AbacusFS integrated storage and computational platform

Isak Nuhić[*], Marjan Šterk[*], Toni Cortes[+]

[*] XLAB d.o.o., Ljubljana, Slovenia
[+] Barcelona Supercomputing Center, Barcelona, Spain
Corresponding author: isak.nuhic@xlab.si

Abstract - Today's applications, especially those in the scientific community, deal with an ever growing amount of data. Among the problems that arise from this explosion of data are how to organize the data so that the information about how the data was produced is not lost, how to ensure repeatability of calculations, how to automate calculations, and how to save computational and storage resources. The storage is currently a passive component that neither imposes any rules on data organization nor helps with associating a calculation with its result. We thus introduce AbacusFS – an integrated storage and computational platform which interacts with the file system and connects the calculations and the storage. Users and applications see it as a normal file system; however, the semantics are changed for files that are results of calculations done with the platform. When such a file is open for reading, AbacusFS first re-generates it if the input files used in the calculation have been modified. Alternatively, result files can be virtual, i.e. not stored anywhere but rather re-generated on every read access.

I. INTRODUCTION

Today's applications, especially those in the scientific community, deal with an ever growing amount of data [1]. Major problem in using these applications is tracking changes of the intermediate data. Users of these applications want to focus mainly on the final results and do not want to be considered with intermediate results, changes in the files, parameters, versions etc. They would like more automation and repeatability.

Using storage efficiently and economically is a major problem because of the increasing amount of data and the fact that advances in storage, particularly storage speed, lag behind advances in CPU speed. Depending on available CPUs, available storage devices and their location, it may be faster or cheaper to re-generate (re-calculate) the data whenever it is needed rather than storing everything. In HPC a lot of data is long term archived on tape. Re-calculating the data could sometimes prove more economical than getting it from tape. Furthermore, from the energy consumption point of view it may be optimal to start the calculation only when the results are needed instead of when the user has requested (because of the differences in costs in different times, i.e. computing costs at night might be lower than at day).

There are thus two significant and related problems: 1) data organization and 2) performance and efficiency. In order to solve the problems of data organization, reproducibility and automation of calculations we need to make calculations aware of changes in input files, parameters, software versions etc. The result that is needed by the user could then be re-calculated if something has changed. In the case of workflows of multiple calculations, changes to any initial or intermediate file could trigger re-calculation of the dependent part of the workflow. One of the problems that may arise here is the problem of nondeterministic applications in which the results might slightly vary in the last digits. For now we do not tackle these problems, rather our focus is on the deterministic applications. The performance and efficiency problem is correlated with the first one. Some intermediate files in the calculations are perhaps changed very often and it may be better to re-calculate them every time rather to store them permanently. Others may be accessed frequently but rarely changed, so they are better stored permanently or even replicated on multiple storage nodes.

To tackle these problems we propose AbacusFS which changes the paradigm in which we observe storage, or more specifically, file systems. Applications currently just use the storage without any interaction or integration between the two. Current storage systems neither impose any rules on data organization nor help with associating a calculation with its result. The basic idea of the solution presented in this paper is to make file systems application-aware in order to improve the efficiency of storage use, shorten the time to obtain calculation results, and automate certain repetitive tasks for the user.

This work was partially supported by the EU Marie Curie Initial Training Network SCALUS under grant agreement, the Spanish Ministry of Science and Technology under the TIN2007-60625 grant, and the Catalan Government under the 2009-SGR-980 grant, no. 238808.

A. *Related work*

There are two important research fields which relate to our solution: virtualizing processes with file systems and building file systems in user space.

One of the recent solutions in this area is the concept of a desynchronizing file system or DesyncFS [2]. DesyncFS deals with the problems of heterogeneity in supercomputers which imposes problems with scientific applications with a lot of internal dependencies. Due to different performances of the hardware the applications run at the speed of the least powerful processor. In order for these applications to perform better they need to be desynchronized.

In the traditional relationship between application and file system, control resides in the application and it calls into the file system as needed for storage. This can be described as a push-pull relationship. The file system is passive and the application actively pushes and pulls data. A desynchronizing file system inverts this relationship, making the file system active and the application passive [2].

Solution which we propose deals with different problem but it also tries to change the relationship between storage and applications. One other difference is that in order to use DesyncFS a user has to adapt the applications for it where in the case of our solution there is no need for any changes in the applications.

Linux has a long tradition of user-space file systems (e.g. Network File System (NFS) [4] was implemented this way for quite some time. User-space file systems are not widely used mostly because of performance and security reasons [5]. However there are advantages in building a file system in user space: an ability to build and modify the file system without changing the kernel; shaping the file system to the needs of the user with the ability to use functions from the user space. One of the most important projects in that area is FUSE [6]. File System in User Space (FUSE) is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces. FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual file systems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.

One of the ever growing problems in the scientific community is managing and running workflows. Provenance in the context of workflows, both for the data they derive and for their specification, is an essential component to allow for result reproducibility, sharing, and knowledge re-use in the scientific community [7]. Scientific workflows has become an important area of research within workflow platforms and environments. Workflows are supported in the AbacusFS and in the future we would like to extend that feature to add more workflow management. Some important platforms and frameworks in this area are: Taverna [8], Askalon [9] and Trident [10].

The concept of using file systems to store dynamic (non-persistant data) is present for some time. The Linux proc file system [11] has been around from 1984 and it is used widely Today. The device file system (devfs) [12] is also widely used for presenting device files. It provides a powerful new device menagement mechanism for Linux.

II. DESIGN AND ARCHITECTURE

A. *General design*

We introduce an integrated file system/computational platform that connects calculations and storage. This platform is currently implemented on a single machine with plans to extend it to distributed systems (cluster/grid/cloud), where the advantages of such a solution will be more evident. The users of the platform will be able to deploy scientific calculations without worrying about re-calculation every time they make some changes in the calculation parameters or input files. They will also be able to see how each file was produced.

Let us illustrate the usage of the platform with an use case. Whenever a user runs some calculation, the full command-line and other information is stored as metadata associated with the calculation's output file(s). Then, each time one of the output files is open for reading, it is first checked whether it is up-to-date, i.e. whether the input files have changed. If they have changed, the file is automatically re-generated by running the original command again and file access is allowed only after the re-calculation has finished. All existing command-line applications can be used without modifications, except for the applications that write to pre-existing files. The fact that input files can also be results of previous calculations does not pose any problem, so workflows are supported natively.

Running calculation when the command is issued and then re-calculating whenever the results are needed and any inputs have changed is not always optimal. For example, in next generation supercomputing, moving data from the storage to the computing nodes may be more time and resource consuming than recomputing this information because we may have more available idle CPU power. A part of our platform is thus a decision-making process that can take into account parameters such as file size, number of read accesses, number of re-calculations of the file, and a user-assigned file importance. In the future we plan to refine the decision-making process and with it add more parameters.

B. *Architecture overview*

AbacusFS has four main modules, as shown in Figure 1. These are:

1. the Abacus file system implementation,
2. the decision making module,

3. the helper script for running the calculations,
4. the database where extended metadata is stored

C. *Abacus file system*

The FUSE-based file system is the core module of our solution. It uses a directory on another file system for backing storage. It is the only module that communicates with all other modules. The users and applications see it as a normal file system; however, the semantics are changed for files that are results of calculations done with the platform, i.e. files that have certain extended attributes set. When such a file is opened for reading, it is by default re-generated if needed, as explained above. After the file is accessed and read it can be 1) stored permanently or 2) not stored – instead re-generated on every access.

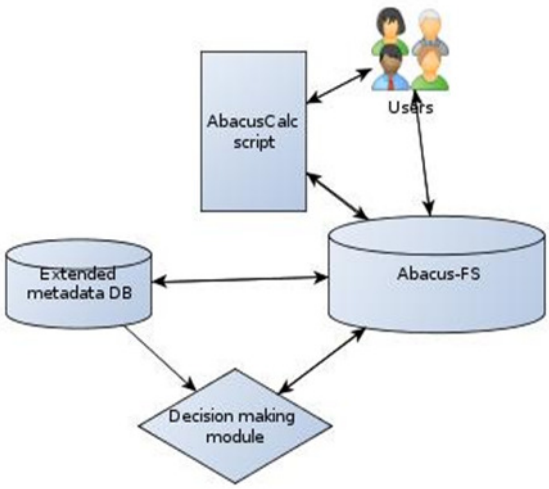

Figure 1. Over*view of the architecture*

D. *Decision making module*

The decision of 1) storing the file permanently or 2) re-calculate it every time it is accessed, is made by the decision making module. Since one of our requirements is to use the storage more efficiently and economically, we need to add some intelligence to these decisions. We propose an intelligent agent based on fuzzy logic [13], a proven principle that is widely used in fields such as computer science (e.g. bioinformatics, image processing, embedded systems), automation, process control,h etc.

The decision making process takes place every time the file is accessed. The agent must currently decide between the two options mentioned above. Even though this may seem trivial, the agent is extensible so that in the distributed version it will also decide where to put files and calculating processes, which files to replicate etc. Efficiency of resource use, performance, and reliability thus depends on this module.

We currently observe the following variables as the basis for the decision: the number of read accesses to the file, the size of the file, the number of time the calculation was run and the value user assigns to the file (optional).

E. *Helper abacuscalc script for running calculations*

The third module is the script for running the calculations. Whenever a user wants to run some calculation for the first time he needs to use this predefined script, which takes care of notifying the file system how the calculation is to be run, what input files it depends on, what output files it produces etc. For a trivial example, let us suppose that the user wants to run:
cat -n infile1 infile2 > outfile,
and he wishes the *outfile* to be managed by the file system as a calculation result.
Outfile must, of course, be located on the abacus file system, while the input files can be anywhere. The user must run the *abacuscalc* script *with* the following modified command line:
./abacuscalc.py cat -n -in infile1 -in infile2 -stdout outfile
Consecutive input files must be preceded by *-in*, consecutive output file by *–out* and consecutive non-file parameters with *–opt*. Similarly, < is replaced with *-stdin*, > with *stdout* and *2>* with *-stderr*.

The helper scripts forwards the information about the calculation using special, write-only files in a reserved part of abacus file system, similar to the Linux' *proc* file system.

F. *The extended metadata database*

The extended metadata database is a Redis [14] based database which is used to store extended metadata of the files and calculations. It is essentially a noSQL key value store in which we store two important types of data:

- associations between each output file and the calculation that produced it,
- all the informations about each calculation that are required to re-run it plus some statistics used by the decision making module.

Whenever a file is accessed the metadata database is consulted to check whether the file depends on (i.e. is the result of) a calculation. If this is the case then the record describing this calculation is also read.

The database may grow large, particularly once the platform is extended to distributed systems. We plan to use similar approaches as the existing metadata servers of Luster[15], PVFS[16] and other parallel file systems.

With this metadata database we have extended the metadata concept which would be harder to do if we were to use only extended attributes fields of the files. The problems with using extended attributes on the local machine would be the increase in file size an thus a performance problem. Using the metadata database is also simpler to develop, has no limitations for data model or size. But the major would be with the files that are not stored permanently for which we can not use extended attributes. Also in the future when AbacusFS is extended on multiple machines the need for a centralized metadata database will appear.

III. IMPLEMENTATION

In this section we will describe some technical details of the modules introduced in the previous section and also interactions between them.

A. *Communication between abacuscalc script and Abacus file system*

As said before, the helper script uses write-only files in a special directory to forward the information on the calculation to the file system. Such a directory is created for each new calculation and named after a calcualtion's unique identifier (CUID) that the file system assigns to it. To illustrate how the communication works, we will show the equivalent commands that the user could enter for the before mentioned example of calculation

cat -n infile1 infile2 > outfile.

The whole process is explained in Table I. We will assume that abacus file system is mounted at */abacusfs*.

TABLE I. EXPLANATION OF THE COMMAND-LINE OF THE ABACUSCALC SCRIPT

Command-line equivalent of helper script action	Comment
# cd /abacusfs/newcalc	
# ls	Each *ls* in directory */abacusfs/newcalc* signals a new calculation. The file system will now generate a new CUID, e.g. 34ac5, and return the directory 34ac5 as the only file in /abacusfs/newcalc.
# cd 34ac5	
# echo 'cat -n \ /home/isak/infile1 \ /home/isak/infile2 \ > outfile' >cmd	Note that if the helper script is bypassed, which we only do here as illustration, absolute file paths must be used everywhere.
# echo '/abacusfs/isak/infile1' > in	
# echo '/abacusfs/isak/infile2' >> in	
# echo 'outfile' > out	
# cat status	Reading the status file signals that all the information has been written.

Once the *status* file is being opened for reading, the file system stores all calculation metadata to the database using the key *34ac5*. It then asks the decision-making agent whether this calculation should be first run right away. If the decision is no, the file system immediately returns as the contents of the *status* file a value that signalizes that the calculation has not been started. If, on the other hand, the decision is yes, the file system will:

1. run the calculation and wait for it to finish,
2. set an extended attribute of file *outfile* that will mark it as the result of calculation *34ac5*,
3. return an *OK* value as the contents of the *status* file. This will results in the user having to wait for the calculation to finish, which is the expected behavior.

B. *Extended metadata database*

Redis, which our metadata database is based on, is an advanced open source key-value store. The key of each record must be string while the value can be a string, a map (termed a hash in Redis), a list, a set, or a sorted set. Redis supports atomic operations such as appending to a string, incrementing the value in a hash etc. Redis works in-memory to achieve better performance but can also make data persistent by dumping the dataset to the disk on request or by appending each command to a log. In our solution we decided to dump the dataset to disk but also to use logging as well in the case of system. Since we use redis hashes as data structures to store information, it is important to underline that redis hashes are stored in such a way that takes very little space, so it is possible to store millions of objects in a small Redis instance.

Two pieces of data we currently store in the metadata server database are shown in Table II.

TABLE II. SAMPLES OF THE DATA STORED IN THE METADATA SERVER DB

Files that are derived from some calculation:	**'34ac5'' calculation hash**	**Explanation**
'outfile1:34ac5'	'in': '/abacusfs/isak/infile1:1234 /abacusfs/isak/infile2:1234'	Input files with absolute paths and ctimes[1] of the files.
	'inno': '2'	Number of input files.
	'out': 'outfile:1234'	Output files with relative paths to /abacusf and ctimes of output files.
	'outno': '1'	Number of output files.
	'count': '45'	Number of times the calculation was run (the file has been re-calculated)
	'time': '0.351'	Time it needed for the calculation to execute (last execution time).
	'cmd': 'cat -n \ /home/isak/infile1 \ /home/isak/infile2 \ > outfile'	Full command-line of the calculation.
	'nfa': '1034'	Number of times tihe file was accessed.
	'uv': '3'	A value that user attaches to the file when running the calculation for the first time trough *abacuscalc* script.

[1] Currently, when working on the local machine we read the ctimes from the stat structures of the files but when the soulution is extended on multiple machines we will need a metadata server which will serve all nodes. Of course that would meen that we also need a synchronization of the clock between nodes.

C. Decision making module based on fuzzy logic

As said before, the intelligent decision making agent currently decides only between two options:

1. the calculation is run the first time with the *abacuscalc* script and afterwards every time the file is accessed but it is out-of-date, and
2. the file is not stored permanently but rather calculated every time it is accessed. Obviously, in this case the calculation should not be run when the helper script is invoked.

The agent bases its decisions on the following variables:

- number of accesses to the file (NFA) – this parameter tells us how often the file was accessed, which is important in the decision-making process, because the more the file is used will tip towards storing it permanently,
- size of the file (SF) – since one of the important issues is using the storage efficiently this parameter is very important,
- number of time the calculation was run (NCR) – this parameter will tell us how often is the file re-calculated,
- the value user attaches to the file (optional) (UV) – we need the parameter which will give the value that user can attach to the file thus making it more or less important.

All variables except SF are stored in the metadata database. UV can be changed by the user and is by default 5 out of 10. The extreme values 0 and 10 are intended to override most of the influence of the other variables.

Each variable is a member of three fuzzy sets (small, medium and large). The boundaries of the sets are different for different variables, e.g. a smallish 30 MB file could have a 0.6, 0.4, 0.0 memberships in the three sets, respectively.

The decision variable DV is obtained to determine what to do with the file. It can take two possible intervals:

[1] [0 –50] – re-calculate every time

[2] [50– 100] – store permanently

The agent uses a set of rules to obtain such as:

if NFA == large and SF == small and NCR == small and UV == large then DV = store_permanently

if NFA == small and SF == large and NCR == large and UV = small then DV = recalculate_every_time

If the file is small in size, accessed frequently and has a high user value assigned it is optimal to store the file permanently. On the other hand if the file is large in size but accessed rarely and has small user value it may prove better to re-calculate every time the file is needed. Thus, user-given file importance, calculation time, frequency of access will tip the scales towards storing the file permanently and the output file size will tip it towards re-calculating the file when needed. One other important variable which will be added latter in the process is the time needed for the execution of the calculation. The shorter the calculation time the more likely it is that the file will be re-calculated and if the time grows large it may prove optimal to store the file permanently.

Since we have four input variables which are members of three sets we get $3^4 = 81$ rules. This currently crude process is a starting point for a search of optimal set of rules. Particularly the distributed version of the platform will require additional variables and a detailed study of the rules that achieve the best performance.

IV. TESTS

The AbacusFS platform obviously performs similarly to the underlying file system, only imposing an overhead when dealing with calculation-dependent files. The following operations could potentially induce a measurable overhead:

1. running the calculation for the first time, as opposed to running it without AbacusFS,
2. checking whether the file is up-to-date on each read access,
3. re-running the calculation when it is not up-to-date.

Note that the overheads of running calculations include the decision-making process. The performance of the in-memory files which are not stored permanently has not yet been tested.

The tests were conducted on the machine with Intel i7-2600 3.40 GHz CPU, Intel 80 GB SDD, on the 3.0.0. Linux kernel.

A. Running the calculation for the first time as opposed to running it without AbacusFS

We have conducted tests with the simple *cat a b > c* command where the input files *a* and *b* are located on the underlying file system and the output file *c* is stored on Abacus file system. The size of input files varied from 0 to 2 MB and tests were conducted with flushing the cache every time before the command was run and without flushing the cache. The results are in tables III. and IV and they show that the overhead increases with file size.

When comparing these results it can be seen that overhead also increases when the cache is flushed. Abacus file system uses a directory on the underlying file system for backing storage. FUSE in its implementation does not handle caches by itself, but relies on kernel caching. This means that the data is being cached twice since kernel sees abacus file system and the backend file system as different ones. We also did more tests to find out which part of the whole process takes most time. These tests showed that the time for execution of the command increases as file size increases and communication between abacuscalc script and abacus file system remains constant. This indicates latency issues more than bandwidth issues. We proved this by testing

with flushing only page cache and directory entries and attributes, which showed that additional overhead comes when the page cache is flushed. Result is anticipated since FUSE always introduces some additional overhead related to data read/write operations. In the future we will try to optimize it and thus decrease the overhead.

TABLE III. TEST OF RUNNING THE CALCULATION FOR THE FIRST TIME ON ABACUSFS AND ON UNDERLYING FS WITH FLUSHED CACHE

Size of input files	AbacusFS – with flushed cache	Underlying FS – with flushed cache	Overhead
0 B	0.0544 s	0.0098 s	44.6 ms
4096 B	0.0639 s	0.0104 s	53.5 ms
0.5 MB	0.0729 s	0.0202 s	52.7 ms
1 MB	0.0825 s	0.0240 s	58.5 ms
2 MB	0.1117 s	0.0373 s	74.4 ms

TABLE IV. TEST OF RUNNING THE CALCULATION FOR THE FIRST TIME ON ABACUSFS AND ON UNDERLYING FS WITHOUT FLUSHED CACHE

Size of input files	AbacusFS – without flushed cache	Underlying FS – without flushed cache	Overhead
0 B	0.0177 s	0.0021 s	15.6 ms
4096 B	0.0180 s	0.0023 s	15.7 ms
0.5 MB	0.0288 s	0.0023 s	26.5 ms
1 MB	0.0360 s	0.0064 s	29.6 ms
2 MB	0.0496 s	0.0113 s	38.3 ms

B. *Checking whether the file is up-to-date on each access and re-calculating if it is not*

Option of checking whether the file is up-to-date and re-calculating the files which are out-of-date is the major option of AbacusFS. Since file systems by default do not have this option the tests were conducted in two parts. First the files are up-to-date and we are measuring the overhead of read access to the files on abacus file system in opposed to the underlying file system. In second part the files are not up-to-date and they are automatically re-calculated. This measurement gives an information how much more overhead does the re-calculation part produces in the system. The tests were done only on small (6 bytes in size) files with and without flushing cache.

TABLE V. READ ACCESS TIME ON UNDERLYING FILE SYSTEM AND ON ABACUS FILE SYSTEM

Underlying FS – flushed cache	AbacusFS – flushed cache	Underlying FS – without flushed cache	AbacusFS – without flushed cache
0.0074 s	0.0151 s	0.0017 s	0.0035 s

TABLE VI. TIME NEEDED FOR READ ACCESS AND RE-CALCULATION WHEN THE FILE IS NOT UP-TO-DATE ON ABACUS FILE SYSTEM

AbacusFS – flushed cache	AbacusFS – without flushed cache
0.0232 s	0.0035 s

When comparing the results from Tables IV. and V. it can be seen that the re-calculation brings no additional overhead without flushed cache and with flushed cache only another 8 ms. This results is very important since the option of automatic re-calculation is one of the key advantages of AbacusFS.

C. *Workflows*

The third test which we conducted considers workflows. Since it was already said that workflow support is one of the key advantages of AbacusFS we needed to test it.

In order to test it we made a simple workflow which consists of simple cat commands:
cat a b > c,
cat a b c > d,
cat a b c d > e,
cat a b c d e > f,
where *a* and *b* are on the underlying file system and all other files or on abacus file system. The files are small in size (10 bytes). In underlying file system if the file *a* is changed it is necessary to manually run all the commands in order to get the file *f*. In AbacusFS if the file *a* is changed and file *f* read the whole workflow is automatically run and the file *f* re-calculated.

D. *Tests overview*

The conducted tests showed that for files smaller than 0.5 MB overhead is between 13 and 30 ms. The problem with larger files is within the FUSE implementation. Tests with the workflows show two things: 1) that the workflows are supported natively in AbacusFS and 2) that this does not causes any extra overhead.

The large files problem needs to be assessed more in the future and with optimizing the file system module decreased.

V. CONCLUSION

This paper has introduced the AbacusFS computational platform which tries to change the paradigm of storage such that the file system interacts with computations, thus connecting computational tasks and storage. It tries to solve two problems: 1) automating the process of running scientific calculations which are complex with lots of intermediate results and lots of dependencies thus reducing the need for user interaction; and 2) optimizing the use of storage to use the resources more efficiently and economically. Access to the files is changed in a way that if the file is a product of some calculation it can be re-calculated if some inputs of the calculation changed. Also some files are always re-calculated rather then saved permanently to save storage resources. All this is done by building a new file system and connecting the calculations and the file system with the extended metadata database that extends the metadata concept.

Preliminary tests show that the overhead of the platform is small for small files and larger overhead for larger files. The tests have also shown that adding the option of automatic re-calculation (which also adds the option of workflow support natively) does not bring any new overhead to the system. The scalability to large number of files and calculations is currently being assessed.

Future work involves primarily refining the decision making process, refining the metadata concept, optimizing the FUSE based file system module and extending the solution to multiple machines (cluster/grid/cloud). The latter will also require incorporating a task scheduler, a resource manager and a distributed file system for backing storage. Furthermore we may add some prediction of users' behavior in order to achieve more automation as well as more efficient use of storage resources.

REFERENCES

[1] A. Ailamaki, V. Kantere, D. Dash, "Managing Scientific Data", Communications of the ACM Vol. 53 No.6, pp 68-78, 2010.

[2] L. Stein, D. Holland, M. Seltzer and Z. Zhang, „Can a file system virtualize a processors?", Association for Computing Machinery, Inc., March 2007.

[3] A. Shoshani, S. Klasky, R. Ross, Scientific data management: Challenges and approaches in the extreme scale era, SciDAC 2010.

[4] C. M. Smith. Linux NFS faq, http://nfs.sourceforge.net/, 2002.

[5]FUSE - implementing filesystems in user space, http://lwn.net/Articles/68104/, 2006.

[6] M. Szeredi. File System in User Space. http://fuse.sourceforge.net, 2006.

[7] S. B. Davidson, J. Freire , „Provenance and Scientific Workflows: Challenges and Opportunities", SIGMOD Conference, 2008.

[8] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble, „Taverna, reloaded", in SSDBM 2010, Heidelberg, Germany, 2010.

[9] R. Prodan, „Specification and runtime workflow support in the ASKALON Grid environment", Scientific programming, vol 15, no 4, IOS Press, 2007.

[10] Microsoft External Research, „Trident Workbench: A Scientific Workflow System", December 2008.

[11] T. Bowden, B. Bauer, J. Nerin, S. Feng, „The /proc file system", original: 1999., update: 2009.

[12] R. Gooch, „The Linux Device File-system", EMC Corporation, 2002.

[13] Z. Kovacic, S. Bogdan, Fuzzy Controller Design: Theory and Applications, Taylor & Francis Group, NW, 2006.

[14] S. Sanfilippo, Redis key-value store, http://www.redis.io, 2009.

[15] F. Z. Boito, R. V. Kassirc, P. O. A. Navaux, „Evaluating the Performance of Lustre File System" VII Workshop de Processamento Paralelo e Distribuído, Porto Alegre, 2009.

[16] Kuhn M. M., Kunkel J. M., Ludwig T, „Dynamic file system semantics to enable metadata optimizations in PVFS", Concurrency and Computation: Practice and Experience, 2009