Application of remote debugging techniques in user-centric job monitoring

T dos Santos[1], P Mättig[1], N Wulff[2], T Harenberg[1], F Volkmer[1], T Beermann[1], S Kalinin[1] and R Ahrens[1]

[1] Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany
[2] Fachhochschule Münster, Hüfferstraße 27, 48149 Münster, Germany

E-mail: dos.santos@physik.uni-wuppertal.de

Abstract. With the Job Execution Monitor, a user-centric job monitoring software developed at the University of Wuppertal and integrated into the job brokerage systems of the WLCG, job progress and grid worker node health can be supervised in real time. Imminent error conditions can thus be detected early by the submitter and countermeasures can be taken. Grid site admins can access aggregated data of all monitored jobs to infer the site status and to detect job misbehaviour. To remove the last "blind spot" from this monitoring, a remote debugging technique based on the GNU C compiler suite was developed and integrated into the software; its design concept and architecture is described in this paper and its application discussed.

1. Introduction

In October 2009, the Large Hadron Collider (**LHC**) at the European Centre of Particle Physics Research, CERN, Geneva, has begun operation. In its four big experiments, **Alice**, **LHCb**, **CMS** and **ATLAS**, it generates massive amounts of data (about 47 PB) per year.

Readout, transfer, storage and - most importantly - analysis of this amount of data is a computationally intensive, challenging task. Not only single computers, but even whole computer clusters are undersized for such dimensions in computing and storage, and users from all over the world need fast access to the data. To address the enormous scope of this computational challenge, the world-wide LHC Computing Grid, **WLCG**, was created. Being a computing grid as defined by Foster, Kesselman et.al.[1], the WLCG is a distributed computing, data transfer and storage facility, which uses the grid middleware **gLite**[2] for operations.

The computational work is divided into fragments called **grid jobs**, of which large amounts can be managed and run in parallel. The jobs are executed on multiple computing sites called **Computing Elements** (CEs), each providing computing resources in form of clusters and batch systems with computers organized as **Worker Nodes** (WNs), and providing the needed data on storage sites named **Storage Elements** (SEs). In beforehand, the user has no knowledge of where his grid job will actually be executed[1].

[1] however, by specifying requirements, for example selecting CEs by available cores, memory, connectivity or available data fragments, he can take influence in the middleware's decision where to send his job

2. Motivation for a user centric grid job monitor

ATLAS grid jobs usually consist of at least five parts, implemented in several different programming languages and data formats.

(i) **An algorithm generating data or performing calculations on data**

This algorithm (= the "user algorithm") is implemented in C++ and utilizes library functions provided by the ATLAS software framework Athena[3] and analysis tools such as ROOT[4]. It is this code that does the actual analysis work: the generation of physics data (e.g. Monte Carlo or detector simulation), the conversion of this data into other formats for further processing or the analysis of such data. The binaries with these algorithms are either compiled locally on the user's machine and then submitted along the grid job, or they are sent in source code form and compiled directly on a worker node by a special form of grid job (the build job).

(ii) **Code setting up Athena and the environment**

The Athena environment is set up by a collection of shell- and python-scripts. Their main purpose is to set up binary search paths, determine platform specifics and other runtime information.

(iii) **Athena runtime code**

Athena itself consists of several modules written in Python which acquire and prepare the analysis[2] data and then call the user algorithm with the prepared data (e.g. physics event data). For this, the user algorithm is loaded before the job's main event loop execution as a shared library by Athena. The runtime furthermore provides services like data I/O, logging and error handling.

(iv) **A short launcher application, usually implemented as shell-script**

The grid job executable (that is, the executable launched by the grid middleware on the worker node) typically is a short launcher script that loads the Athena environment-setup scripts and triggers execution of Athena. It is either written by the user or automatically created by a job submission and management tool like PanDA[5] or Ganga[6].

(v) **The data that is being processed**

The data usually is stored on the SE located at the site the user job got scheduled to run at. According to the ATLAS computing model, jobs generally are only to be sent to sites providing the needed data locally. The data is accessed by the job by means of mass storage software developed specifically for ATLAS needs, for example **dCache**[7].

To summarize, ATLAS grid jobs are generated or user-written shell scripts running a framework of Python scripts that use algorithms from C++ shared libraries (figure 1), executed on arbitrary grid computing elements with varying available software- and hardware infrastructure, performing calculations on large, distributed datasets in multiple data formats.

The fact that this application model consists of several layers depending on each other, implemented by different groups of developers using different programming languages, already implies a number of possible failure conditions and errors. On top of that, one observes the usual errors contained in all human-written software (invalid data and memory access, floating point exceptions, logical errors, etc) and errors caused by the context, that is, the grid (submission errors, failures to query needed local and remote services, data lookup- and acquisition errors, environment errors and so on). Hence, there is a wealth of failure possibilities from which only a small subset is under control of the user.

[2] or simulation, data generation, conversion, etc.

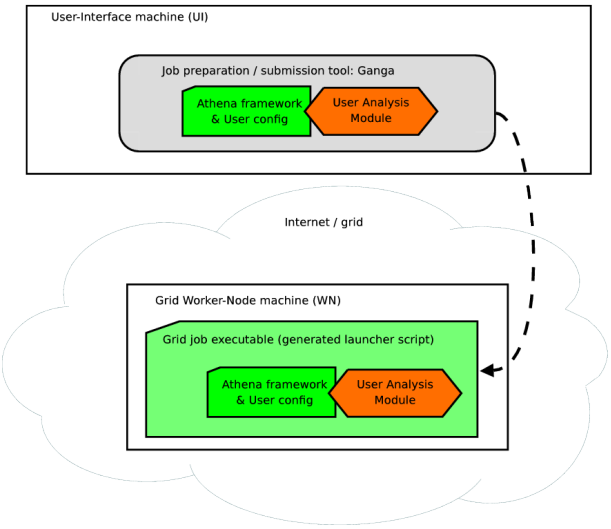

Figure 1: Submission of an Athena-analysis job to the grid using Ganga. The Athena-launcher is wrapped into a job script and uses the binary user analysis module.

To make things worse, a grid job's execution is an intransparent process for the user ("black box"), comparable to a **batch job** on a legacy batch computer: from the point of submission to the termination of the job (successful or not), one has no (or almost no) insight into the job's processing. Non-critical run time errors and invalid output data[3] can only be detected after the job finished. Likewise, the general progress of the job is not known during its run time. Finally, the health of the worker node the job is being executed on (resource usage, service status, etc) is not visible to the user.

The **Job Execution Monitor** (JEM), a user centric grid job monitoring software, has been developed to deal with these problems.

3. The Job Execution Monitor

3.1. Overview

The Job Execution Monitor[8, 9, 10, 11] (JEM) is a distributed application developed to provide real-time worker node health- and job progress-monitoring. JEM provides job status-, job error- and environment data to the user in real-time. Its main module is submitted alongside user jobs to grid computing elements, and is executed in parallel to the user job on its worker node (see figure 2).

During the run of the user job, vital system information on the worker node the job was scheduled to run on, like CPU and memory usage and free disk space in the job's working directory and temporary directories, is gathered periodically. The scripts the user job consists of can be run in a supervised line-by-line mode, during which events like function and method calls, returns from functions and methods, exceptions being raised, or shell commands being executed are logged. Furthermore, log files can be watched and monitoring events can be created on key phrases (like progress annotations, error messages, etc).

All monitoring data gathered during the job run is transmitted in real-time to a messaging broker to allow for direct analysis by the user. The user can either subscribe directly to the

[3] invalid from the user's point of view / semantically invalid

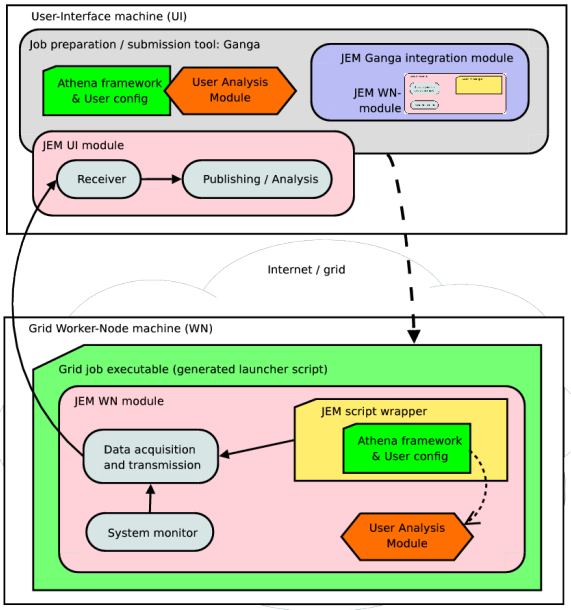

Figure 2: Workflow of job submission with JEM enabled. JEMs worker node module is run in addition to the Athena-launcher. At the same time, on the UI machine, the receiving module of JEM is run, receiving and publishing the monitoring data.

broker, thereby receiving all monitoring data of "his" job in real-time, or he can use JEM-provided summary web pages displaying aggregated monitoring data generated automatically from the data stream transferred via the broker. This analysis may lead to user decisions like aborting and re-submitting the job. "Post-mortem" job failure analysis even if all of the job's output was discarded by the grid middleware as a result of the failure is also possible by referring to the recorded monitoring data.

3.2. Limits of JEM without C tracing

Using the former version of JEM, the supervised line-by-line execution of the user job performed on the WN is performed for bash- and python scripts only. Hence, all of the job's initialization- and Athena management functions, like the environment setup and the staging of needed analysis data, is monitored. Errors in this part of the job run are discovered easily this way. The actual data analysis or generation code, however, cannot be monitored as it is implemented in C/C++ and shipped to the WN as object file or shared library. This user algorithm usually accounts for the biggest part of the job's run time.

So, great parts of the job's run - parts where logical errors might happen which lead to undesired and faulty job results - are only monitored in terms of system metric data. Because of that, such invalid job results are only discovered after the whole job has completed, possibly after several hours (or more) of job run time. If monitoring data would be available also during the execution of the user algorithm, the user might be alerted that the results differ from the expectations and so would have the possibility to abort the job run, correct the algorithm and re-submit it, saving time and increasing resource efficiency.

Similarily, job failures caused by infrastructure problems like depleted resources (disk space, memory), expired grid proxy certificates and the like can be more easily identified if monitoring information is provided during the execution of the user algorithm. Such problems often surface as algorithm crashes (null pointer references, floating point exceptions, segmentation faults etc.)

and are not distinguishable by itself from similar crashes caused by sloppy implementation, unless such monitoring data is available.

Finally, more often than not, grid jobs fail in a manner where the execution of the job just stalls in mid-processing, and the job eventually fails due to exceeded runtime. Usually, also this kind of failure occurs with the job's execution progress inside the binary user analysis, not inside the set-up scripts written in python. In such cases, when the job from a user's perspective "just doesn't continue", or "hangs", the user previously didn't have any possibility to handle the situation apart from just killing and resubmitting the job, hoping that this time the execution will succeed. By using a job execution monitor featuring a binary tracing module, the exact location in code or the exact data file the job stops on can be determined, allowing an analysis of the cause of the job failure.

4. C tracing library

4.1. Overview

To address the issue explained in the preceding section - the absence of monitoring data for the periods of the job's run inside the user algorithm - we developed an addition to JEM allowing to trace the progress of execution in compiled binaries (executables and/or shared libraries).

This library, named C-Tracer, logs every function- or method call and the corresponding return together with its code location, timestamp and associated data (function parameters, local scope variables). This data is then passed to JEM, where it gets inserted into the stream of other monitoring data transmitted by JEM to the UI machine (figure 3). In this way, the job's progress inside the user algorithm is no longer a blind spot for the job run analysis.

To use the C-Tracer, the target binaries must be prepared in a special way by the user, and additional configuration of the grid job is neccessary before submission. These are explained in section 4.6, after the functionality and implementation of the C-Tracer have been discussed in the following sections.

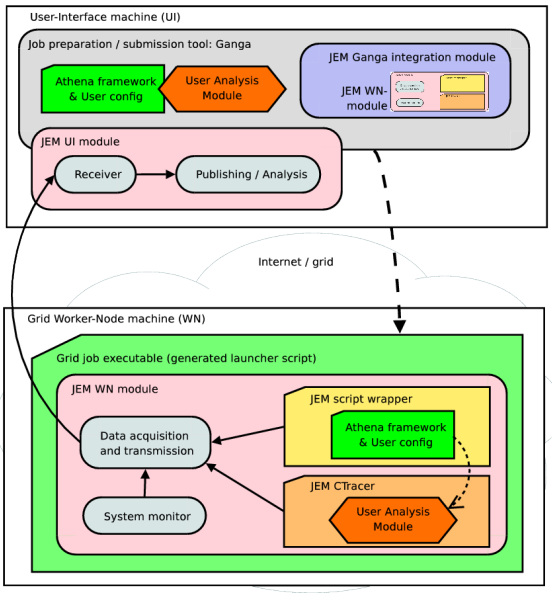

Figure 3: The C-Tracer is added to JEMs WN module and monitors the user analysis module (binary), transparently inserting the gathered monitoring data into JEMs publishing mechanism.

compile flag	description
`-g`	Include debug (DWARF) data in the object file.
`-finstrument-functions`	Add trace system call to each function call and return in the application.

Table 1: Compile flags needed to be passed to gcc when building the user application to prepare it for usage with the C-Tracer.

4.2. GNU C compiler trace hooks

Systems in the grid environment (CEs, WNs, UI machines, etc) almost exclusively run Linux as operating system (mostly Scientific Linux 5 [12], a Red Hat [13] derivate). Consequently, all binaries used in relation with Athena are compiled using the GNU C compiler [14] (GCC).

GCC offers the possibility at compile time (with the compile flag `-finstrument-functions`, see table 1) to instrument every function- or method call and every corresponding return in the built binary with an additional call to two callback functions (figure 4) which then separately must be implemented[15]. The implementation can be done directly within the built binary or, most usefully, in a library dynamically linked to the binary at run time. This is how our C-Tracer uses this feature of GCC.

To the callback functions, the addresses of the called and of the calling function are passed. The pointers to the stack frames of both functions can be determined using another of GCC's built-in functions (figure 5). These information are sufficient to resolve all wanted data about the called and calling functions (code location, arguments and local variables).

JEMs C-Tracer uses this callback system to log every[4] call/return and to publish it the same way as bash commands and python calls. To make the data useful, however, human-readable identifiers are needed, as well as further metadata like symbol types and line number information. These data can be gained by parsing the DWARF[16] debug data compiled into the binaries by GCC when the appropriate flag is appended to the compiler call.

```
void __cyg_profile_func_enter (void *func, void *callsite);
void __cyg_profile_func_exit (void *func, void *callsite);
```

Figure 4: Trace callback functions in binaries compiled by gcc.

```
void * __builtin_frame_address(unsigned int level);
```

Figure 5: Function used to resolve stack frame adresses.

[4] If the function/method called or returned from is contained in an include-list configurable for the C-Tracer to filter unwanted bloat data, e.g. calls inside system libraries like the STL

4.3. Resolving of symbol information

If an application is compiled with GCC set up to include debug symbols, DWARF data is written into the object file. This data contains information about every symbol (function/method, variable, etc) in the application's source code, including (but not limited to):

- The symbol's name
- Its location in source (file, line number)
- Its location in memory when the application is run (absolute address, offset from stack pointer or offset from structure start address for members)
- Its visibility and accessibility mode (for struct members), and its type

Furthermore, every type is listed in the debug data with its name, byte size and possibly linked types (e.g. "pointer to"-associations). Using this information, the monitoring data is augmented with the identifiers used in source code. This enables the user to follow the job's run through the algorithm. Furthermore, it is possible to interpret the algorithm's memory, displaying variable and parameter values and resolving whole structure contents. This allows the user to inspect the data the algorithm is working on, and possibly to discover logical errors caused by (or causing) invalid data in real time. One can call this operation mode of the C-Tracer as "remote debugging", although stopping and stepping-through the user algorithm is not (yet) supported.

4.4. Safe inspection of application memory

Because this data inspection takes place in the user application's working memory, one has to be aware of invalid or corrupt data and, most importantly, invalid or uninitialized pointers and references. To protect the C-Tracer - and with it the user application itself - from crashes caused by invalid memory access operations, special precautions must be taken.

To allow the C-Tracer to inspect user application memory and resolve pointers, a victim/watchdog system was implemented. Knowing that inspecting the user memory may very probably lead to an invalid access (segmentation violation), a dedicated victim thread that "is allowed to crash" is used to perform these accesses. The C-Tracer library, when loading and initalizing, starts a supervising "watchdog" thread, that in turn starts the "victim"-thread.

A piece of shared memory, protected from concurrent access by a mutex semaphore, is used to pass memory addresses to the victim and to return the inspected memory's contents back to the main thread of the C-Tracer monitoring the user application. The victim thread's main function just consists of an endless loop in that the victim waits for a memory address to try to access. If an address is written into the shared memory, the victim tries to copy the contents of the memory at this address into a buffer. If this copy attempt fails due to a protection fault (invalid memory access), a synchronous signal is sent to the victim thread by the Linux kernel, aborting this thread's execution without disturbing the other threads (main thread and watchdog).

The watchdog's task is to restart the victim-thread whenever it crashes. In this case, the watchdog informs the main thread about the failure. Instead of the value at the memory address referenced by the user pointer, a special value declaring the memory as invalid is then passed to JEM. Otherwise, if the memory content was successfully copied into the buffer, this content is published via JEM.

Memory inspected by this means includes the application's stack and heap, and pieces of code memory (static variables). By adding an offset to the base address of a structure (struct, class) in memory, also class members can be inspected. DWARF information, again, is used to resolve these offsets. The number of nested indirections inspected this way by the C-Tracer is limited; the exact number can be configured by the user. This way, endless loops are prevented when inspecting user data structures containing circular references. The structure of the victim/watchdog-system is displayed in figure 6.

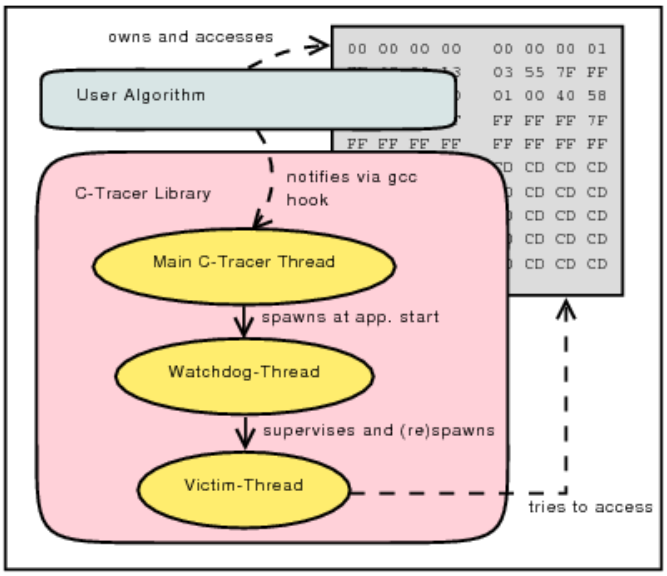

Figure 6: The Victim/Watchdog system's structure.

4.5. Summary of data gathered

Using the C-Tracer, as described in the preceding sections, rich monitoring information is gathered during the user algorithm's execution.

For each function (C-style function or class method, in the following just called "function") that is called, the function's location (file and line number), signature (name, return type and number, type and names of arguments - including an implicit this-pointer if existing) and the time of execution is recorded. If the C-Tracer is configured accordingly, for each argument, also the value is tried to be recorded.

The availability of those data is dependant on several factors. If the target binary has been compiled with optimizations, for example, not all variable data can be resolved. Even in non-optimized binaries, however, some variable content is not resolvable for the C-Tracer (variables assigned to registers instead of memory locations by the compiler). Those variables are denoted as "unresolvable" by the C-Tracer. If, otherwise, the value is available, it is read and inserted into the monitoring data.

From the caller (That is, the function containing the call to the monitored function), the location (again, file and line number), name as well as all local variables with their respective type, name and value are recorded (if available). The values of the local variables are resolved similarly to the function arguments as described above. For each return from a monitored function, the time of execution, the function's location, signature and its local variables are recorded similarly. For those events, as well, the availability of all variable contents is not guaranteed.

The data gained this way can be used to observe the parameters (sub)algorithms are called with, allowing to spot logical errors for example caused by specific input data. As the local variables of the calling method are recorded, the context of those calls is available, as well. This information also includes member variables and methods of user classes in C++ code.

4.6. Usage

In addition to the need to compile the user algorithm in a way providing the needed symbol information and callbacks for the C-Tracer by adding two special compile flags (see tab. 1) to the compiler call, the C-Tracer has to be configured before the user algorithm can be launched. This is done by means of several environment variables the C-Tracer reads and which have to be set beforehand by the user. To have the C-Tracer itself loaded, the dynamic library loader has to be advised to do so before the user application is launched. This, also, is configured just by setting an environment variable.

Since in most cases the user algorithm is used from inside of some launcher script, the environment variables can be set there. If the user job - and with it, JEM and the C-Tracer - is launched via a job submission tool like PanDA or Ganga, the configuration environment variables are set automatically according to the launcher-specific configuration of the job.

While the user job is executed on the foreign worker node, and if JEM has been set up to transmit data in real time, the monitoring data received by the UI-module of JEM is presented to the user in log-files and data files for further analysis; which files exactly are created depends on the local JEM configuration.

4.7. Data acquisition steps and performance impact

Using the C-Tracer adds a significant overhead to the user algorithm's run time. Measures are taken to allow for fine tuning the information gain against the performance impact of the C-Tracer. It is currently under investigation how much the performance of user algorithms degrades if the C-Tracer is used in different configurations and log levels. Already in its current version, the C-Tracer can be configured up to a certain degree, and very basic performance comparisons have already been taken. The performance impact is now explained by taking a detailled look into the processing of a single event (function call) in the C-Tracer.

Besides the natural possibility to skip the loading of the C-Tracer library completely by omitting the dynamic linker directive (`LD_PRELOAD`), it is possible to globally disable the C-Tracer for a user application run. This allowed us to measure the performance impact of just having the tracing library linked to the user application, without gathering any actual monitoring data. This also means no symbols are resolved, no user memory accesses are done and no data is passed to JEM. In this configuration, every call and every return inside the user algorithm causes a call of the tracing routines, with these routines then just immediately returning. This causes a performance degradation, for an empty user function, by a factor of 2.5, while a factor of roughly 3 could be expected for two additional function calls per original function call.

For actual user functions containing algorithm code, the factor is likely to improve, as the per-call overhead of the C-Tracer is constant.

If the C-Tracer was not globally disabled, the first action that is taken is to search the address of the called and calling functions in a whitelist (implemented as balanced red-black tree). If the address is not found there, the event is discarded by the C-Tracer. This mechanism was implemented to be able to aggressively filter unwanted events, such as all events occuring inside system- or thoroughly tested user-libraries, to focus the monitoring on specific user code and keep the performance costs as low as possible.

All other function calls and returns then are processed by looking up symbol-data stored in data structures optimized on fast by-address lookup. If the inspection of variable contents was enabled, this data also is used to discover the memory addresses that need to be accessed by the victim thread (plain memory adresses for heap-stored variables and offsets from the function's frame pointer for variables stored on the stack). The resolved data is written to memory and passed to JEMs data publishing process.

To give some absolute numbers (that are hardly comparable, due to being very test machine specific): Looking up all needed symbol data for a single function call event on average takes in

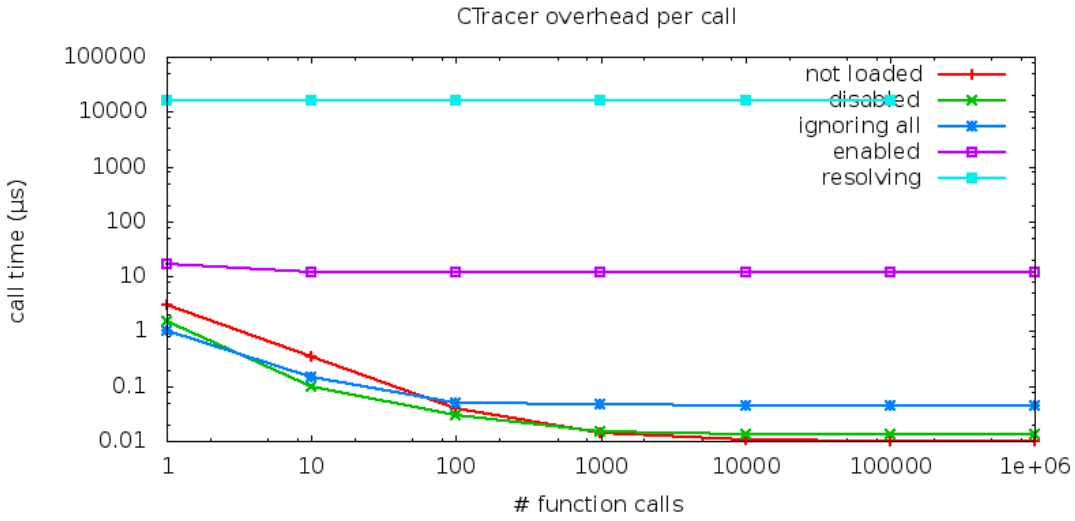

Figure 7: Per-Call overhead of the CTracer in several configurations.

the order of 50 ns. This delay is increased to up to 100 ns if variable values are to be resolved, and an extra 100 ns if a value couldn't be accessed, due to restarting of the victim thread. Similarly, each function return causes delay for monitoring data lookup and memory inspection.

The measurements on the test machine are shown in figure 7. One can see the basic overhead of loading and calling the C-Tracer library decreasing with increasing numbers of function calls. This is to be expected due to the (partial) static overhead nature of this basic functionality. Enabling the C-Tracer, but ignoring 100% of the events (thus causing one address lookup in the search tree per call/return) causes overhead in the order of one order of magniture for sufficiently large numbers of events.

Enabling the monitoring for all events raises the per-call overhead to three orders of magnitude; this, however, is an unlikely scenario, because in real application, only a small subset of the functions is expected to be white-listed. Resolving memory contents is a very expensive operation and will only be feasible for single key events, or on explicit user request (see the outlook in section 5).

Interestingly, the overhead of not loading the C-Tracer library at all is larger than the all-disabled overhead for low numbers of events. This effect is not yet fully understood, but may be a consequence of the Linux dynamic library loader unsuccessfully trying to find a suitable implementation for the trace-callbacks.

5. Conclusion and Outlook

We have presented a binary trace application that is integrated into a user-space real-time Grid job monitoring software. This binary tracer, or C-Tracer, adds progress- and user data-monitoring capabilities to the monitoring software, and allows for extension of the transparent portion of the job's execution into compiled binary modules.

By protecting the C-Tracer's memory accesses using a novel victim-watchdog-architecture, tracer crashes affecting user job efficiency and reliability are prevented.

Provision of the data gathered by the C-Tracer into the job monitor's data stream allows for easy correlation with other monitoring data like system metrics, watched log file contents and job control script progress events.

Finally, the application of the C-Tracer to a user job is a straight-forward and uncomplicated

operation if the monitoring software itself is already available and enabled. Integration into the two most common job control frameworks in ATLAS - PanDA and Ganga - has been implemented, allowing all ATLAS-based grid users to enable JEM by simply adding a configuration flag to their job submission.

Further development projects are being considered. Performance optimizations aside, several possible feature additions are worth to be considered for the C-Tracer. A user-defined rule base to narrow down the generated data to "interesting" events - by means of a "trigger"-like infrastructure - could increase the C-Tracer's applicability by a large amount.

Further, the possibility to control the C-Tracer's verbosity during the job's run time could be a helpful feature. Development is currently in progress to add a secure control channel from the user (or site admin, expert shifter, etc.) to the job's JEM instance; addition of the C-Tracer to this control channel is strongly being considered.

Such a control channel would allow the C-Tracer to provide in-depth "snapshot" information on demand to the user. This information could contain whole memory dumps, stack traces and call graphs, among other interesting options. By coupling these snapshot functionality to the aforementioned trigger-architecture, the C-Tracer would truly become a "remote debugger".

References

[1] Ian Foster and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[2] E. Laure et al., *Programming the Grid with gLite*, in *Computational Methods in Science and Technology*, 2006.

[3] The Atlas UK Collaboration, *The Athena software framework*
http://www.atlas.ac.uk/comp.html

[4] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, AIHENP conference, Laussanne, 1996.
http://root.cern.ch

[5] The PanDA Production and Distributed Analysis System.
https://twiki.cern.ch/twiki/bin/view/Atlas/PanDA
Accessed August 2010.

[6] K. Harrison et al., *Ganga: a Grid user interface for distributed analysis*, Proc. Fifth UK e-Science All-Hands Meeting, Nottingham, UK, 18th-21st September, Ed. Simon J. Cox (National e-Science Centre, 2006), pages 518-525.
http://ganga.web.cern.ch/ganga

[7] M. Ernst, P. Fuhrmann, T. Mkrtchyan, J. Bakken, I. Fisk, T. Perelmutov, and D. Petravick, *Managed data storage and data access services for data grids*, in *Computing in High Engergy and Nuclear Physics*, 2004 (A. Aimar, J. Harvey, and N. Knoors, eds.), (Geneva), CERN, 2005.

[8] D. Igdalov, *Entwicklung eines Systems zur Analyse und Überwachung der Verarbeitung von Rechenanforderungen im LHC Computing-Grid*, Diploma thesis, Fachhochschule Niederrhein, August 2005.

[9] A. Hammad, *Entwicklung eines Überwachungssystems für verteilte Prozesse im LHC-Computing Grid*, Master thesis, Fachhochschule Niederrhein, December 2005.

[10] T. München et al., *Job centric monitoring for ATLAS jobs in the LHC Computing Grid*, XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT) 2008, Ettore Majorana Foundation and Centre for Scientific Culture, Erice, Sicily, November 3-7, 2008

[11] D. Igdalov, A. Hammad, S. Borovac, M. Mechtel, T. München et.al., *JEM, The Job Execution Monitor.*
https://svn.grid.uni-wuppertal.de/trac/JEM

[12] Scientific Linux 5 Distribution.
http://www.scientificlinux.org

[13] RedHat Linux Distribution.
http://www.redhat.de

[14] GNU foundation. *The GNU C Compiler.*
http://gcc.gnu.org

[15] R. M. Stallman et al., *Using the GNU Compiler Collection*, GNU GCC 3.4.6 reference manual, May 2004.
http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc.pdf

[16] The DWARF Standards Committee, *The DWARF Debugging Format Standard*, January 2006.
http://dwarfstd.org