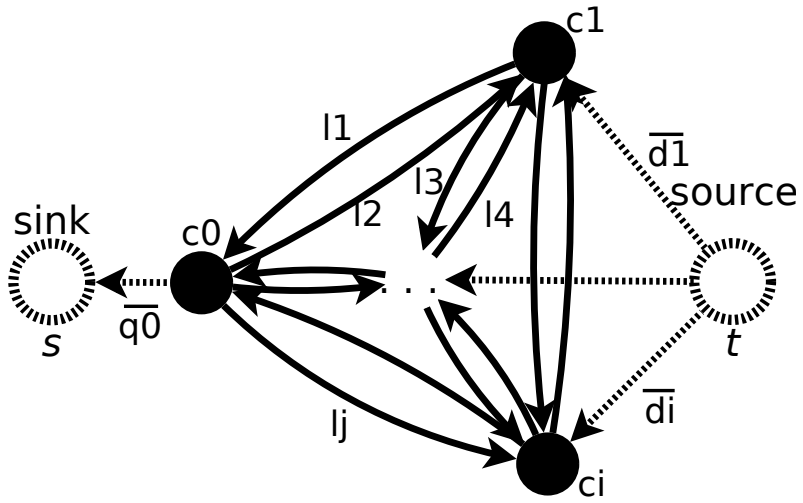

Figure 5: Data production Grid represented as an capacitated $\{t, s\}$ network for output transfer planning (general case). c_0 is a central NFS storage (Tier-0), c_i are computational nodes (where $i > 0$), l_j are network links, $\overline{d}_i$ are dummy edges from the source t to computational nodes, $\overline{q}_0$ is a dummy edge leading from c_0 to the sink s.

be deleted from the storage, because the jobs using these input data will be completed; Del_i^{out} is the amount of output data which will be deleted from the storage, because it will be transferred out of the node; New_i^{out} is the size of new output files to be created during ΔT; Min_i^{in} is minimal amount of input data (this includes input files of running jobs and files in the local queue to ensure stable CPU saturation), Min_i^{out} - total size of output files which can not be transferred because the jobs which produce them are not finished (output files of running jobs).

In the end of ΔT there should be enough input data to keep CPUs busy:

$$I_i^{in} + New_i^{in} - Del_i^{in} \geq Min_i^{in} \geq 0 \tag{5}$$

In the end of ΔT the storage capacity should not be exceeded:

$$0 \leq I_i^{in} + I_i^{out} + New_i^{in} + New_i^{out} - Del_i^{in} - Del_i^{out} \leq Disk_i \tag{6}$$

The balance of the output data at the end of ΔT is:

$$I_i^{out} + New_i^{out} - Del_i^{out} \geq Min_i^{out} \geq 0 \tag{7}$$

If the scheduling interval ΔT is long enough, then we can use the following approximation:

$$Del_i^{in} \approx \frac{1}{\alpha_i} \cdot NCPU_i \cdot \Delta T \tag{8}$$

and

$$New_i^{out} \approx \frac{\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T \tag{9}$$

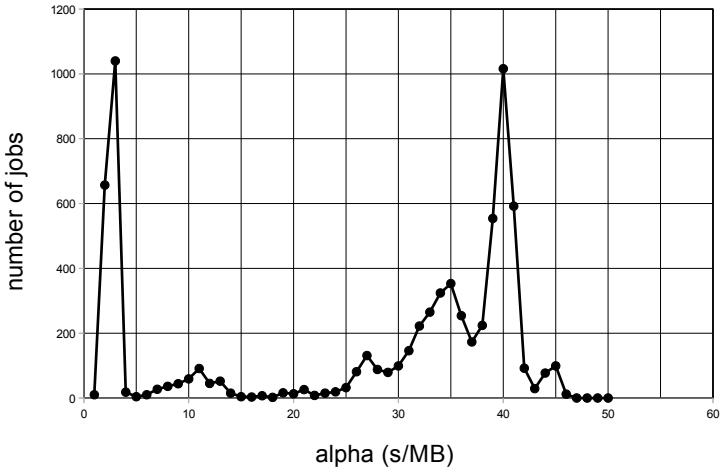

Figure 6: Distribution of parameter α (job duration divided by input file size) calculated for sample of 7000 jobs executed at KISTI during June - September 2014. Narrow peaks correspond to jobs using input files of different types (`"st_physics"` at 40 s/MB, `"st_physics_adc"` at 3 s/MB, `"st_jet"` at 35 s/MB, etc.). For this reason, parameter α can be considered constant for a given type of data processing.

Then, combining Equations 5, 6, 8 and 9 we can estimate amount of new input data New_i^{in} which can be transferred to a node.

$$Min_i^{in} + \frac{1}{\alpha_i} \cdot NCPU_i \cdot \Delta T - I_i^{in} \leq New_i^{in} \leq Disk_i - I_i^{in} - I_i^{out} + \frac{1-\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T + Del_i^{out} \tag{10}$$

Similarly, using Equations 7 and 9 the amount of output data which can be deleted from a node is

$$Del_i^{out} \leq I_i^{out} + \frac{\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T - Min_i^{out} \tag{11}$$

Since w_i was defined in Subsection 2.1 as maximal amount of input data that can be transferred to the computing node c_i, we can now define it using Equation 10:

$$w_i = Disk_i - I_i^{in} - I_i^{out} + \frac{1-\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T + Del_i^{out} \tag{12}$$

where Del_i^{out} is equal to the amount of data which will be transferred from node c_i, i.e. the solution to the output transfer problem $f^{out}(\overline{d}_i)$ (see Subsection 2.2). The other values used in Equation 12 can be obtained from monitoring data.

Similarly, $\overline{w}_i$ was defined in Subsection 2.2 as maximal amount of output data which can be transferred from node c_i. From Equation 11 we obtain:

$$\overline{w}_i = I_i^{out} + \frac{\beta}{\alpha_i} \cdot NCPU_i \cdot \Delta T - Min_i^{out} \tag{13}$$

where ΔT, Min_i^{out} are parameters of the scheduler, and the rest of the values can be extracted from monitoring system.

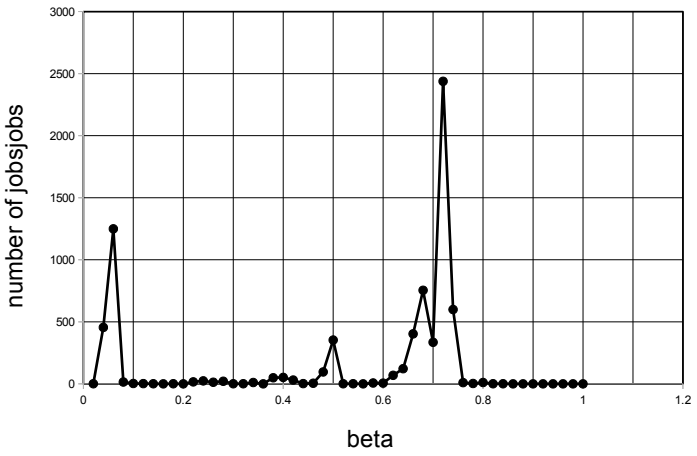

Figure 7: Distribution of parameter β (ratio of output file size over input file size) calculated for sample of 7000 jobs executed at KISTI during June - September 2014. Narrow peaks correspond to jobs using input files of different types (`"st_physics"` at 0.72, `"st_physics_adc"` at 0.04, `"st_fms"` at 0.5 , etc.). For this reason, parameter β can be considered constant for a given type of data processing.

2.4 Disjoint input and output transfer

In this subsection we will prove that maximum flow problems for input and output transfers can be solved independently under assumptions: (a) all the real network links in considered Grid are full-duplex, i.e. network throughput between two nodes is the same in both directions (b) in stable regime size of output transferred from each node is proportional to size of input transferred to that node at each scheduling interval, i.e. $f^{out}(\overline{d}_i) = \beta \cdot f^{in}(d_i)$, where $\beta \leq 1$.

Let us consider two distinct computational nodes c_1 and c_2 connected by two opposite directed links $l_1 = (c_1, c_2)$ and $l_2 = (c_2, c_1)$ with equal capacities $cap(l_1) = cap(l_2)$. If a solution of the input transfer problem assigns flows to this links such that $f^{in}(l_1) \geq f^{in}(l_2) > 0$ then we can substitute such solution $f^{in}(e)$ with a new one $\widehat{f}^{in}(e)$ where $\widehat{f}^{in}(l_1) = f^{in}(l_1) - f^{in}(l_2)$, $\widehat{f}^{in}(l_2) = 0$ and flows over the rest of the links are unchanged. The same is true for output transfer. This proves that in an optimal solution the same type of files (input or output) are transferred between any two nodes in one direction only, i.e. over one of directed links only.

If we have an optimal solution for the input flow maximization problem $f^{in}(e)$ we can produce from it an optimal solution for the output flow maximization problem $f^{out}(e)$ such that for any opposite pare of links $l_1 = (c_1, c_2)$ and $l_2 = (c_2, c_1)$ the output flow is $f^{out}(l_1) = \beta \cdot f^{in}(l_2)$ and since $\beta < 1$ the capacity of links is not exceeded $f^{out}(l_1) = \beta \cdot f^{in}(l_2) \leq f^{in}(l_2) \leq cap(l_2) = cap(l_1)$. Due to symmetry of the two problems, this solution $f^{out}(e)$ is also the maximum flow for the output transfer problem. Combining this with what was proven in previous paragraph, if $f^{in}(l_1) > 0$ then $f^{in}(l_2) = 0$ and thus $f^{out}(l_1) = 0$. This means that in the optimal solution input and output files are never

transferred over the same link. And thus, maximum flow problems for input and output transfers can be solved independently.

2.5 Solving Procedure

The maximum flow problems for input and output transfers can be solved independently under assumptions: (a) all the real network links in the considered Grid are full-duplex, i.e., a network throughput between two nodes is the same in both directions (b) in a steady state the size of the output transferred from each node is proportional to the size of the input transferred to that node in each scheduling interval, i.e., $f^{out}(\overline{d}_i) = \beta \cdot f^{in}(d_i)$, where $\beta \leq 1$.

Since in real environment the assumption (b) will not strongly hold due to resource performance fluctuations we propose the following approach to solve the problem:

1. Calculate values for $\overline{w}_i$ using Eqn. 13.
2. Solve the problem for output data flows to obtain $f^{out}(e)$.
3. Using Eqn. 12 and $Del_i^{out} = f^{out}(\overline{d}_i)$ calculate w_i.
4. For real links $l \in L$ reduce the capacity by the amount which is used by output transfers: $cap(l_j) = b_j \cdot \Delta T - f^{out}(l_j)$.
5. Solve the problem for input transfers with w_i and $cap(l_j)$ defined in previous steps. Find input data flows $f^{in}(e)$.

To conclude, this procedure is expected to compute feasible data transfers such that CPUs in Grid are busy with computational jobs while not exceeding local disk capacities.

3 Constraint programming approach

Problems of scheduling, planning and optimization are being commonly solved with the help of Constraint Programming (CP) [20]. It is a form of declarative programming which is widely used in scheduling, logistics, network planning, vehicle routing, production optimization *etc...* In the this section we will introduce our Constraint Satisfaction Problem (CSP) formulation for a data production at multiple sites and provide a simulation-based evaluation of the proposed model. In contrast to the model in previous section which plans global data flow this model describes per job/file/transfer scheduling.

We will introduce only the core concepts of our CSP formulation and search algorithms, omitting detailed mathematical expressions. The following input parameters are necessary to define our CSP.

Computational Grid (see Figure 2) is described by directed weighted graph where nodes are computational sites c with a given number of CPUs cpu_c and storage space $disk_c$; edges are network links l with weight $slowdown_l$ which is the time required to transfer a unit of data ($slowdown_l = \frac{1}{throughput_l}$). A dedicated storage facility, such as HPSS [21], can also be modeled as a node with $cpu_c = 0$.

Set of jobs. Each job j has a $duration_j$, it needs one input file of $inputSize_j$, produces one output file of $outputSize_j$, input file is placed at $inputSourceNodes_j$ and output file must be transferred to one of $outputDestinationNodes_j$.

Our goal is to create a schedule of jobs at computational sites, transfers over links and a placement of files at storages for a given computational Grid and a set of jobs. In order to solve this problem the variables of our model define the *resource selection* and *timing* of each task:

Resource selection variables define a node $ProcessingNode_j$ where the job j will be executed and a transfer path for each file f (either input or output of a job). The transfer path is described by a set of boolean variables X_{fl} where $true$ means that a file f will be transferred over a link l and $false$ means the opposite.

Time variables are: Js_j is a start time of a job j, Ts_{fl} is a start time of a transfer of a file f over a link l, Fs_{fc} is a start time of a placement of a file f at a node c, $Fdur_{fc}$ is a duration of a placement of a file f at a node c.

In our model we assume that a network link can be modeled as an unary resource with no loss of generality. The measurements in [9] have shown, that a sequential transfer of a set of files does not require more time than a parallel transfer of the same set of files over the same link.

3.1 Search overview

We use an incomplete search which can provide a suboptimal solution of required quality within a given time limit because the final goal is to create a planner that can process requests online. For a better search performance the overall problem is divided into two subproblems and the search is performed in two stages:

1. Planning Stage: instantiate a part of variables in order to assign resources for each task.
   (a) Assign jobs to computational nodes.
   (b) Select transfer paths for input and output files.
   (c) Estimate a makespan for a given resource assignment $estMakespan$.
   (d) Find a solution for the subproblem with a minimal estimated makespan.
2. Scheduling stage: define a start time for each operation.
   (a) Define the order of operations.
   (b) Put cumulative constraints on resources in order to avoid their oversaturation at any moment of time.
   (c) Find a solution with a minimal *makespan* which is the end time of the last task.

3.2 Constraints at the planning stage

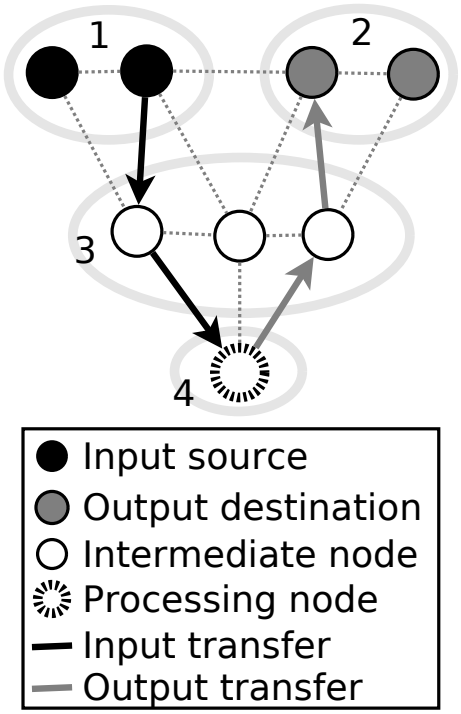

Figure 8: An example of a transfer path. Illustration for constraints 1-4 in section 3.2.

At the planning stage we have to assign a transfer path for an input and an output file of each job which can be defined by the following constraints (see Figure 8):

1. An input file has to be transferred from one of its sources over exactly one link.

2. An output file has to be transferred to one of its destinations over exactly one link.

3. An intermediate node (neither source, destination nor selected for the job execution) either has exactly one incoming and outgoing transfer or is not on a transfer path: $\exists$ incoming transfer $\Leftrightarrow \exists$ outgoing transfer.

4. There must exist exactly one incoming transfer of an input file and exactly one outgoing transfer of an output file at the node which was selected for the job execution.

5. A file can be transferred from/to each node at most once.

In addition, we use constraints for loop elimination similarly as it is described in [22].

3.3 Constraints at the scheduling stage

At the scheduling stage the problem is to assign a start time for each task. The following constraints on order of tasks are implemented:

- An outgoing transfer of a file from a node can start only after an incoming transfer to that node is finished. The first transfer of an input file from its source and the first transfer of an output file from the processing node are exceptions from this constraint.
- A job can start only after the input file is transferred to the selected processing node.
- An output file can be transferred only after the job is finished.
- A reservation of space for a file at a node is made when a transfer to that node starts.
- A file can be deleted from the start node of a link after the transfer is finished.
- A reservation of space for an output file is made at the processing node when the job starts.
- An input file can be deleted from a processing node after the job is finished.

Cumulative constraints are widely used in Constraint Programming for description of resource usage by tasks. Each cumulative constraint requires that a set of tasks given by *start times*, *durations* and *resource usage*, never require more than a *resource limit* at any time. In our case we use three sets of cumulative constraints: for CPUs, storages and links (see Table 1).

3.4 Simulations

The constraint satisfaction problem was implemented using MiniZinc modeling language [23] and Gecode [24] was used as a solver. The timelimit was set to 3 minutes for both planning and scheduling stages. The simulations were running under Windows 8 64-bit on a computer with Intel i5 (4 cores) 2.50 GHz processor and 6 GB of memory installed. The Gecode solver was running in a parallel mode using 4 threads.

Table 1: Variables and parameters used in cumulative constrains on resources.

Task	Start	Duration	Usage	Limit
Job	Js_{jc}	$duration_j$	1	cpu_c
Transfer	Ts_{fl}	$size_f \cdot slowdown_l$	1	1
File placement	Fs_{fc}	$Fdur_{fc}$	$size_f$	$disk_c$

The simulated environment consisted of 3 nodes: a central storage HPSS ($cpu_{HPSS} = 0$) which was the single source for input files and the single destination for output files, a local processing site and a remote processing site. The slowdown of links between the central HPSS and the local site was set to 0, which means that transfer overheads to/from the local site are negligible comparing to a job duration. The slowdown of the links to/from the remote site was increasing in each simulation proportionally to a slowdown factor. The parameters of jobs were taken from logging system of the STAR experiment's data production at computational site KISTI (Korea Institute of Science and Technology Information) [25]. The average job duration was 3,000 minutes and average time of transfer was 5 and 10 minutes to/from the remote site respectively (in the simulations where the slowdown factor = 1). Then, in further simulations the transfer times increase proportionally to the slowdown factor. In the simulated environment 80% of CPUs were available at the local site and 20% at the remote site. 2,000 of jobs were scheduled stepwise by subsets (chunks) of 200. Storage constraints were not considered in these simulations. Four different scheduling strategies were compared:

Local: All the jobs are submitted to the local site only. This strategy was used as a base line for comparison against other strategies.

Equal CPU load: Jobs are distributed between nodes with the goal to maintain an equal ratio of job duration per CPU. Each input file is transferred prior to the start of a job. At each node jobs are executed in input order.

Data transferred by job: Each CPU pulls a job from the queue when it is idle, then it has to wait for an input transfer before the job execution starts.

Optimized: This strategy is based on the model proposed in this Section.

The plot at Figure 9 shows the gain in a makespan delivered by different scheduling policies compared to the job execution at the local site only. The curves show the performance of the scheduling policies when an overhead of transfer to the remote site increases proportionally to the slowdown factor. When the transfer overhead becomes significant both heuristics ("Equal CPU load" and "Data transferred by job") fail to provide an efficient usage of the remote resources (the makespan improvement goes below zero). Negative makespan improvement means that, in this case, it would be faster to process all the data locally than to distribute it between several sites relying on the heuristic. The proposed global planning approach (Optimized) systematically provides a smaller makespan and adapts to the increase of transfer overheads better then the other

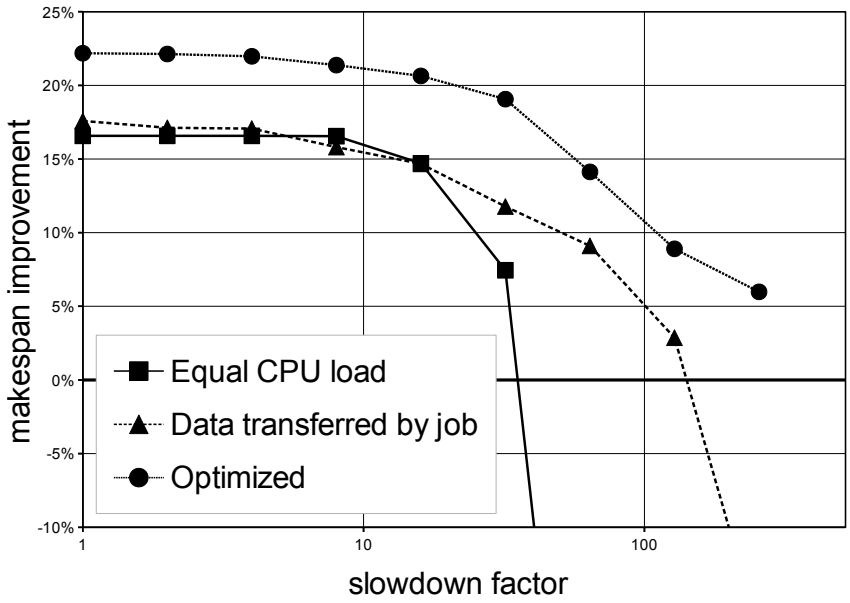

Figure 9: Results of simulations for real data production. Three strategies were evaluated and compared to a ideal local production. The optimized solution (our model) clearly provides the highest gain.

simulated heuristics. It provides a positive gain in makespan by using remote resources even when the transfer overhead is comparable to a job duration.

4 Enabling caching

Efficient usage of available cache space is important for transferring and accessing data in computational grids. Though, a great variety of caching algorithms is known, a study is needed to evaluate which one can deliver the best performance in data access considering the realistic demand patterns.

Cache cleaning algorithms can be applied to keep in the cache of data-transfer tools files that may be reused. The size of those cache is relatively small (several percent of the entire dataset) and the clean up has to take place regularly to make space for further transfers. Another task can, for example, be to delete a part of local data replica if no longer in use or requested. The problem posed by cache cleanup is to select and delete files which are the least likely to be used again. An investigation to find the most appropriate algorithm is required.

In this study, all the caching algorithm were implemented following the concept known as "water-marking". Water-marking is an approach where thresholds are set for the cache cleanup starts and stops. It considers the current disk space occupied by the data in cache and the high-mark and the low-mark for cache size are externally set up and can be adjusted as needed. When the used cache size exceeds the high-mark, the cache clean-up starts, and files are deleted until the used cache size gets below the low-mark. The time interval between clean-ups depends on combination of high/low marks, cache size and data-flow. With watermarking concept more computational demanding algorithms can be implemented as the cleanup may be independent of the data transfers.

4.1 Access patterns

Several data access patterns were extracted from log files of data management systems at sites of HEP/NPP experiments in order to simulate caching. Three different access patterns were used as input for our simulations:

STAR1: the pattern was extracted from Xrootd [3] logs taken from the STAR experiment's Tier-0 site of RHIC Computing Facility at Brookhaven National Laboratory (RCF@BNL), it consist of records made during a 3 months period (June-August 2012) of all datasets available in STAR.

STAR2: extracted from the same data management system, but of records made during a 7 months period (August 2012 - February 2013) under similar conditions.

GOLIAS farm is part of regional computing center for particle physics at the Institute of Physics (FZU) in Prague, and is part of a Tier-2 site for the CERN/ATLAS experiment. The facility also performs data processing for another experiment - AUGER, which makes less than 1% of the total requests. The pattern was extracted from DPM [4] logs for a 3 months period (November 2012 - February 2013).

The usage of access patterns corresponding to different time periods and experiments helps to verify the results of our simulations. As input of our simulations, we tried to focus on a few characteristic access patter. The key parameters we came up with for the three access patterns are given in Table 2. Both STAR access patterns have similar

Table 2: Summary of three user access patterns used as input for simulations. The selection of two time sequence in STAR and one from a different experiment aims at verifying stability of our result and findings.

		STAR1	STAR2	GOLIAS
Time period	months	3	7	3
Number of requests	$\times 10^6$	33	52	21
Data transferred	PB	50	80	10
Maximal number of requests for one file	–	192	203	94260
Average number of requests per file	–	19	15	5
Number of unique files	$\times 10^6$	1.8	1.7	3.8
Total size of dataset	PB	1.45	2	1
Maximal file size	GB	5.3	5.3	18
Average file size	GB	0.8	1	0.3

parameters. It is noteworthy to mention that the first one was taken right before the Quark Matter 2012 conference and the second one, right after. This is important as the access to data is intensified before a conference and without verification, it would be doubtful if our findings would be stable across time. The number of files requested only once during the period, is less than 10% in both cases.

The FZU/GOLIAS access pattern is taken from another experiment with different data-storage structure, DPM is used here within a Tier-2 data access context (user analysis). This access pattern is much less uniform and differs from the other two: the size of files is not explicitly limited and can reach 18 GB, the number of requests for a file varies from 1 up to 94260, with an average 5, while it varies from 1 to 203 for the STAR patterns. 44% of files were requested only once

Our analysis is not sensitive to the particular Data Management system, (Xrootd or DPM), further we explain differences in the access patterns with specifics of experiment and role of the cluster in the system (tier level).

4.2 Cache simulation

The efficiency of caching can be estimated by two quantities, the cache hits H and cache hits per megabyte of data H_d from here on referred to as *cache data hits*:

$$H = \frac{N_{cache}}{N_{req} - N_{set}} \tag{14}$$

$$H_d = \frac{S_{cache}}{S_{req} - S_{set}} \tag{15}$$

where N_{req} is the total number of requests, S_{req} -the total amount of transferred data in bytes, N_{set} -the number of unique files witch were requested at least ones during the considered period, S_{set} - the total size of those unique files in bytes, N_{cache} - the number of files transferred from cache, S_{cache} - the amount of data transferred from cache in bytes.

By maximizing the cache hits H one reduces the number of files transferred from external sources and thus reduces the overall make-span due to transfer startup overhead

for each file. By maximizing the cache data hits H_d one reduces the network load, since more data is reused from the local cache.

If the access pattern is completely random, the expected cache hit and cache data hits would be equal to *cache size/storage size*, so it can be useful to compare the actual cache performance to this estimation.

Altogether 27 different caching algorithms were simulated. But the majority of studied algorithms did not bring any improvements over the simplest one (FIFO). Only the algorithms that appeared to be the most efficient are discussed below:

- **First-In-First-Out (FIFO)**: evicts files in the same order they entered the cache. Performance of this trivial algorithm provide a good comparison benchmark against more sophisticated ones which can demand significant computational resources.

◯ **Least-Recently-Used (LRU)**: evicts the set of files which were not used for the longest period of time.

● **Least-Frequently-Used (LFU)**: evicts the set of files which were requested less times since they entered the cache.

★ **Most Size (MS)**: evicts the set of files which have the largest size.

\+ **Adaptive Replacement Cache (ARC)**[26]: splits cached files into two lists: L1 - files with *access count* = 1, and L2 - files with *access count* > 1. LRU is then applied to both list, the self adjustable parameter $p = cache\ hits\ in\ L1/cache\ hits\ in\ L2$ defines the number of cached files in each list. The general idea is to invest more into the list which delivers more hits.

∗ **Least Value based on Caching Time (LVCT)**[27]: Deletes files with the smallest value of the Utility Function:

$$UtilityFunction = \frac{1}{CachingTime \times FileSize} \tag{16}$$

where **Caching Time** of a file F is total size of all files accessed after the last request for the file F.

▽ **Improved-Least Value based on Caching Time (ILVCT)**[28]: Deletes files with the smallest value of the Utility Function:

$$UtilityFunction = \frac{1}{NumberOfAccessedFiles \times CachingTime \times FileSize} \tag{17}$$

where **Caching Time** is the same as for LVCT and **Number Of Accessed Files** is a number of files requested after the last request for selected file.

4.3 Simulation results

Three series of simulations with three access patterns were performed for each algorithm (90 simulations in total for each algorithm):

- 10 simulations with cache size 1-90 % of storage with fixed low-mark 75% and high-mark 95%. Those simulations aim to understand what would happen if we have to periodically clean a large storage. Those cases are aligned with a DPM and Xrootd use where most (if not all) the dataset(s) are in the system.

- 10 simulations with cache size 1.2 - 0.0025% of storage with fixed low-mark 75% and high-mark 85%. This high-mark was selected in order to leave enough margin (15%) in case when a large file is requested at the moment when the cache is almost full. We used those simulations to understand the behavior of cache cleanup if the cache size is by several orders less than the dataset size. This is in fact a most common case for transfer cache on data transfer nodes.

- 10 simulations with fixed cache size 10% of storage,fixed high-mark 95% and variable low mark within 0-90%. We performed those simulations to better understand the effect of data retention on cache (delete the least in hope of re-use).

In order to compare one algorithm against another an average improvement can be calculated in a following way:

$$Average\ improvement = \frac{\sum_{i=1}^{n} \frac{value2_i - value1_i}{value1_i}}{n}, \tag{18}$$

where n is the total amount of simulations with equal parameters for both algorithms, i is the number of the simulation,$value1$ - cache hits or cache data hits of a reference algorithm (FIFO), $value2$ - cache hits or cache data hits of a compared algorithm.

Table 3 contains the results of comparison of all the algorithms represented in this paper against FIFO. 60 values for each algorithm were taken from results of simulation series 1 and 2 in order to calculate the average improvement. According our results, the LFU algorithm does not bring any improvement over FIFO due to its well known flaw - it accumulates files which were popular for a short period of time, and those files prevent newer ones from staying in cache. The ARC algorithm was developed as an improvement to LRU, and not surprisingly, it outperforms LRU by ∼5% in cache hits and ∼7% in cache data hits. Therefore, LFU and LRU algorithms could be excluded from the further analysis in our case studies.

The graphical detailed results of simulations for all 3 series are given at Figures 10-12. The performance of FIFO and 3 algorithms appeared to be the most efficient (MS, ACR and LVCT) is presented at the plots.

Difference between Tier-2 and Tier-0 access patterns leads to distinct cache performance. Only the data dedicated for the ongoing analysis is placed at the Tier-2 site, while at the Tier-0 site all the experimental data is stored. As a result – the access pattern at the Tier-2 site has stronger access locality. STAR1 and STAR2 access patterns correspond to Tier-0 site and GOLIAS to a Tier-2 site. Thus, any particular algorithm at the plots delivers higher cache hits and cache data hits for GOLIAS access pattern than for STAR1 and STAR2.

The behavior of algorithms is similar within each dataset that is, their respective performance ordering is the same. This observation implies that if one of the algorithm appears to be most efficient for one of the datasets it is also the most efficient for the

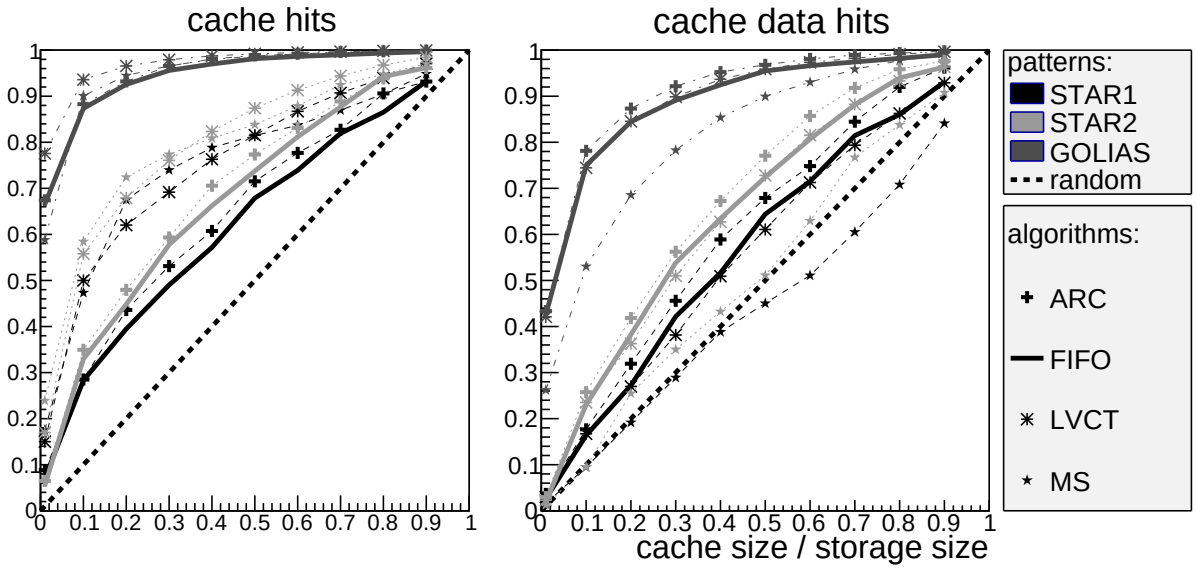

Figure 10: Results of simulation. Performance of algorithms for cache of larger size can be compared. For all of the simulations on this plot the following parameters were fixed: low mark = 0.75, high mark = 0.95

other datasets. This statement is also true for the rest of simulated algorithms not present on our figure. Though the communities represented by the STAR and GOLIAS access patterns are somewhat similar, this result is slightly surprising as our case studies represent two time sequence within the same usage and totally uncorrelated experiments. It would be interesting to study those algorithms in a different experimental context (outside the HEP/NPP communities) but this study is outside the scope of our research.

The MS algorithm has shown outstanding cache hits, but the lowest cache data hits. At the same time the LVCT has cache hits comparable to the MS while cache data hits are 2% improved over the FIFO. This algorithm could be an optimal when the cache hits is the target optimization parameter. The ARC algorithm has shown the highest cache data hits for studied access patterns.

The dependence of algorithms performance on low mark is presented at Figure 12. With higher low mark the number of clean-ups increases and that is why the difference between algorithms becomes more notable. Performance of efficient algorithms (FIFO, LRU, ARC and LVCT) increases steadily with the low mark. One should be careful when setting up a cache low mark at a particular site, since a higher low mark can increase cache performance significantly, but at the same time it can result in running cache clean-ups too often, consuming significant computational resources (and potentially increasing the chance to interfere with data transfers hence, degrading transfer performances if delete/writes/read overlap).

4.4 Selection of a caching policy

Performance of cache algorithms implemented with watermarking concept was simulated for a wide range of cache sizes and low marks. Three access patterns from Tier-0 and

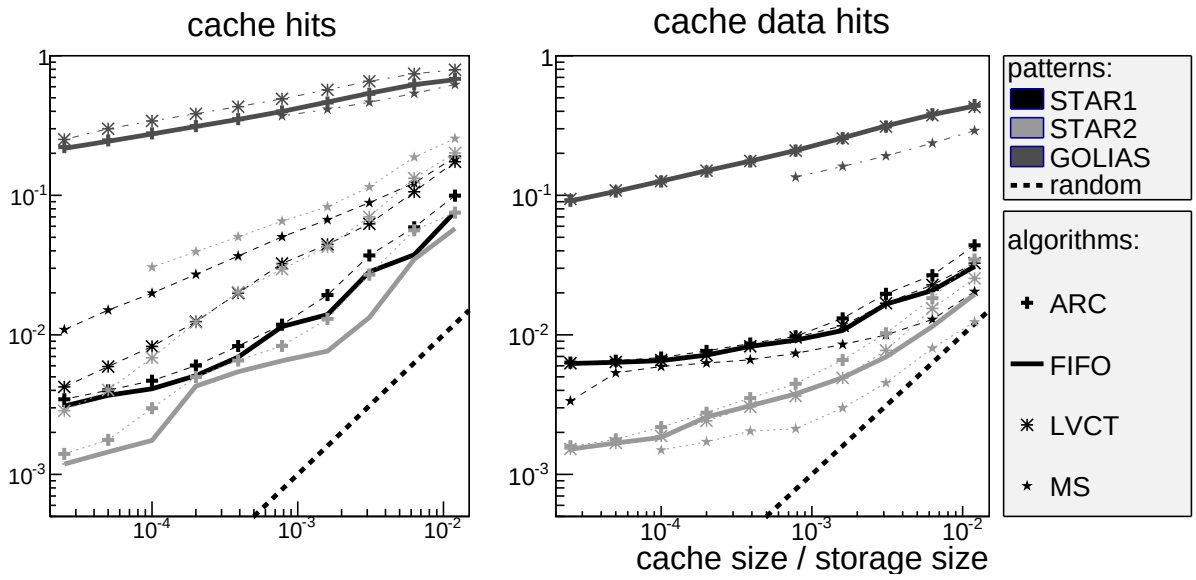

Figure 11: Results of simulation. Performance of algorithms for cache of smaller size can be compared. For all of the simulations on this plot the following parameters were fixed: low mark = 0.75, high mark = 0.85

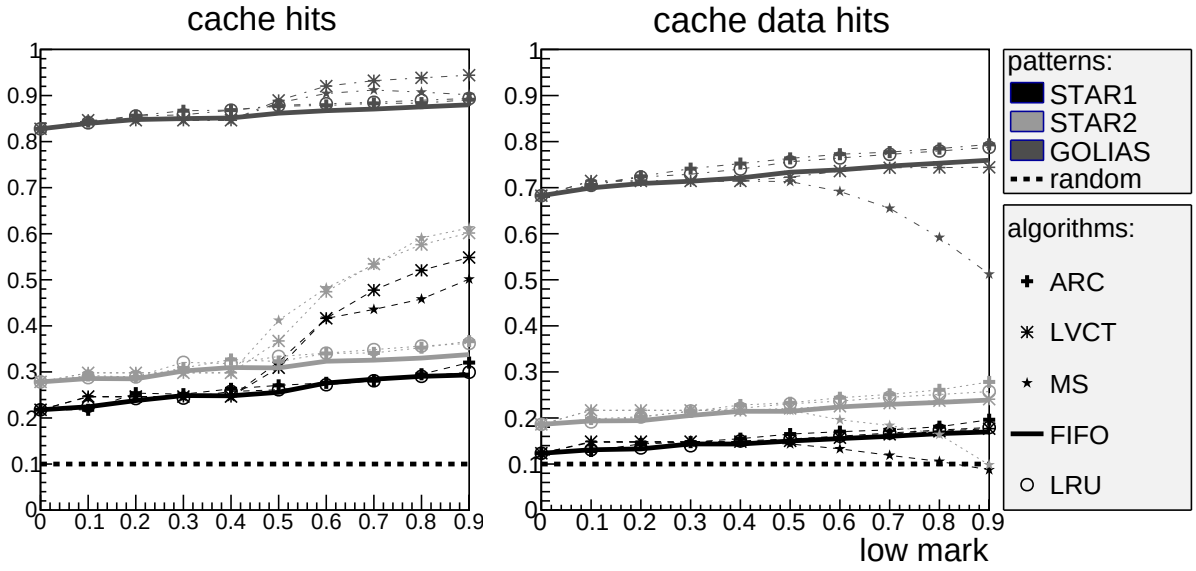

Figure 12: Results of simulation: dependence of cache performance on low mark. For all of the simulations on this plot the following parameters were fixed: cache size / storage size = 0.1, high mark = 0.95

Table 3: Average improvement of algorithms over FIFO.

Algorithm	cache hits	cache data hits
MS	**116** %	-20 %
LRU	8 %	5 %
LFU	-27 %	-18 %
ARC	13 %	**11** %
LVCT	**86** %	2 %
ILVCT	28 %	2 %

Tier-2 sites of 2 different experiments were used as input for simulations. Regardless of the cache size, Tier-level and specificity of experiment the LVCT and ARC appeared to be the most efficient caching algorithms for the communities we investigated. While we found the stability of relative algorithms' performance surprising at first, we attribute this result to an access pattern which is intrinsically similar in nature. An extension of this work could be the investigation of this result in a different experiment context which is a work beyond our initial goal. LVCT and ARC could certainly be safely implemented in tools such as RIFT.

- If the goal is to minimize makespan due to a transfer startup overhead the LVCT algorithm should be selected.

- If the goal is to minimize the network load the ARC algorithm is an option.

5 Conclusion and future plans

Distributed data processing in HENP as well as other data intensive applications for Grid require further improvement of load planning and job scheduling. Inspired by success of global planning for distributed systems [9] and for media streams [22] we extended [11, 12] these approaches to the entire data production in HENP, combining job scheduling and data management.

In Section 2 we proposed [12] a model of distributed data production, where all the files from a single source has to be processed once and transferred back. This model allows planning of WAN, storage and CPU loads using the network flow maximization approach.

In Section 3 a model for scheduling of data production over Grid was formulated in form of constraint satisfaction problem and solved using constraint programming [11]. The simulations based on data extracted from log files of batch and data management systems of the STAR experiment has shown that the proposed global planning approach systematically provides a smaller makespan and adapts to the increase of transfer overheads better then the other simulated heuristics.

Both described approaches can be combined in a two-stage scheduling. At the first stage the load of the resources will be planned using the network flow maximization problem knowing the current state of the Grid. At this point the amount of data that should be transferred over each link and processed at each site will be calculated. Then, at

the second stage, particular file transfers and jobs will be scheduled using our constraint programming model.

In addition to planning of data production, caching algorithms for user analysis were studied. The most widely used known algorithms were compared in multiple series of simulations using data access patterns extracted from log systems of two distinct experiments. Two best performing algorithms were selected to be used for cache management in HENP computational Grids.

The achieved results will be used in a distributed data production planner which is being developed. The planner will enable automated and scalable planning and optimization of distributed computations which are highly required in data intensive computational fields such as High Energy and Nuclear Physics.

The future development of global planning for data processing in Grid is ongoing. In future we plan to: combine both proposed approaches in a global scheduler; test this approach using full scale Grid simulations with the help of modeling tools widely used in Grid research community; improve the planner performance in order to enable online scheduling in real environment.

Acknowledgements

This work has been supported by the Czech Science Foundation (13-20841S, P202/12/0306), the MEYS grant CZ.1.07/2.3.00/20.0207 of the European Social Fund (ESF) in the Czech Republic: "Education for Competitiveness Operational Programme" (ECOP) and the Office of Nuclear Physics within the U.S. Department of Energy.

References

[1] L Betev et al. "Performance optimisations for distributed analysis in ALICE". In: *J. Phys.: Conf. Series* 523 (2014).

[2] J Horký, M Lokajíček, and J Peisar. "Influence of Distributing a Tier-2 Data Storage on Physics Analysis". In: *15th Int. Workshop on Advanced Computing and Analysis Techniques in Phys. Res.* 2013.

[3] *Xrootd*. `http://xrootd.slac.stanford.edu/`.

[4] *Disk Pool Manager (DPM)*. `https://svnweb.cern.ch/trac/lcgdm/wiki/Dpm`.

[5] *TORQUE Resource Manager*. `http://www.adaptivecomputing.com/products/open-source/torque/`.

[6] *Portable Batch System (PBS)*. `http://www.pbsworks.com/`.

[7] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency and Computation: Practice and Experience* 17.2-4 (2005), pp. 323–356. ISSN: 1532-0634. DOI: 10.1002/cpe.938. URL: http://dx.doi.org/10.1002/cpe.938.

[8] Jan Balewski et al. "Offloading peak processing to virtual farm by STAR experiment at RHIC". In: *J. Phys.: Conf. Ser.* 368.012011 (2012).

[9] M Zerola et al. "One click dataset transfer: toward efficient coupling of distributed storage resources and CPUs". In: *J. Phys.: Conf. Ser.* 368.012022 (2012).

[10] Dzmitry Makatun, Jérôme Lauret, and Michal Šumbera. "Study of cache performance in distributed environment for data processing". In: *Journal of Physics: Conference Series*. Vol. 523. IOP Publishing. 2014, p. 012016.

[11] Dzmitry Makatun et al. "Planning for distributed workflows: constraint-based coscheduling of computational jobs and data placement in distributed environments". In: *Journal of Physics: Conference Series*. Accepted. IOP Publishing. 2014.

[12] Dzmitry Makatun et al. "Model for planning of distributed data production". In: *Proceedings of the 7th Multidisciplinary International Scheduling Conference (MISTA 2015)*. Accepted. 2015.

[13] Dalibor Klusáček and Hana Rudová. "Multi-resource aware fairsharing for heterogeneous systems". In: *Job Scheduling Strategies for Parallel Processing*. 2014.

[14] Srividya Srinivasan et al. "Selective reservation strategies for backfill job scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2002, pp. 55–71.

[15] Dalibor Klusáček and Hana Rudová. "Performance and fairness for users in parallel job scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2013, pp. 235–252.

[16] Henri Casanova et al. "Heuristics for scheduling parameter sweep applications in grid environments". In: *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*. IEEE. 2000, pp. 349–363.

[17] K Ranganathan and I Foster. "Decoupling computation and data scheduling in distributed data-intensive applications". In: *11th IEEE International Symposium on High Performance Distributed Computing* (2002), pp. 352–358.

[18] Olivier Beaumont et al. "Bandwidth-centric allocation of independent tasks on heterogeneous platforms". In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*. IEEE. 2001, 6–pp.

[19] R K Ahuja, T L Magnati, and J B Orlin. *Network flows : theory, algorithms, and applications*. Prentice Hall, 1993.

[20] F Rossi, P Beek, and T Walsh. *Handbook of Constraint Programming*. Amsterdam: Elsevier, 2006.

[21] *High Performance Storage System (HPSS)*. `http://www.hpss-collaboration.org/`.

[22] Pavel Troubil and Hana Rudová. "Integer linear programming models for media streams planning". In: *ICAOR* 11 (2011), pp. 509–522.

[23] Nicholas Nethercote et al. "Minizinc: Towards a standard CP modelling language". In: *Principles and Practice of Constraint Programming–CP 2007*. Springer, 2007, pp. 529–543.

[24] Guido Tack, Mikael Lagerkvist, and Christian Schulte. "Gecode: an open constraint solving library". In: *Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP)*. 2008.

[25] *Korea Institute of Science and Technology Information KISTI*. `http://en.kisti.re.kr/`.

[26] Megiddo, Nimrod, and Modha. "Outperforming LRU with an adaptive replacement cache algorithm". In: *Computer* 37 (2004), pp. 58–65.

[27] Song Jiang and Xiaodong Zhang. "Efficient Distributed Disk Caching in Data Grid Management". In: *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER'03)* (2003), pp. 446–51.

[28] J P Achara et al. "An improvement in LVCT cache replacement policy for data grid". In: *Proc. of the 13th Int. Workshop on Advanced Computing and Analysis Techniques in Physics Research* (2010), p. 44.