Leibniz-Rechenzentrum

der Bayerischen Akademie der Wissenschaften

Technical report

Performance Monitoring

- A Generic Approach -

Richard Patra, Matthias Brehm, Reinhold Bader
Ralf Ebner, Sascha Haupt

Dec 2006 **LRZ-Bericht 2006-06**

1 Introduction

There has always been a gap between the peak performance a system can theoretically deliver and the performance it delivers in user operation. While the peak performance is what hardware-vendors promote the application performance in user operation is what counts in the end. The application performance is directly related with the turnaround frequency the users achieve with their codes. To reflect this the selection of new supercomputers at LRZ is based on a benchmark that consists to a large extent of user application kernels. Not all applications that will run on a supercomputer can be included in such a benchmark nor are tests included in the benchmark on how an application mix performs on such a machine.

In order to ensure high quality of operation for supercomputers LRZ also monitors the performance during regular user operation. The meaning of the term “quality” is manifold here:

1.) quality from the system operations point of view: detection of bottlenecks in the system configuration and detection of malfunctions, e.g. of the batch queueing system
2.) quality w.r.t. the jobmix: detect applications that have an impact on the whole system and eventually degrade the performance other users can achieve
3.) quality w.r.t. a single application: find applications where the performance is low (such that one can try to optimize the respective applications)
4.) quality w.r.t. to a call for tenders for a new computer: allow selection of a representative set of benchmark programs highlighting different aspects of the hardware

To achieve the above goals the performance is periodically measured on a granularity of several minutes on each supercomputer at LRZ. These measurements are used for monitoring the current machine behaviour and are also evaluated in a statistical analysis.

In the past the performance data has been written into text files. While this approach has proven as stable and usable there exist several disadvantages:

1.) The large amount of information that is produced is difficult to oversee and to process: especially the format of the text files is not fixed and has changed inevitably multiple times in the past. This makes subsequent processing for analysis purposes tedious.
2.) Another problem with text files is that processing a subset of the data is difficult and often requires parsing large amounts of data to extract only a bit of relevant information.

This has led to the approach of storing all performance data in a database. However the database schema has to be implemented in such a way that it is flexible enough to keep the performance data of various system architectures and devices but also generic and structured enough such that the later processing can be implemented efficiently.

The following section 2 will give an overview of how processing of performance data was implemented in the past. Section 3 will give an overview of the database schema that we have developed so far and that is in use now. Section 4 will detail the data aquisition and measurement process. A short overview of data QA is given in section 5 and section 6 shows how the data produced is used currently.

2 Processing performance data in text files

In the past all performance data at LRZ was processed in text files. The approach in those days was to have a text file for each node of a supercomputer containing the performance data for that respective node. Whenever a new measurement for that node had been carried out the data was appended to the corresponding file as a new text record. These text files tend to become very long. Therefore a new file was begun each month or each time the machine was rebooted while the old file was archived. Having to collect data from various files makes the statistical analysis process complicated.

The structure of the records written into these text files was tailored to the architecture of the nodes. An example record for an SR8000 node (1 service processor SP, 8 instruction processors IP0-7) is shown in :

```
78:10:00      SP     IP0     IP1     IP2     IP3     IP4     IP5     IP6     IP7
usr(s)         0     274     299     299     298     298     299     299     298
   (us)     1276  825523   62323   49797  784306  939984   48441   26113  921034
sys(s)         2       0       0       0       0       0       0       0       0
   (us)   404853  404624   13361       0  273593  119768   10980   35052  190766
usage       0.80   91.74   99.69   99.68   99.69   99.69   99.69   99.69   99.70
inst      320230 113253M  48838M  48787M  48868M  48864M  48712M  48855M  43243M
CPI         1.43    0.91    2.29    2.29    2.29    2.29    2.30    2.29    2.58
LD/ST      91501  49092M  16501M  16488M  16512M  16511M  16469M  16507M  14535M
ITLB           4    1165      16      16      16      16      16      16      16
DTLB          16     193       1       1       1       1       1       1       1
Icache      2516  44051K  117341  115473  151570  137566  116878  132119  144102
Dcache       193  22175K  13476K  13188K  13370K  13823K  13248K  13258K  11541K
FU           683  81055M  40140M  40095M  40164M  40162M  40032M  40157M  35610M
fault        150       0       0       0       0       0       0       0       0
zero           0       0       0       0       0       0       0       0       0
react         36       0       0       0       0       0       0       0       0
pagein         0       0       0       0       0       0       0       0       0
COW            0       0       0       0       0       0       0       0       0
nswap          0       0       0       0       0       0       0       0       0
syscall       10      50       0       0       0       0       0       0       0
align          0    9580     339       0    7362    2551     256     903    5214
(7.5.1)[ 1664MB free, 54MB active, 1MB inacitve, 6470MB wire ]
send_packets                          30316
send_short_packets                    20748
send_bytes                       1303345263
send_time(us)                      10570507
recv_packets                          30452
recv_short_packets                    20898
recv_bytes                       1303324779
recv_time(us)                      10525476
barrier_counts                            0
barrier_wait_time(us)                     0
interrupts                             1350
packets_from_Y-XB                     15800
packets_to_Y-XB                       14344
packets from Z-XB                      7621
```

Figure 1: example performance data record for an SR8000 node

In the first half of the record the rows list the different counters measured, the columns show the different processors. A line containing memory information for the node in the middle of the record is followed by performance data from the network interface of the node.

All data is given in absolute values for the 5 minute measurement intervals. The data is not in floating point notation; the data structure is clearly tailored to the structure of an SR8000 node (processors, memory, network interface). These facts are especially problematic since for another machine the measurement interval is likely to be different, the format of the data can be different and the structure of the hardware is also very likely to be different. Therefore the programming effort for evaluating all the data of the different machines increases with the number of different machines to be evaluated.

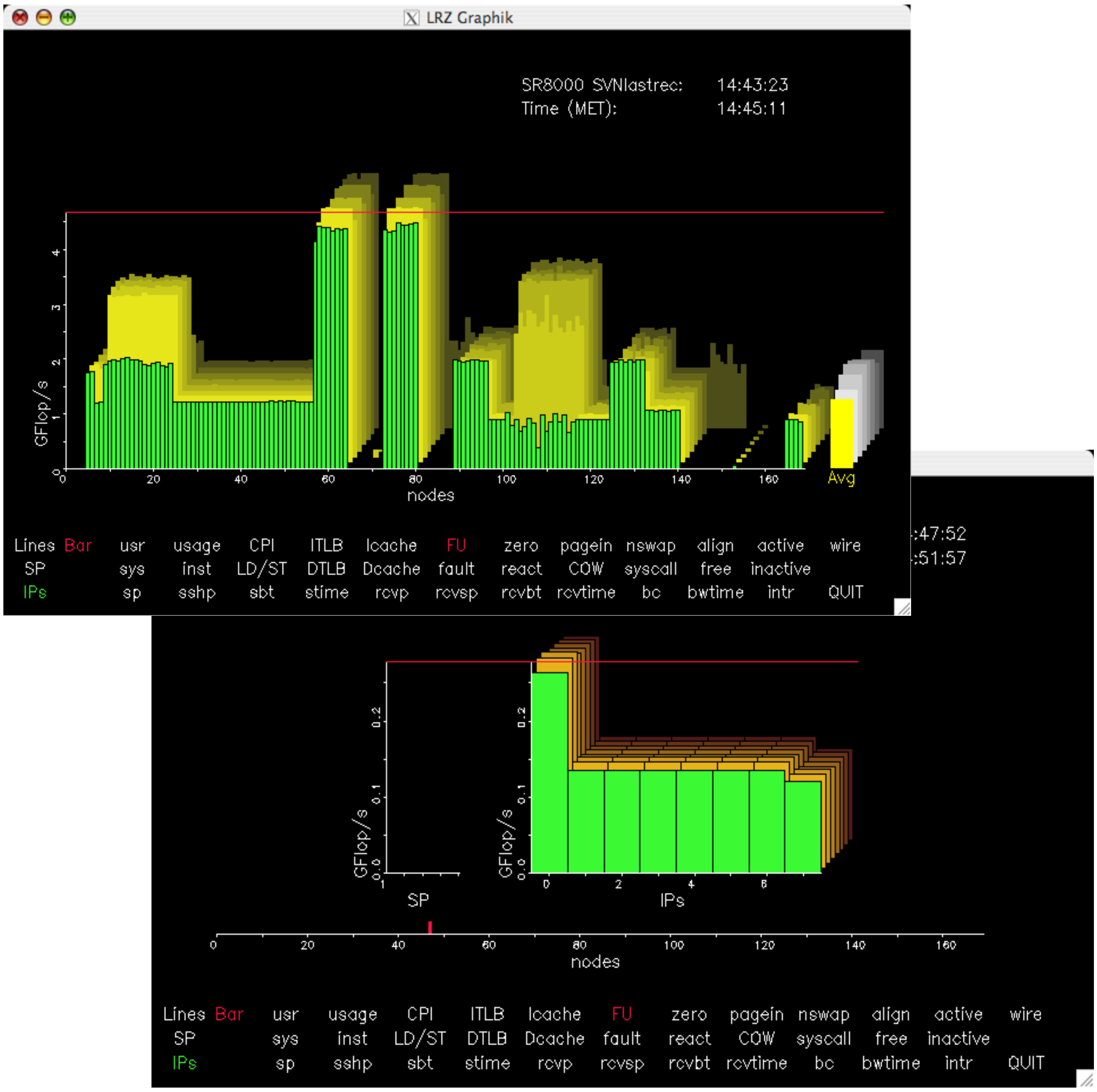

LRZ Graphik
SR8000 SVNlastrec: 14:43:23
Time (MET): 14:45:11
GFlop/s
nodes
Avg
Lines Bar usr usage CPI ITLB Icache FU zero pagein nswap align active wire
SP sys inst LD/ST DTLB Dcache fault react COW syscall free inactive
IPs sp sshp sbt stime rcvp rcvsp rcvbt rcvtime bc bwtime intr QUIT
:47:52
:51:57
GFlop/s
SP
IPs
nodes
Lines Bar usr usage CPI ITLB Icache FU zero pagein nswap align active wire
SP sys inst LD/ST DTLB Dcache fault react COW syscall free inactive
IPs sp sshp sbt stime rcvp rcvsp rcvbt rcvtime bc bwtime intr QUIT

Figure 2: tool showing the performance of the SR8000 (upper part: whole machine with 168 nodes, lower part: a single node)

The output of a tool that was written for monitoring of SR8000 system performance is shown in figure 2. The tool directly processes the records shown in Figure 1. It parses the last 8 records of each nodes performance file and displays the average for each of the 168 nodes of LRZs SR8000 graphically. The performance of the processors in a particular node can also be visualised as is shown in the lower part of figure 2 (drill down capability). The tool is also tailored to the particular hardware structure of the SR8000. Adaption to another machine requires some programming effort. It is necessary to parse a different record format and adapt the GUI.

The experience we have so far leads us to the following guidelines for performance data in the future:

- Data should be stored without prefix in the unit given (i.e. events, bytes, packets,...; Flop instead of Gflop): this will make comparisons of different architectures possible in an easier way. Data should be transformed to the target units as late as possible, e.g. a visualisation tool can choose the appropriate format (MFlop/GFlop/TFlop) automatically.
- Data should be stored in rates (events/s) because measurement intervals might be different among devices or change from time to time.
- Data should be stored as FLOAT (this precision is sufficient, because the order of magnitude is most important and no more than 4 significant digits will be required mostly; compared to DOUBLE it saves a considerable amount of space).
- Generic data structures which are not tailored to a specific device should be used.
- Tools should also be kept generic whenever possible.

3 CASE Study: Deriving Characteristics of Applications by using SR800 Hardware Performance Counters

Although programming models and languages appear to be converging, the computational workloads and communication patterns for scientific applications vary dramatically, depending in part on the nature of the problem the applications are solving.

Typical job accounting does not provide sufficient information about the characteristics of applications running on HPC systems. It is also impractical to use trace-based tools to monitor the behaviour of all applications on a system. Instrumented versions of the MPI library can be used to provide a detailed summary of the hardware performance counters and of the MPI calls, but this produces no immediate information during the run time of a job.

LRZ uses a more general approach to monitor all applications on its HPC systems. Samples of the most important hardware counters are taken from all nodes in 5 minute intervals, and are stored and are subsequently processed in a database. On the Hitachi SR8000, the following hardware counters and information from processor and communication network are used for the analysis (Tab.1):

User and system CPU time
Memory Usage
Number of Instructions (Memory/Floating Point/Integer)
Number of Load/Store instructions.
Number of Data/Instruction-TLB misses.
Number of Data/Instruction-Cache misses
Number of Floating instructions
Number of system-calls
Number of packets and bytes sent or received on a node
Barriers (time and number)
Data on pages, I/O etc.

Tab.1: HW counters on the SR8000

Users and support staff are automatically informed when an application falls below predefined values or combinations of parameters which usually indicates severe performance problems in the application itself. Also users can query the database for the performance of any job run in the present or past, and detect how the performance changed due to their modifications, or to get a qualitative global perception of the application behaviour. Furthermore, for the computing center itself such measurements contain important information for future procurements of new systems.

Here, we now want to present an analysis of data collected for all jobs on the SR8000 system over a period of four years. About 35,000 jobs have been processed, and more than 42 million samples have been taken during this period..

One of the most important metrics for the performance of applications is the computational intensity, i.e. the ratio of Load/Store operations (LdSt) and Floating Point operations (Flops).

In Fig. 1 we depict the number Ld/Ss per cycle vs. the number floating point operations per cycle on a single CPU basis. Every blue dot is a single measurement point. Due to the limited resolution many points fall upon each other. Therefore we have also plotted contours of the frequency of occurrence. The general regression analysis gives a slope of 1.7 Flops per Ld/St-operation (thin red line), but there are two separated islands (indicated by red ellipses) where most of the points cluster. For the upper region we get roughly 4 Flop/LdSt while in the lower region we have about 1.6 Flop/LdSt. Looking deeper into the user codes, we can assign the upper region to optimum library routines, e.g. for BLAS and FFT operations.

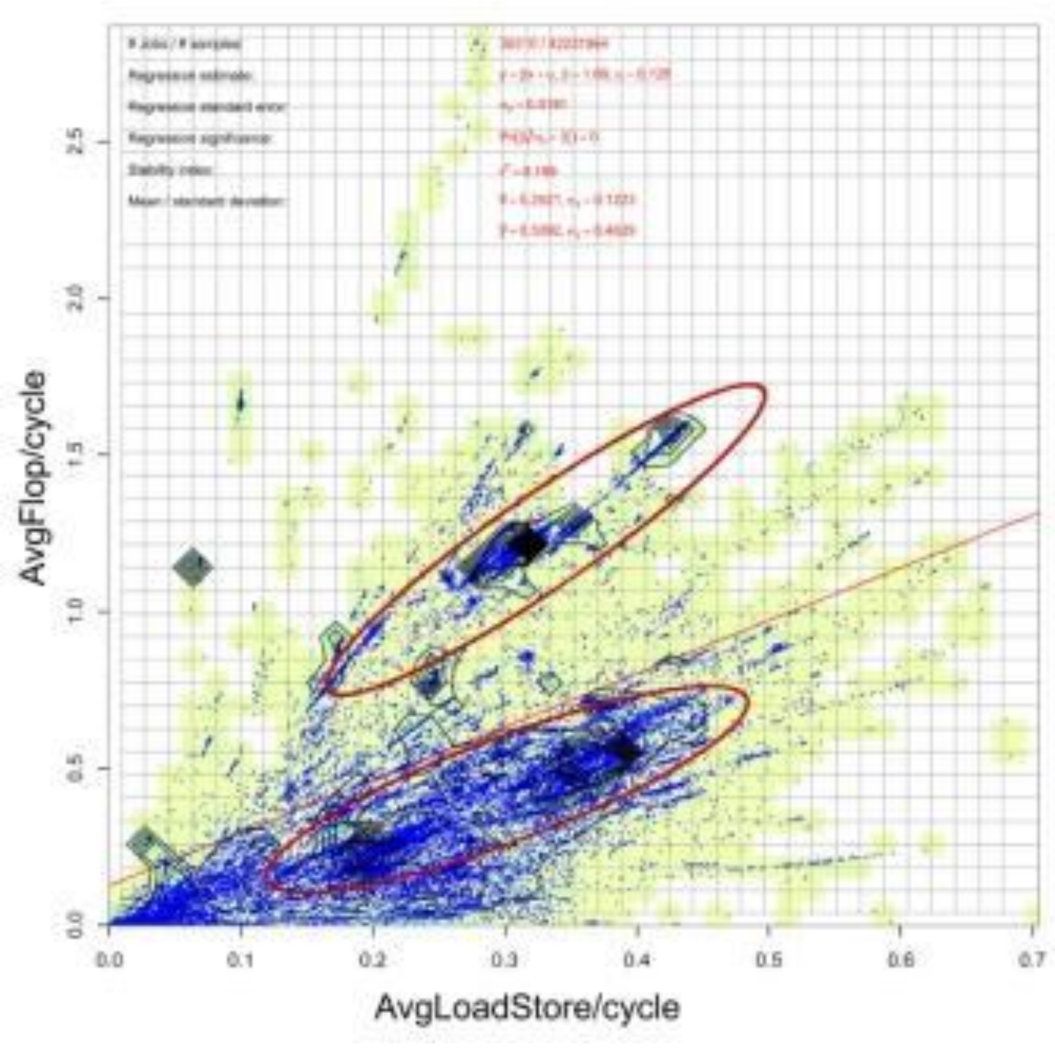


Fig 1.: LdSt vs. Flops for all application areas

If we limit our statistical samples to fluid dynamic applications (Fig. 2), we get a rather different picture, since we now see only one region with a slope of 1.6.

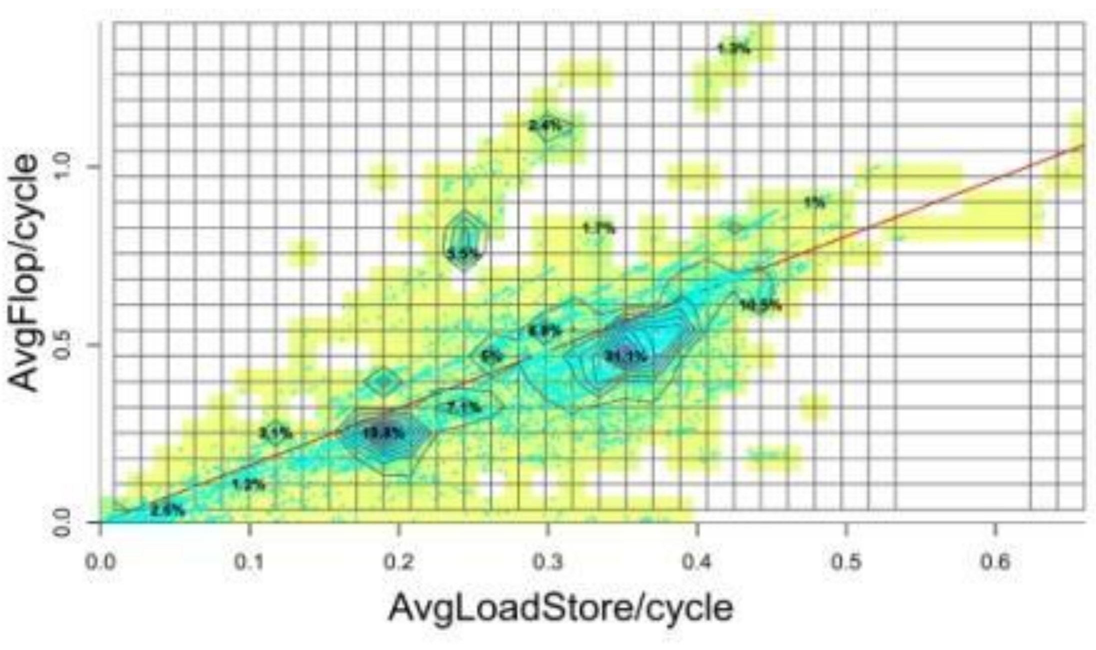


Fig 2.: LdSt vs. Flops for CFD jobs only. Slope is 1.6 Flop per Load/Store.

Typical ratios for other application fields are summarized in Table 1. It is clear that the performance of most jobs is limited by the available memory bandwidth; only for the optimized library usage the required bandwidth roughly matched the available one on the machine.

Application Field	Flops per Load/Store	Required Bandwidth Byte/Flop
All jobs	1.7	6.6
Optimized Libs	4.0	2.8
Geophysics	1.0	11.2
CFD	1.6	7.0
Chemistry	2.0	5.6
Solid State Physics	2.4	4.7

Table 1: Computational Intensity for various application areas
(roughly 70% of LoadStores are actually Quad-LoadStores)

The theoretical bandwidth of the SR8000 is 2.6 Byte/Flop.

In Fig. 3 we have illustrated the cumulative ratios of the **sustained** Flop/s vs. the sustained MPI traffic (Bytes/s) across the internal network interconnect, a crossbar switch. Having to decide what is an appropriate ratio for a well-balanced system, we came to the conclusion that 0.1 Byte/s per **sustained** Flop/s is sufficient for most applications (look for the arrow in Fig. 3) since more than 85% of the time the applications' requirements are below that value. Furthermore, requiring higher bandwidths would incur costs quite out of proportion to further gains on code performance. As the memory bandwidth is the limiting factor for most applications (see above), we compared this with the theoretical bandwidth of the SR8000-F1 (32GByte/s vs. 12GFlop/s per node). We conclude that an appropriate ratio between(unidirectional) network interconnect and memory bandwidth is of the order of 1:30.

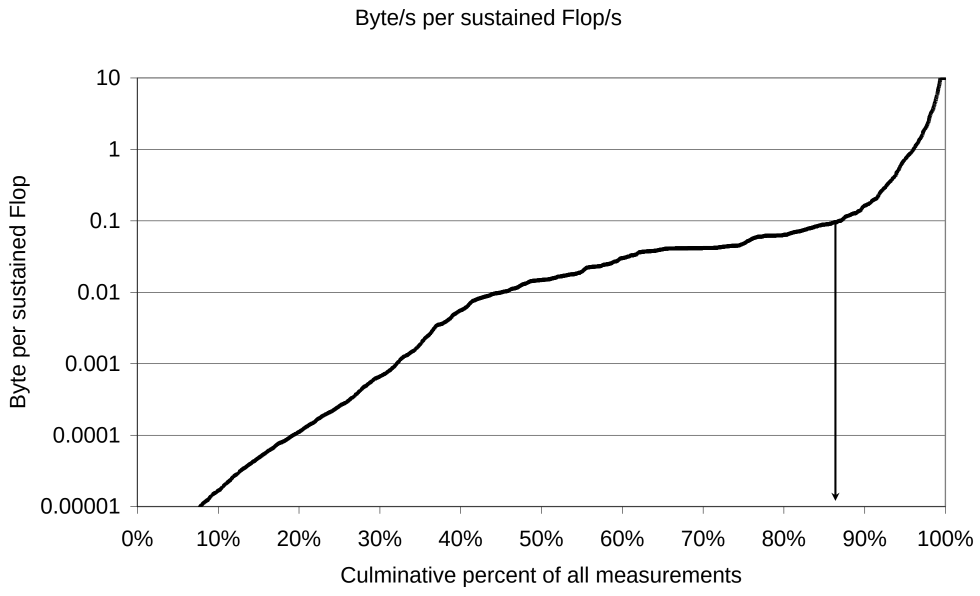


Fig. 3: Cumulative distribution of

MPI Bytes sent through the communication network per sustained Flop.

Looking at the ratio of Ld/St to DCache Misses (Fig. 4) we can conclude that the software pipelining as well as the preload and prefetch mechanisms of the SR8000 are very efficient in hiding latency. The majority of codes show only ratios of the order of a few cache misses per 1000 Ld/Stores. Here we note, that there is only a small 128 KByte L1 cache available on the Hitachi processor but enhanced prefetching mechanisms are implemented, which are obviously able to prefetch the data to the L1 cache ahead the issue of the corresponding Load instruction.

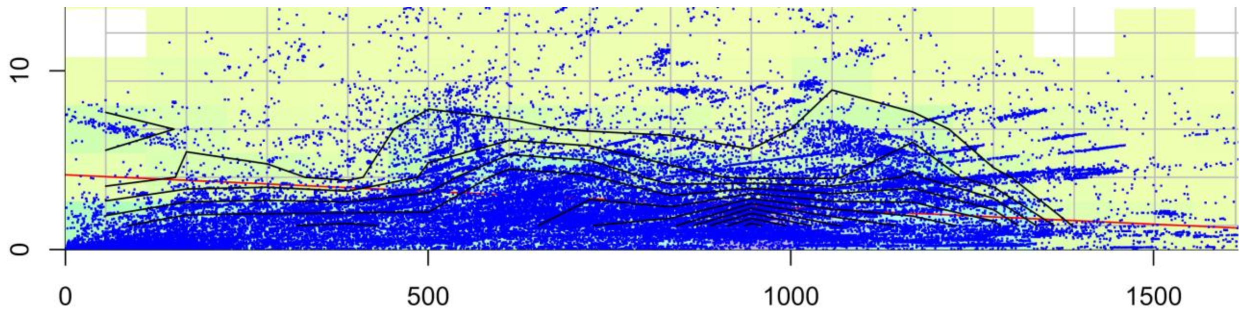


Fig 4: LdSt vs. CacheMisses per se. Contour lines show the relative frequency

The new Itanium based systems, which will be the successor machines for the SR8000, provide far more performance counters. Similar analyses will be performed on this follow-on architecture in the future.

4 Generic database schema

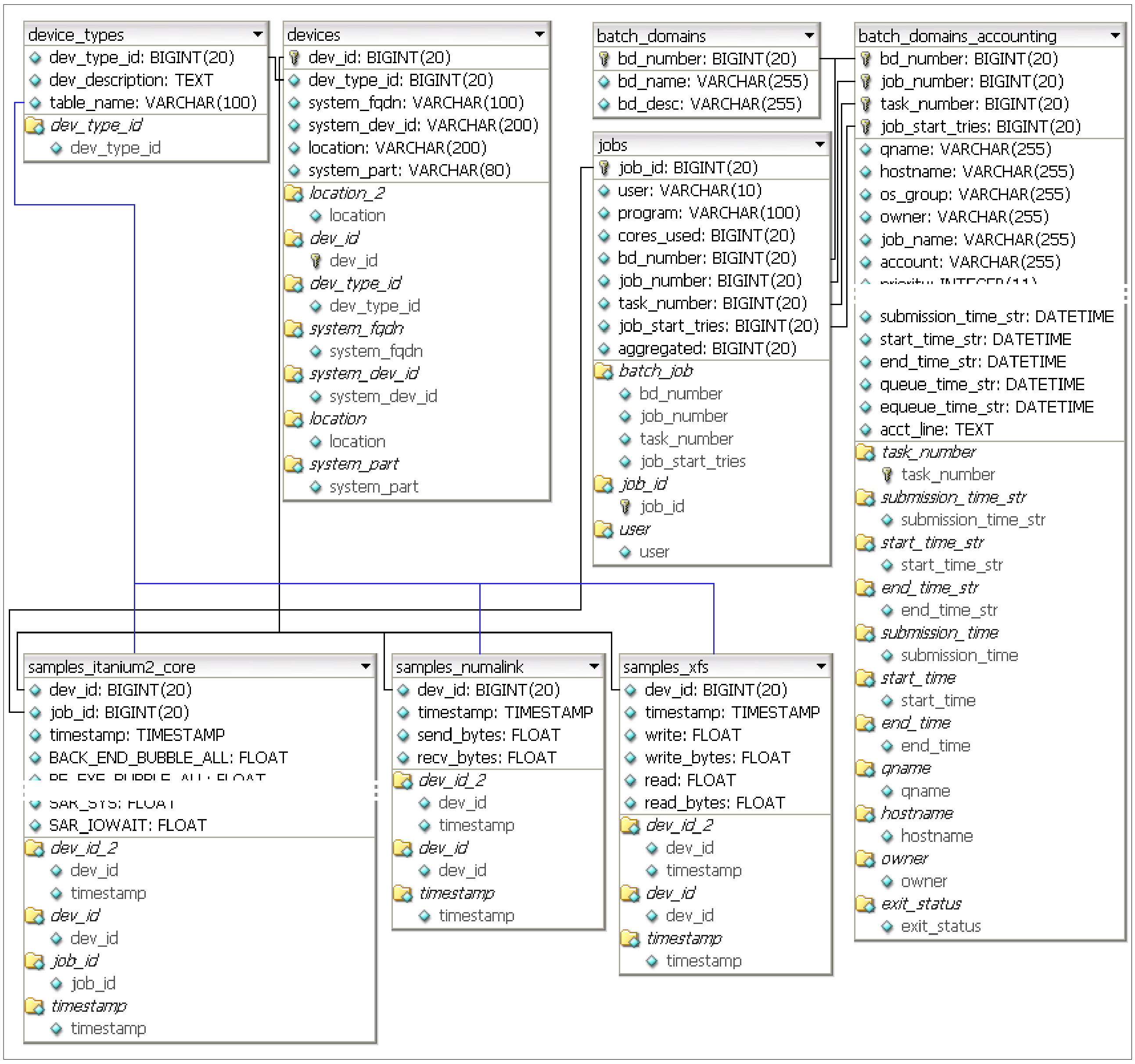
The fundamental idea with the database schema is that we will store measurements there together with some metainformation. More specifically, we need tables to store:

where the measurement was made, i.e. on which device

when it was made

and the measurement results of course.

The database schema is designed to keep performance data measurements from various devices of different type. The schema can be extended for arbitrary device types. Furthermore there should be information where a particular device is located in the hardware structure of the data center. On LRZs HPC systems most compute jobs are processed via a batch queueing system. On particular devices (processing cores) no more than a single batch job is running at a certain time. The information which particular batch job runs on a device at a certain time should be stored in the database too. The schema is shown in figure 3 and will be described table by table below. How the tables are filled will be subject to section 4.



device_types
- dev_type_id: BIGINT(20)
- dev_description: TEXT
- table_name: VARCHAR(100)
- *dev_type_id*
  - dev_type_id

devices
- dev_id: BIGINT(20)
- dev_type_id: BIGINT(20)
- system_fqdn: VARCHAR(100)
- system_dev_id: VARCHAR(200)
- location: VARCHAR(200)
- system_part: VARCHAR(80)
- *location_2*
  - location
- *dev_id*
  - dev_id
- *dev_type_id*
  - dev_type_id
- *system_fqdn*
  - system_fqdn
- *system_dev_id*
  - system_dev_id
- *location*
  - location
- *system_part*
  - system_part

batch_domains
- bd_number: BIGINT(20)
- bd_name: VARCHAR(255)
- bd_desc: VARCHAR(255)

jobs
- job_id: BIGINT(20)
- user: VARCHAR(10)
- program: VARCHAR(100)
- cores_used: BIGINT(20)
- bd_number: BIGINT(20)
- job_number: BIGINT(20)
- task_number: BIGINT(20)
- job_start_tries: BIGINT(20)
- aggregated: BIGINT(20)
- *batch_job*
  - bd_number
  - job_number
  - task_number
  - job_start_tries
- *job_id*
  - job_id
- *user*
  - user

batch_domains_accounting
- bd_number: BIGINT(20)
- job_number: BIGINT(20)
- task_number: BIGINT(20)
- job_start_tries: BIGINT(20)
- qname: VARCHAR(255)
- hostname: VARCHAR(255)
- os_group: VARCHAR(255)
- owner: VARCHAR(255)
- job_name: VARCHAR(255)
- account: VARCHAR(255)
- priority: INTEGER(11)
- submission_time_str: DATETIME
- start_time_str: DATETIME
- end_time_str: DATETIME
- queue_time_str: DATETIME
- equeue_time_str: DATETIME
- acct_line: TEXT
- *task_number*
  - task_number
- *submission_time_str*
  - submission_time_str
- *start_time_str*
  - start_time_str
- *end_time_str*
  - end_time_str
- *submission_time*
  - submission_time
- *start_time*
  - start_time
- *end_time*
  - end_time
- *qname*
  - qname
- *hostname*
  - hostname
- *owner*
  - owner
- *exit_status*
  - exit_status

samples_itanium2_core
- dev_id: BIGINT(20)
- job_id: BIGINT(20)
- timestamp: TIMESTAMP
- BACK_END_BUBBLE_ALL: FLOAT
- BE_EXE_BUBBLE_ALL: FLOAT
- SAR_SYS: FLOAT
- SAR_IOWAIT: FLOAT
- *dev_id_2*
  - dev_id
  - timestamp
- *dev_id*
  - dev_id
- *job_id*
  - job_id
- *timestamp*
  - timestamp

samples_numalink
- dev_id: BIGINT(20)
- timestamp: TIMESTAMP
- send_bytes: FLOAT
- recv_bytes: FLOAT
- *dev_id_2*
  - dev_id
  - timestamp
- *dev_id*
  - dev_id
- *timestamp*
  - timestamp

samples_xfs
- dev_id: BIGINT(20)
- timestamp: TIMESTAMP
- write: FLOAT
- write_bytes: FLOAT
- read: FLOAT
- read_bytes: FLOAT
- *dev_id_2*
  - dev_id
  - timestamp
- *dev_id*
  - dev_id
- *timestamp*
  - timestamp

Figure 3: database schema for performance data measurements and job accounting information

Table **device_types** holds an entry for each device type for which measurements exist in the database. Each device type is given an ID (*dev_type_id*), a short description (*dev_description*) and the name of the table that will store the measurements for that particular device (*table_name*). It makes sense to divide measurements by device type. Putting all measurements into a single table would be possible, but this would be a waste of disk resources and/or degrade database performance: if one would put measurements for all device types into a single table one would either

- create a table with fields consisting of a superset of fields needed for each particular device type
- or have only a single column for all measurements and another column where the type of the measurement stored is indicated

Both is a waste of resources: in the first case the number of empty cells is very large, in the second case the index tables will consume most of the space.

Each device to be monitored is listed in table **devices** and has an ID (*dev_id*) as a primary key there. It is assumed that all devices are located in a computer which has a hostname. A device is uniquely determined by

- its type (*dev_type_id*)
- the machine it belongs to (*system_fqdn*)
- and the internal number it has in the machine (*system_dev_id*; e.g. processornumber, numalink port number,...).

As an additional metainformation there is a field telling to which compute complex the device belongs to (*system_part*) and also a more detailed information about where the device is located (*location*). A few examples for entries in table devices can be found in the appendix.

The tables **samples_*** all contain measurements for particular device types. A measurement is uniquely determined by

- the ID of the device measured (*dev_id*)
- and time of the measurement (*timestamp*)

These two together form a unique index in each sample table. The remaining fields contain measurements for the particular device type. This is motivated by the idea that a device is measured at a certain time. However with this approach there is the slight inconsistency in that for some device types not all measurements can be made at the same time. E.g. for the Itanium2 there are not enough PMC registers on the processor to make all measurements simultaneously.

Some device types (processor cores) are devoted to a single batch job at a particular time. The ID of the respective job (*job_id)* can be considered as a kind of measurement too and also makes up one of the data fields for those device types for which a job ID can be assigned.

The remaining tables **batch_domains**, **jobs**, and **batch_domains_accounting** hold data from the batch queueing systems. Every batch domain operated has a unique ID (*bd_number*) in batch_domains together with a short name (*bd_name*) and a description (*bd_desc*). The table jobs contains all batch jobs that were observed (cf. section 4) during a measurement on an arbitrary device. Every batch job in jobs is determined by its unique index made up of

- the batch domain it runs in (*bd_number*)

- the number it has in there (*job_number*)
- the subtask number it has in there (*task_number*)
- and possibly the number of the try it is (*job_start_tries)*; for explanation: when a node crashes, the batch queueing system can restart jobs marked as restartable automatically, such a job does not have to be resubmitted and therefore is not given a new number. Alternatively we now handle the different tries all together as a single job in the database.

Table batch_domains_accounting has the unique index of table jobs as a primary key. It contains all batch jobs that have been run in one of the batch domains, not only the ones that were observed during a measurement (as table jobs does). batch_domains_accounting also includes a large number of additional fields that are generated from log entries of the batch queueing systems. There are 2 cavities with this approach, since different batch queueing system might log different information:

- it is mandatory that for all systems the same primary key can be generated
- we have not yet fully decided whether it makes sense to store the union or the cut of the different types of additional information in the log files of the queueing systems

However: using a single table for all different batch queueing systems forces one to store all information in a common format. This reduces programming overhead in the subsequent processing stages.

It has recently also been discussed to store the *batch_job_id* and *task_number* together as a single field in string format in the database. This would allow to keep up with the different formats for job Ids and task numbers in different batch queueing systems.

5 Data acquisition and measurement process

In the following section we will show how the tables described in the Section 3 are filled with information. What is described below can be considered as a concept and the implementation can be extended in the future. All the concepts given below have been tested already.

The tables described in section 3 can be divided into 2 classes: static/lookup tables that are filled once manually or by a manually started script and automatically filled tables, i.e. tables that are updated periodically in an automatic way.

5.1 static/lookup tables

Tables device_types and batch_domains are currently filled manually since they have a very limited number of entries and it is therefore not worth to have a script that generates these tables.

Table devices was at first generated by a script written for each new machine that simply generated the appropriate number of INSERT-statements for devices present in that machines. However all the devices to insert into table devices were hard coded in the script which is obviously – on the long run – not a good idea. Therefore now we use a single script called 'configure_devices'. This script can be run on each new machine. It should check out which devices are present in that machine and insert entries for those devices into the database.

'configure_devices' is a script that – in analogy to the well-known configure – checks out the machine type by type of devices in different sections of the script. In each section of the script the *system_fqdn* and *dev_type_id* is already known, so the script checks out which *system_dev_id*'s are present for a particular device. It then creates a new object for that device and calls a determine_location- and then a db_sync-routine for that device to determine the location field and to sync the data to the database

(For the devices we have a small object hierarchy of the following structure:

```
Device.pm <- Device_filesystem.pm
          <- Device_itanium2_core.pm
          <- Device_network.pm
          <- Device_numalink.pm
          <- Device_opteron_core.pm
...
```

where Device.pm contains common routines (like the db_sync), and the derived classes contain device specific routines, like e.g. determine_location).

[at LRZ all these scripts and classes are located in `/lrz/sys/sbin/perf`; this may change to `/lrz/sys/lrz_perf/sbin` in the future.]

5.2 automatically filled tables

Tables automatically filled are all the samples_*-tables and the tables jobs and batch_domains_accounting. For each of the samples_*-tables there exists a corresponding script in `/lrz/sys/sbin/perf` called `samples_*.pl` which will fill the respective table. Currently the scripts to be started on a particular machine are hardcoded in the cron table; another possibility would be to let cron run a short startup script determining the device types present in the machine and then fork off the appropriate scripts.

Scripts implemented currently:

script	***purpose***
`samples_itanium2_core.pl`	Fills table *samples_itanium2_core*
`samples_ia32_core.pl`	Fills table *samples_ia32_core*
`samples_em64t_core.pl`	Fills table *samples_em64t_core*
`samples__pcp.pl`	Fills tables *samples_filesystem, samples_network, samples_numalink, samples_xfs*; this is because all data can be generated conveniently with pcp; may need specific scripts for each table in the future.

Table 1: measurement scripts

A measurement script for a particular *device_type* works in the following way

1. Determine the timestamp (local time) and truncate it to full minutes (should hide slight delays in startup when machine is heavily loaded).
2. Determine
   - host name
   - which devices of the particular device_type are present in the local machine (i.e. check out their *system_dev_id*s)

   for being ready to identify the devices in the database and carrying out the measurements for them.
3. Determine the batch jobs eventually running on the devices and try to insert them into table jobs.
4. Carry out the measurements.
5. Try to deliver the results to the database: for each device measured look up its ID (*dev_id*), the timestamp is known from 1. and insert (*dev_id*, *timestamp*,<measurements>) into the database; the *job_id* can be considered as a special measurement, however it has to be looked up from table jobs first.
6. (not yet implemented: if result delivery to the database was not successful, the INSERT-statements could be written to disk and performed during one of the next runs of the script).

Notes:

- for 1. We lose 2 hours per year with local time: the one we skip in spring and the one we would have twice in autumn; alternative: use GMT
- for 1. The timestamp in the *samples_**-tables marks the start of the measurement periods; there is currently no end-timestamp since the length of the measurement periods is the same for all devices of a particular device type.
- for 4. The sampling period should cover the time from the start of the measurement script to the next start of the measurement script, but some devices, like e.g. Itanium2 do not allow to measure all performance numbers simultaneously, therefore one after the other has to be measured; this leads to an inaccuracy which however is in general not significant when carrying out a statistical analysis
- for 4. The actual measurement step usually relies on vendor-supplied tools like e.g. pfmon, pcp,... but for some device types it also uses utilities which were implemented at LRZ.

Part 3: aggregated tables

When a large number of devices is measured with a short measurement period the automatically filled tables tend to become quite huge. During later evaluation one is often interested in averages per compute job or in averages for a larger period of time only. Therefore several average tables are generated (per

HOUR, per DAY, per JOB). The corresponding tables are named like the tables where the data originates from, appended by the extensions `_AGG_BY_HOUR`, `_AGG_BY_DAY` and `_AGG_BY_JOB` respectively. The main advantage is that these tables have a much smaller size than the original ones and can therefore be processed much faster. Evaluation via ODBC/JDBC becomes possible then.

Furthermore the aggregates by job are denormalised such that OLAP tools like Mondrian and JPivot can be applied for generating statistical evaluations on the fly.

6 Data Q/A

6.1 statistical sampling

As noted in measurement step 4. of section 4 there can be device types for which not all measurements can be made simultaneously due to hardware restrictions. Therefore the measurements are taken one after the other. Thus one cannot be sure that a sample taken during a fraction of the measurement period can be considered as representative for the whole measurement period. This largely depends on the type of code that was running. Therefore we made an experiment with a fluid dynamics code which is running with constant timesteps. One timestep of the code takes roughly 16s wallclock. The experiment was carried out on the Itanium2 processor for which the restriction mentioned currently applies.

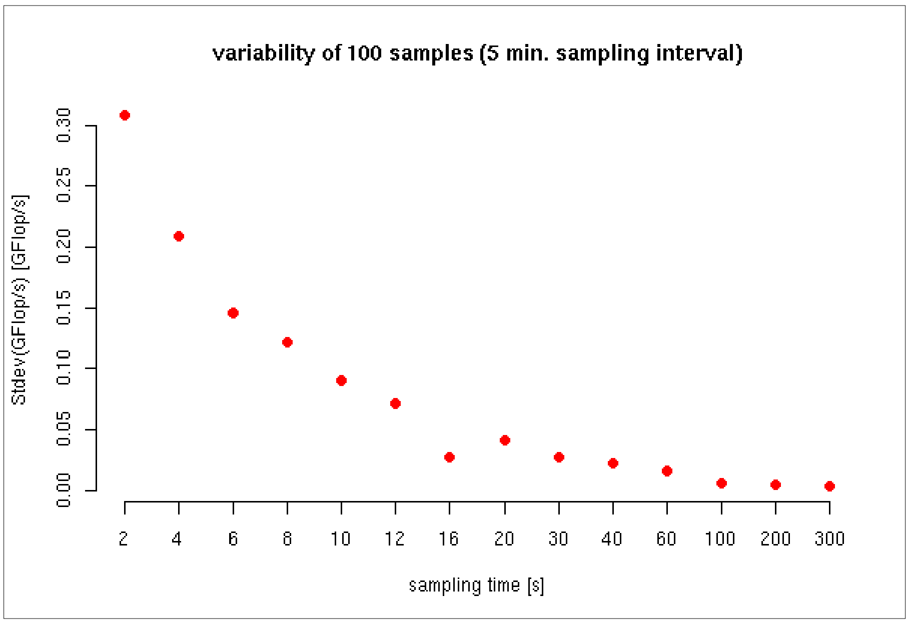


Figure 4: standard deviation versus sampling time (avg: 0.85Gflops) for an example code

The standarddeviation of the measurements goes down as the sampling time decreases which is as ex-

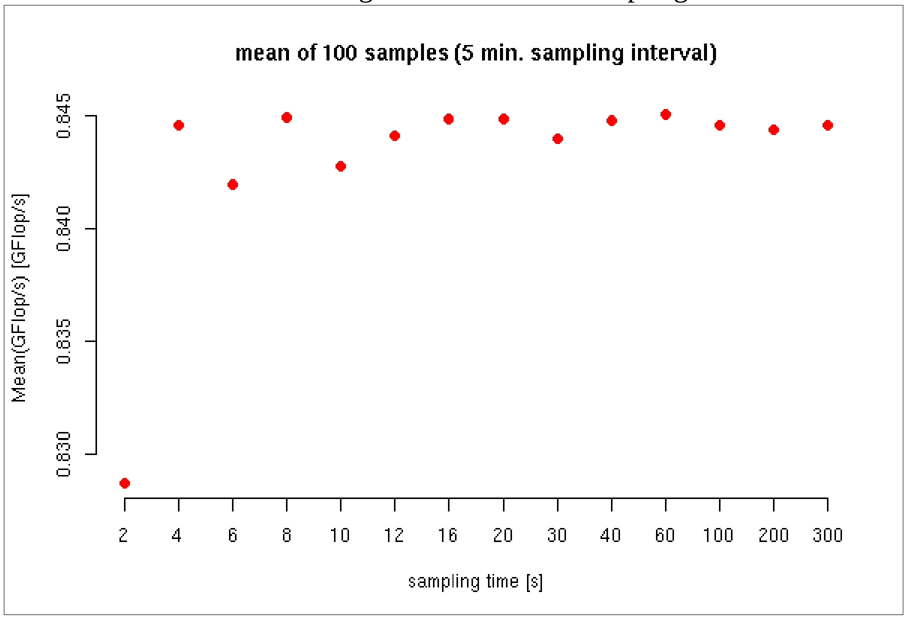


Figure 5: mean versus sampling time for an example code

pected. It can also be seen that the standarddeviation is especially small when the sampling time is close to the wallclock-length of a single timestep (~16s) (Figure 4).

Nevertheless the mean values calculated can be used starting from 2s sampling time (and probably even less) because the error seems to be on the order of 1 percent or even less (Figure 5).

6.2 automatic checks of the measurement process

- Data loss caused by system configuration changes:

It has turned out that the reliability of the measurement processes suffers from configuration changes (OS upgrade, upgrade of the kernel,...). As a consequence when such a malfunction event occurs data is lost until someone notices that the measurement process is not running properly and corrects the respective script or installs the respective measurement tool properly. In a large data center it is practically not possible to be aware of all configuration changes taking place. So the only possibility is to ensure that the data loss arising from failing measurement processes can be kept small. Therefore we have implemented a script which is automatically detecting missing data and allows us to fix such errors promptly.

In order to be able to detect malfunctions we have to provide the information to the check script whether and how much data from a particular device should be stored in the database during proper operation. Hence we have added 2 columns to table *devices*:

sampling_active	is the device being measured at present or is it deactivated
cron_entry	information about the length and placement of the measurement period

Table 2: additional column names for checks of the measurement process

The check script then checks for every device in table *devices* whether there are enough (i.e. corresponding to the cron_entry) measurements in the last hour. If it cannot find enough measurements it will send an email to the person running the database listing the devices that have incomplete or non-existing datasets in the database.

- Data loss caused by system crash:

It can also happen that a system crashes and therefore no data from this system is available in the database. In such a situation a monitoring machine detects that the data from the system is not available in time. The monitoring machine then writes the data for the system to the database setting all data fields to 0.0. So the performance monitoring tools can then also display that the system is down.

- Data loss caused by database maintenance:

It can also happen that the database server is in maintenance or that the network is not available. If this occurs then the data measured cannot be delivered to the database. Currently the data is lost then.

The database has proven as very stable in the test operation. Therefore we think that it is not worth to make the measurement process robust enough such that it can cope with the non-availability of the database and can deliver the data later.

However it has turned out that it is necessary to avoid long-running “SELECT” operations. This will block the delivery of data to the database and the database server will refuse to accept connections when the number of active connections has reached 200. Therefore all evaluations should be run on the replication machine which is read-only anyway.

7 Data intermediate aggregates

Many analysis operations need only very coarse grained access to the performance data. Therefore we have decided to build many data aggregates in advance. Currently the following aggregates are implemented for the Itanium2 cores only:

- samples_itanium2_core_AGG_BY_DAY
- samples_itanium2_core_AGG_BY_DAY_AND_SYSTEM_PART
- samples_itanium2_core_AGG_BY_HOUR
- samples_itanium2_core_AGG_BY_HOUR_AND_SYSTEM_PART
- samples_itanium2_core_AGG_BY_MINUTE_AND_SYSTEM_PART
- samples_itanium2_core_AGG_BY_JOB

It does not make sense to have aggregates by MINUTE (and not by SYSTEM_PART) since this is what samples_itanium2_core itself contains.

The aggregation process is driven from a monitoring machine. We have a single script for all the aggregates by time (the first 5 tables in the list above) and a special script for the aggregates by job (samples_itanium2_core_AGG_BY_JOB). All aggregates are updated once per day in the morning (cron job), with the exception of the aggregate by minute which is updated every 5 minutes. This is because the aggregate by minutes is also used for generating web performance graphics which should be up to date as much as possible.

It is also worth to note that table samples_itanium2_core_AGG_BY_JOB is denormalised, i.e. it contains a lot of information that could be extracted from other tables as well. While this is contradictory to good database modelling practise we have followed this approach here since the additional information does not increase the size of the table very much. The benefit we have is that the denormalised structure of this table facilitates the application of OLAP tools. An online evaluation using JPivot has been implemented and tested successfully.

The aggregate tables are used mainly for generating the webpages showing the status of the machines and for coarse grained analysis of jobs. Theoretically we could use the detailed data sets. Our experience has shown however that this is too much data for several evaluations per hour. We were able to reduce the time for generating the web performance graphics from tens of minutes to several seconds by using aggregate tables. It seems that there is no way around this at present.

For other devices currently no aggregates are being generated. This is because we have a limited amount of data for other devices and furthermore do not use the data very often. However this might change in the future and it would be worth thinking about implementing a generic script for calculating the aggregates for all device tables.

8 Usage of performance data

8.1 Web performance graphics

The performance data is visualised to inform users and the public about the performance achieved on LRZs supercomputers. Only the overall performance of the machines is shown; no data for particular devices and no information on particular jobs is available via the web-frontend at present.

The graphics on the webpages can also support operators when they want to check if the batch-queueing-system works well or whether the machine runs out of jobs.

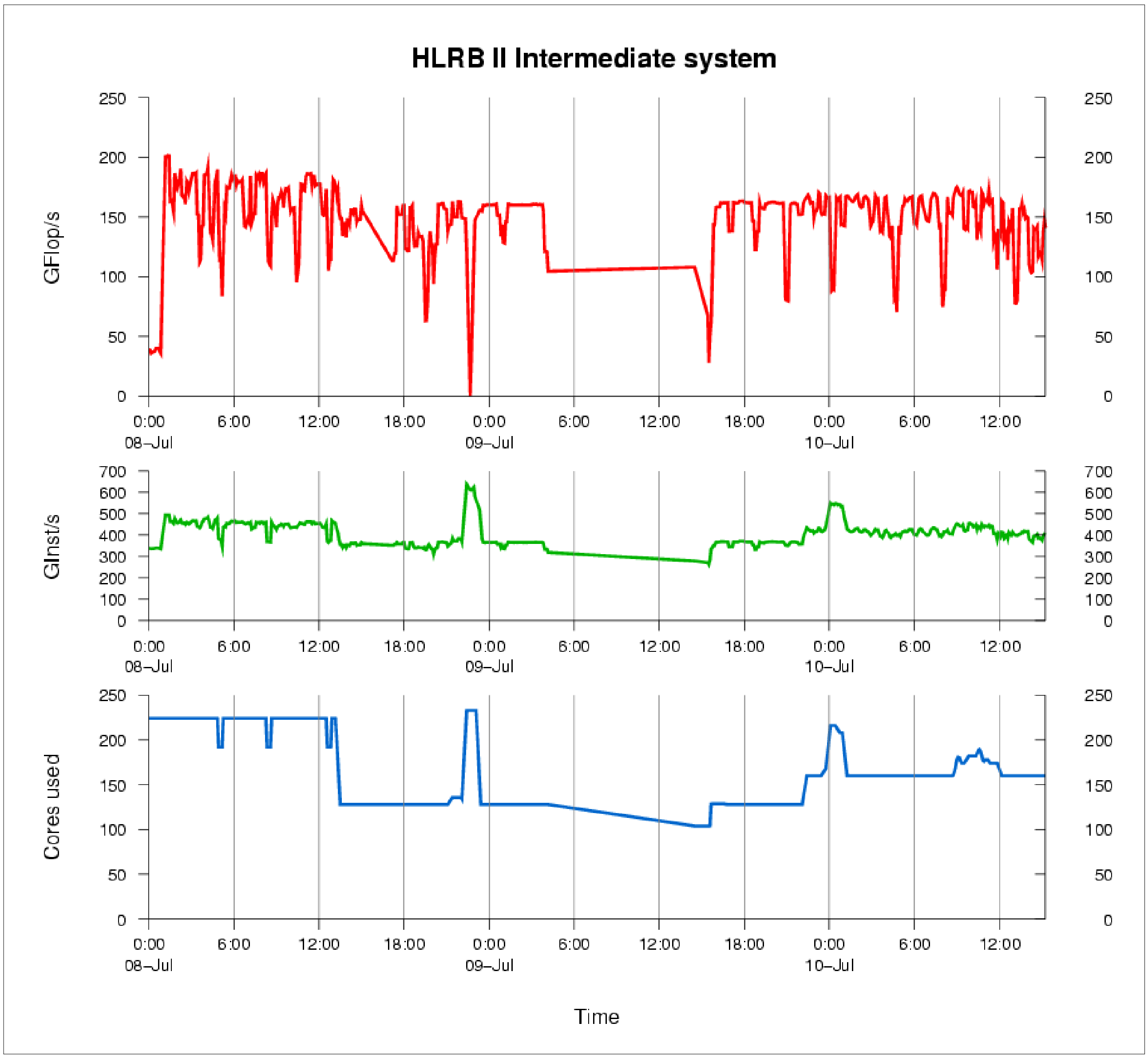


Figure 6: sample performance web page for HLRB II

8.2 Command line tool

A command-line utility has been implemented which allows the user to query the database for the average performance per processor of her/his own jobs. Sample output is shown in figure 7.

```
a2832bf@lx64i69:~> jobperf -runtime
+--------+------------------+------------------+-----------+
| JOB-ID | MFLOPS           | MIPS             | RUNTIME   |
+--------+------------------+------------------+-----------+
| 181318 |     518.68369130 |    2513.64834319 |  09:20:00 |
| 181663 |     528.09093417 |    2596.14109320 |  08:40:00 |
| 182539 |     506.33135391 |    2377.92655523 |  38:30:00 |
| 182540 |     505.25441836 |    2362.62428596 |  38:30:00 |
| 192277 |       5.49004612 |     440.48534581 |  00:00:00 |
| 192415 |       4.63990671 |     424.78949749 |  00:00:00 |
| 192583 |     488.12449644 |    2394.28966731 |  29:20:00 |
| 196553 |     340.65971743 |    1727.52697333 |  03:40:00 |
| 197412 |     485.78094145 |    2397.27543854 |  29:45:00 |
| 198139 |     399.50124711 |    2272.98122201 |  24:50:00 |
| 198420 |     101.61976419 |    1976.98793988 |  14:25:00 |
| 199547 |       0.03951436 |    1695.41428651 |  00:00:00 |
| 202793 |       4.66575669 |    2148.73388100 |  39:50:00 |
| 202936 |       0.12693860 |    1744.26754404 |  00:10:00 |
+--------+------------------+------------------+-----------+
a2832bf@lx64i69:~>
```

Figure 7: sample output of the command line tool jobperf

All available options are given with the `-help` commandline-switch. The functions of the command-line tool might be extended in the future to give hints on where the performance bottlenecks are located. For fine-grained performance analysis and performance optimization other more suitable tools will be needed in most cases.

It should be noted that the number of samples taken for a job depends on its runtime. For very short job-runtimes - a few hours or even less - the values calculated might not be representative for the overall performance. This is because the standarddeviation decreases with the number of samples taken.

8.3 Operator tool

As a feasibility prototype for the various tools for system operators we have implemented a generic monitoring tool that is able to handle different machines in a single user interface. The tool reads information about the distribution of devices in the datacenter from the database and then builds up a device tree on the left column. A subset of the tree can be selected by marking the respective node. The granularity level in the tree up to which performance data will be summed up, i.e. the granularity of the graphs on the right can be adapted to the users needs. This allows for drill down to a single node or even a single device if necessary. The counters which should be displayed can also be selected.

This tool can currently only handle Itanium2 processors but could be extended such that it can handle arbitrary device types.

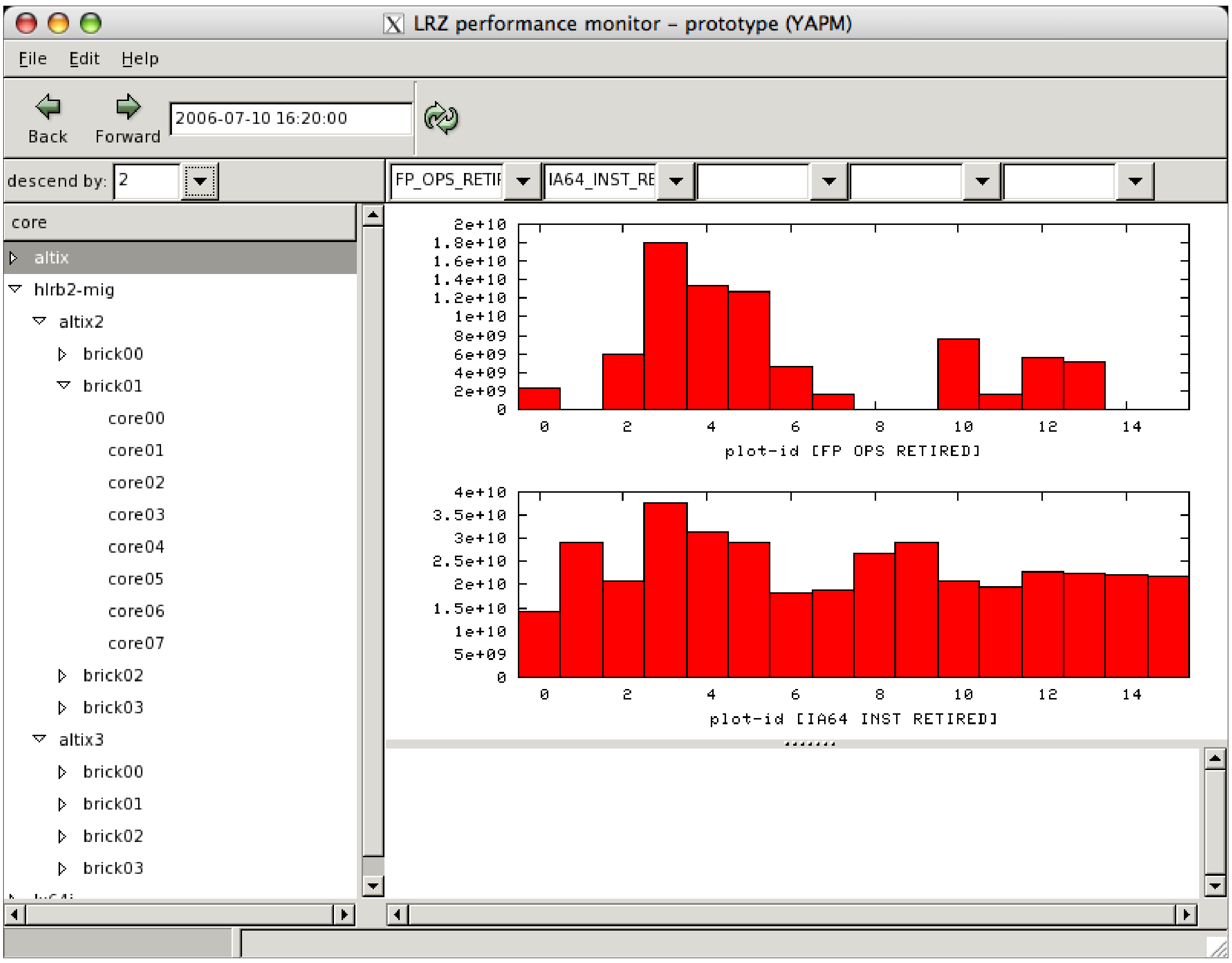


Figure 8: prototype for a generic performance monitoring tool

8.4 Statistical analysis

It is possible to access the MySQL-Database via JDBC/ODBC. This allows easy import of the data to office products like OpenOffice/MS Office or statistical analysis tools like R. It should however be noted that it is important to process larger datasets right inside the database such that the datasets which are imported are not to huge. Otherwise processing them with these tools is practically not possible.

The example diagram in figure 8 shows an evaluation of the average GFlop/s versus the average of L3 Misses/s per core. It should be noted that on machine hlrb2i twice the performance of the other machines was observed. The reason for this seems to be that there is only one core per memory subsystem interface on this machines while the other ones have 2 or even 4.

A cumulative performance distribution of LRZs Altix 3700 is shown in figure 9. It can be seen that the performance of the jobs shown is very evenly distributed. The curve ideally would increase in steps, each step representing an application class. However these steps are a bit smoothed since the machine is run as a single system image so jobs can gradually influence each other. The average performance of all jobs is well above 1GFlops/s per core which corresponds with approx. 15% of the peak performance. A more detailed analysis of the operational characteristics will follow and will therefore not be further extended in this section.

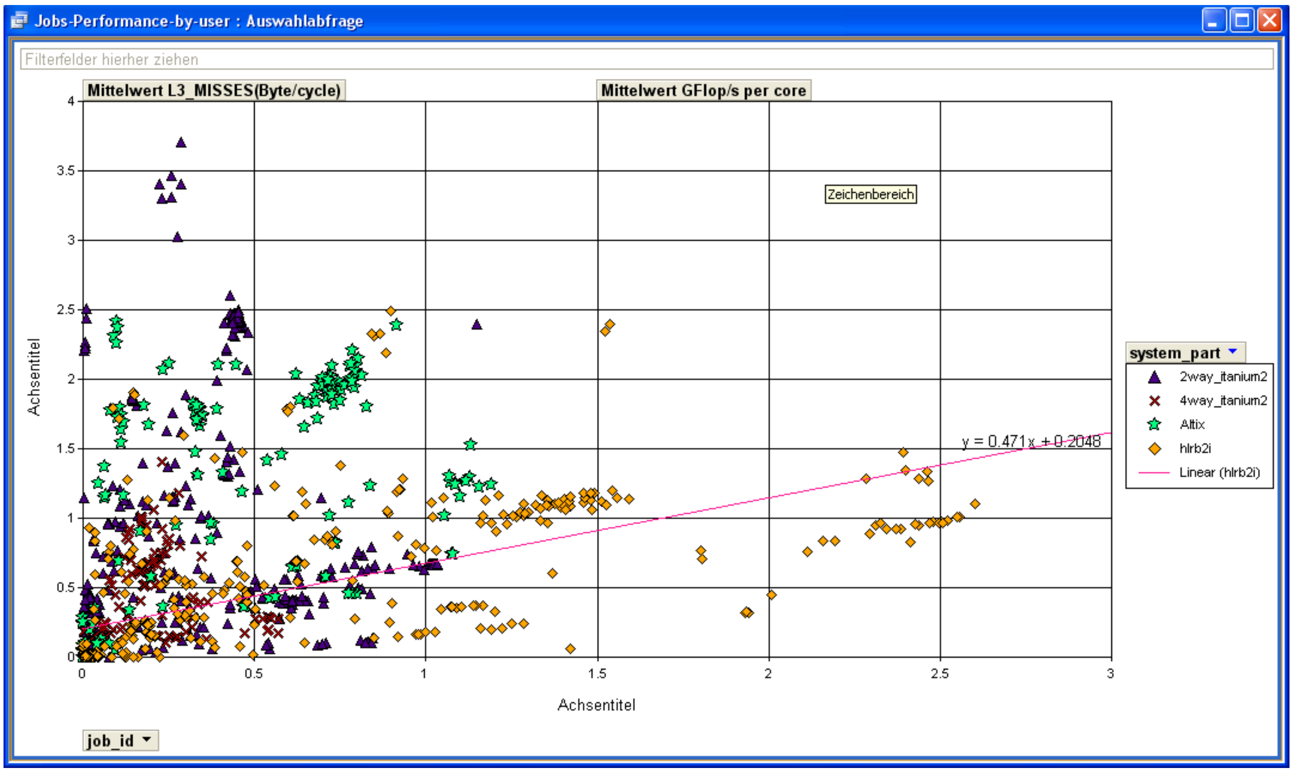


Figure 9: statistical analysis: flops/core versus L3 misses/core

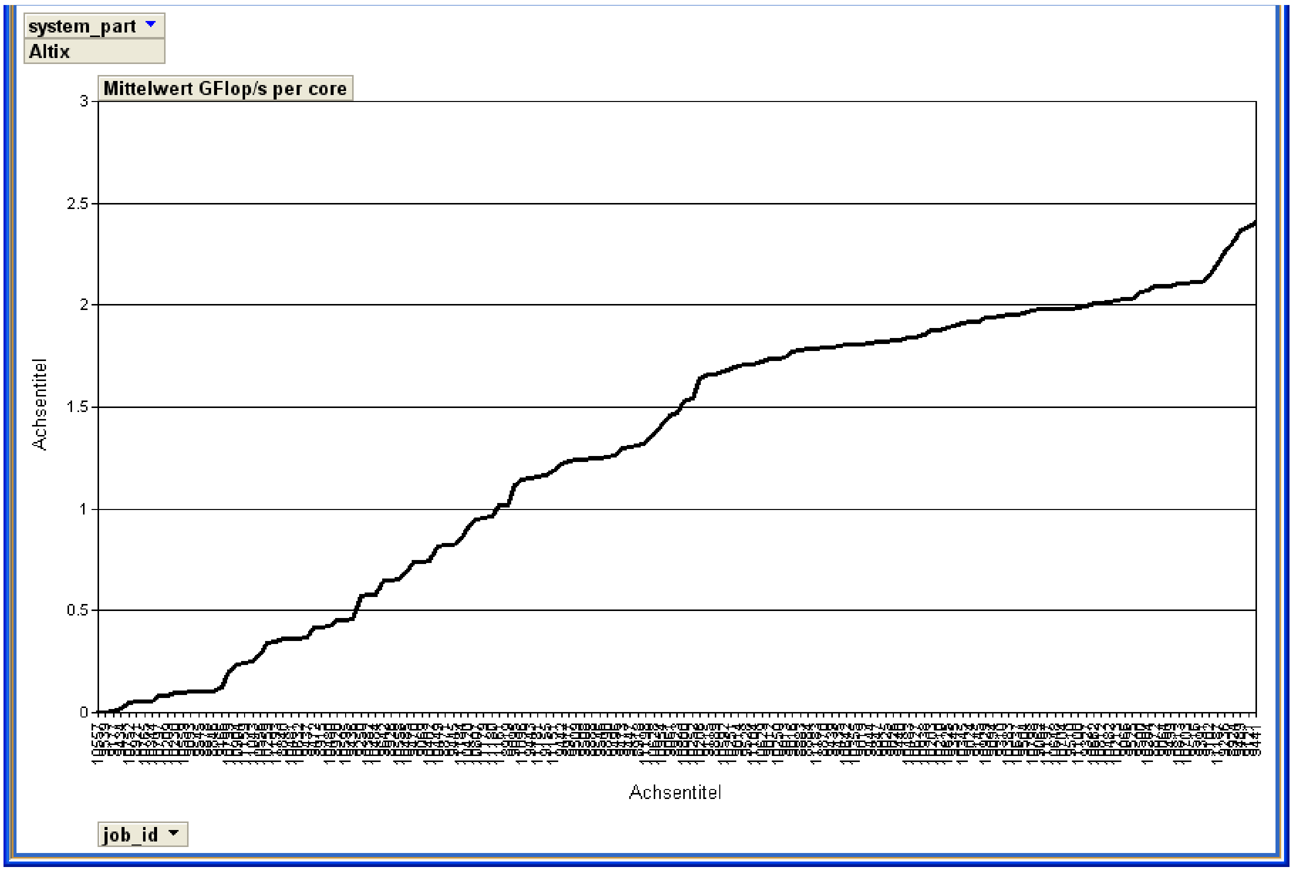


Figure 10: cumulative performance distribution for LRZ's Altix 3700

9 Case Studies: The Inner State of a Supercomputer: Getting Insight from Performance Counters

9.1 Motivation

The ranking of machine power is still based on peak performance or benchmarks but not on the actually delivered (floating point) operations over a given timescale in every day operation. However, computational workloads and communication patterns for scientific applications vary dramatically, depending in part on the nature of the problem the applications are solving. Recent works show that the characteristics of scientific applications differ significantly and the practical use of ranking or predicting system performance via single metrics and benchmarks such as High Performance LINPACK, STREAM or the HPC Challenge Benchmarks is quite limited. Only if enough information about the target applications is acquired, some simplified metrics may be combined and weighted appropriately to predict performance with reasonable accuracy. As hardware counters are ubiquitously available in modern processors, we argue that monitoring all applications in a system is an adequate way to get enough information on the system and finally achieve a good understanding of the potential of a given architecture.

In our study we monitored all applications on one of our HPC systems, an Altix 3700 Bx2 with 128 processors (see Table 1).

Processors	128 x Itanium2 Madison 9M
Clock	1.6 Ghz
Peak per proc.	6.4 Gflop/s (4 FP Ops per cycle)
L3 Cache	6 MB
L3 Cache Line Size	128 Byte
Bandwidth to L3	32 Gbyte/s
Bandwidth to memory, shared by two processors	6.4 Gbyte/s (4Byte/cycle)

Table 1: Characteristics of the Altix 37000 Bx2

Samples of the most important hardware counters are taken from all processors in 5 minute intervals and are stored and subsequently processed in a database. The sampling time for each measurement was 1 second. Hence, more than 120.000 “fine grained” measurements are taken in a two month interval. We must stress the point that we measured the every-day performance of a system including badly optimized programms, test runs etc. Nevertheless the results give us deep insight into the inner state of the system.

9.2 Average Performance

The average values of some of the most important counters and their maximum increment per cycle are given in Table 2. Ratios between the counters are given in Table 3.

	Max Incr.	---------- Mesurements ----------		
	per cycle	per cycle	% of Peak	per second
Instructions retired	6	1.938	32.3%	3.10E+09
Nops retired	6	0.666	11.1%	1.07E+09
Useful Instructions	6	1.272	21.2%	2.03E+09
Floating Point Operations	4	0.617	15.4%	9.86E+08
Stalled Cycles	1	0.545	54.5%	8.72E+08
Back_End_Bubbles	1	0.545	54.5%	8.72E+08
Loads Retired	4	0.250	6.3%	4.00E+08
Stores Retired	2	0.072	1.8%	1.15E+08
Loads+Stores, 16 byte assumed, Bytes	48	3.866	12.1%	6.19E+09
Loads+Stores, 8 byte assumed, Bytes	24	1.933	6.0%	3.09E+09
L2_References	4	0.274	6.9%	4.39E+08
L2_Misses	0.16	0.011	6.8%	1.70E+07
L2_Misses , Bytes	20	1.363	6.8%	2.18E+09
L3_References	0.16	0.014	8.9%	2.22E+07
L3_Misses	0.03	0.004	12.2%	6.08E+06
L3_MISSES, Bytes	4	0.487	12.2%	7.78E+08
w.r.t. second processor	2		24.4%	

Table 1: Performance counters, overall average

Ratio	
Useful Instructions / Unstalled Cycles	2.79
Useful Instruction / (Loads+Stores)	3.95
FP Inst / Inst retired	0.32
FP Inst / Useful Inst	0.48
FP Inst / (Loads+Stores)	3.95
FP Inst / (L2 Misses)	57.91
FP Inst / Byte (L2_Misses)	0.45
FP Inst / (L3 Misses)	162.21
FP Inst / Byte (L3_Misses)	1.27
L3_References / L3_Misses	3.65
L2_References / L2_Misses	25.75

Table 3: Ratio of counters

One of the key architectural features of the Itanium2 processor is to execute multiple instructions per clock. The burden for this explicit parallelism is put onto the compiler which encodes multiple operations for multiple functional units in every instruction. Each of the multiple operation instructions is called a bundle. Three instructions fit into a bundle, and two bundles can be executed simultaneously in each cycle. If slots can not be filled with “useful” instructions, due to dispersal constraints, NOPS are inserted. If the processor would not be stalled, six instructions per cycle could be delivered out of which two could be floating point instructions. Taking a fused multiply-add operation into account, four floating point operations per cycle could be delivered.

Our measurements show that due to the EPIC system architecture approximately two instructions are retired per cycle, but this is only one third of the maximum number. 11.4 % of all instructions are NOPS, showing that there is still room left for more instruction parallelism but the compilers, algorithms or the resource requirements are not suited to deliver this.

The “useful instructions” delivered can be computed by subtracting the NOPS from the retired instructions, leaving only 21% of maximum. On average the system runs with approx. 1 GFlop/s per processor, or with 15% of its peak floating point performance, a value which is very good compared to results of former RISC systems which delivered 8-10% of peak. The reason for this relatively high value is the good exploitation of the L2 and L3 caches. The large on-die L2 and L3 caches provide a significant performance potential.

The processor requires functional unit stalls to assure that results are computed correctly. The Itanium2 back-end has five counters for the various types of stalls in the processor backend. Each counter is associated with a given stage in execution pipeline. The Back_End_Bubbles accumulate the cycles where the instructions pipeline stalls for any reason. We must realize the fact that 54% of all cycles are stalled.

9.3 Memory Hierarchy

With the given cache line sizes of the processor we can calculate the consumed bandwidth to the next level in the memory hierarchy, especially the data rate between memory and L3 cache. For each byte that is transferred between the memory and the L3 cache, 1.27 floating point operations are performed. Unfortunately the measured load and store counters provide no direct way for an interpretation as bandwidth achieved between L2 or L1 cache and the registers since there may be single or paired load instructions of various sizes. As an estimate we can assign 8 Byte or 16 Byte to each load/store instruction and get a rough picture for what is happening in the memory hierarchy (see Fig. 1). At least 75% of all Loads and Stores can be satisfied from the high levels of the memory hierarchy.

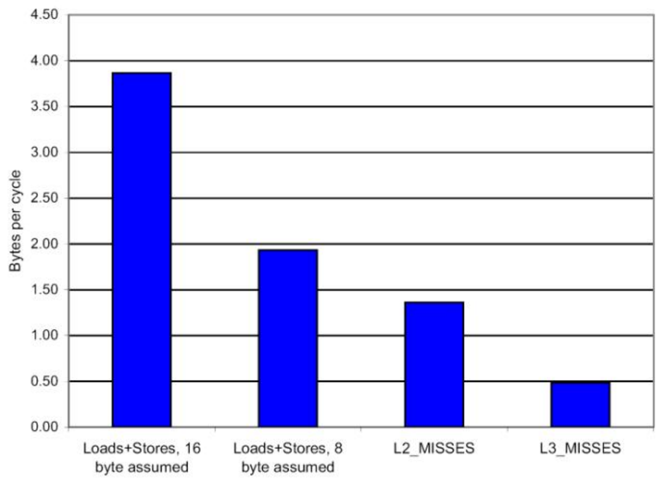


Fig.1: Transfer Rates in the Memory Hierarchy

The biggest surprise in our results was to find that on average the bandwidth to memory (expressed by L3 misses in byte per cycle) is not saturated and only 12.2% of it are used (24.4% if we consider that two processors share the same memory channel). In Fig.2 we plotted the floating point operations against the L3 misses of all samples. This kind of figures from the counters act like X-ray pictures which show us the backbones of the system. It is obvious that only high floating point performance has been achieved when the L3 misses have been less than 0.6 byte/cycle. A second pattern is the linear increase of performance with L3 Misses in the lower part of the figure. Approximately 75% of all samples fall below the inclined line. From benchmarks with known counter profiles one can deduce that this is the region where typical loops tend to reside.

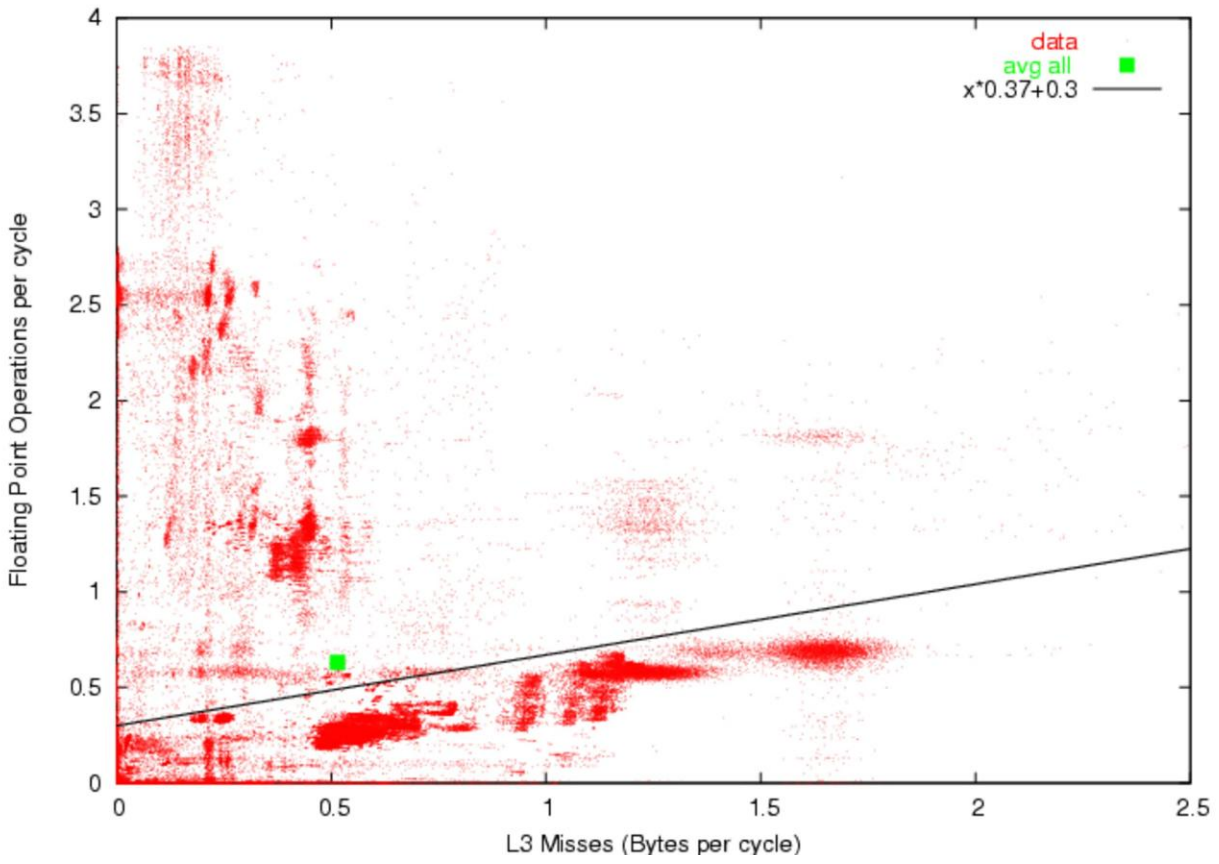


Fig.2: Floating Point Operations and L3 Misses. The green square indicates the average of all 120,000 measured samples (see Table 2.)

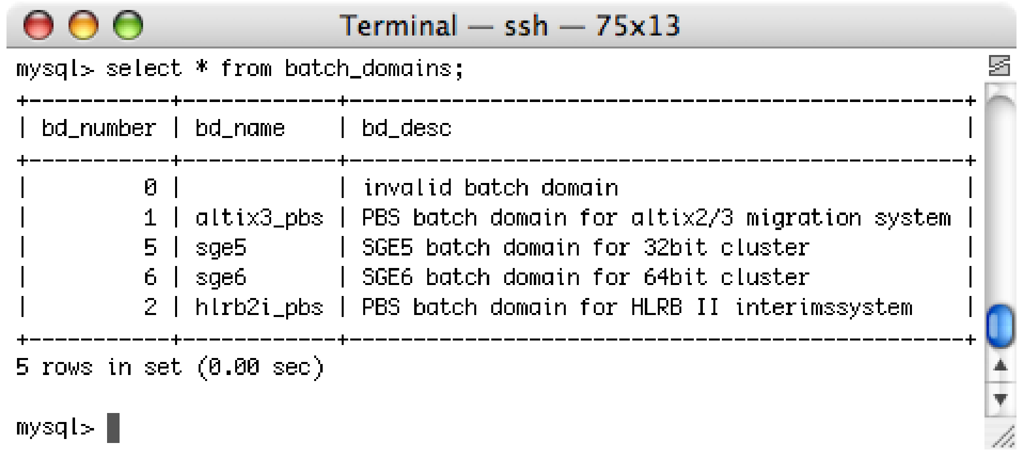
9.4 Conclusions

Caches have now reached a size where many applications can draw significant advantages from them. The measurements show that overall bandwidth to memory is not fully used, therefore emphasis must be put on latency hiding, on evenly distributing the workload to memory, and improving temporal locality for nested loop execution.

10 Dscription of the Database Tables

10 Dscription of the Database Tables

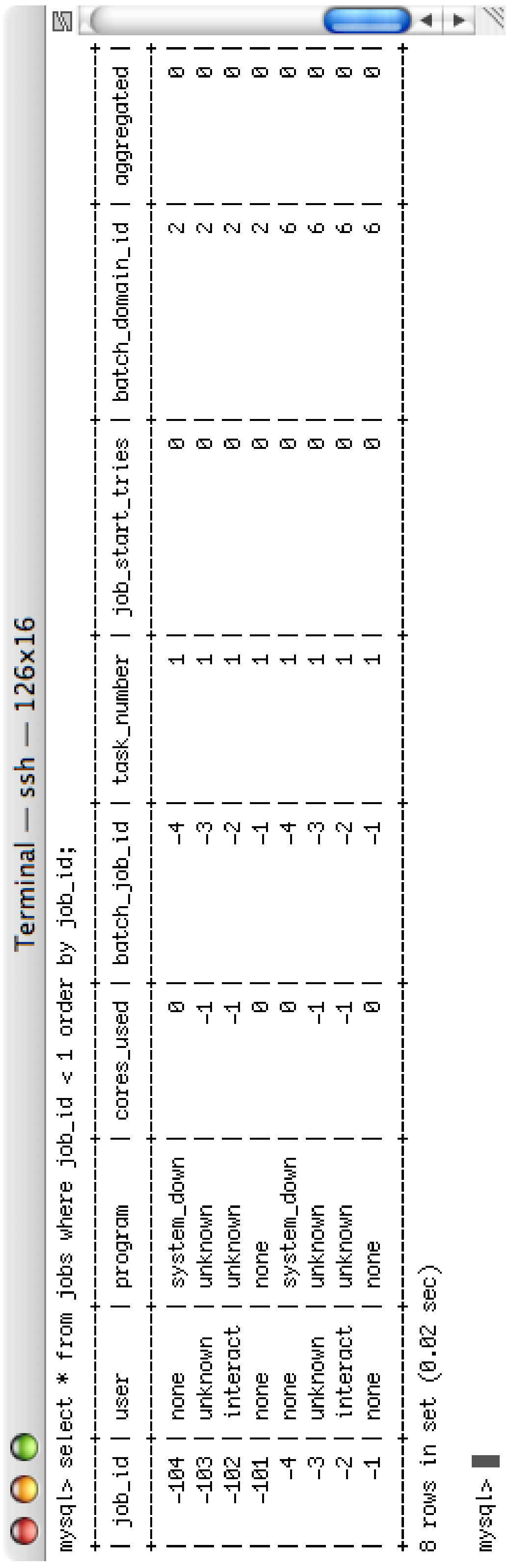
10.1 Table *batch_domains* (complete):



```
mysql> select * from batch_domains;
+-----------+------------+----------------------------------------------+
| bd_number | bd_name    | bd_desc                                      |
+-----------+------------+----------------------------------------------+
|         0 |            | invalid batch domain                         |
|         1 | altix3_pbs | PBS batch domain for altix2/3 migration system |
|         5 | sge5       | SGE5 batch domain for 32bit cluster          |
|         6 | sge6       | SGE6 batch domain for 64bit cluster          |
|         2 | hlrb2i_pbs | PBS batch domain for HLRB II interimssystem  |
+-----------+------------+----------------------------------------------+
5 rows in set (0.00 sec)

mysql> 
```

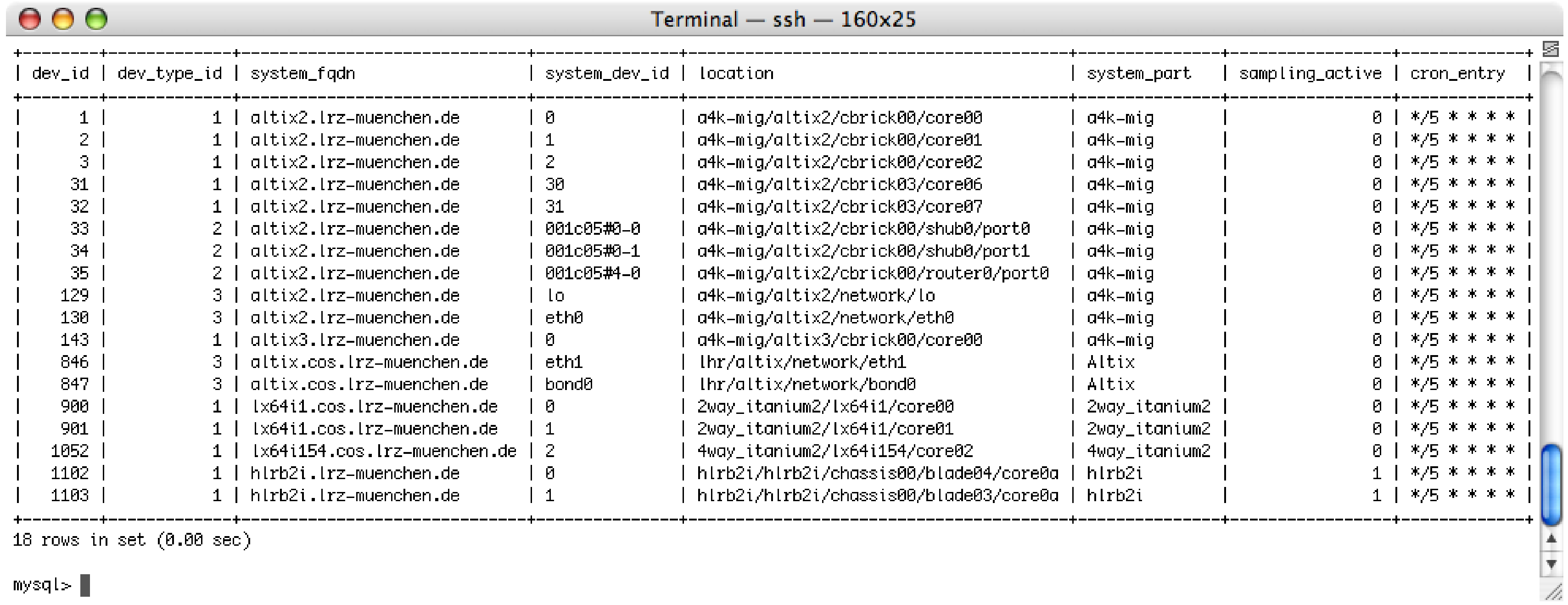
10.2 Table *jobs* (some selected job_ids with a special meaning):



```
mysql> select * from jobs where job_id < 1 order by job_id;
+--------+----------+-------------+------------+--------------+-------------+-----------------+-----------------+------------+
| job_id | user     | program     | cores_used | batch_job_id | task_number | job_start_tries | batch_domain_id | aggregated |
+--------+----------+-------------+------------+--------------+-------------+-----------------+-----------------+------------+
|   -104 | none     | system_down |          0 |           -4 |           1 |               0 |               2 |          0 |
|   -103 | unknown  | unknown     |         -1 |           -3 |           1 |               0 |               2 |          0 |
|   -102 | interact | unknown     |         -1 |           -2 |           1 |               0 |               2 |          0 |
|   -101 | none     | none        |          0 |           -1 |           1 |               0 |               2 |          0 |
|     -4 | none     | system_down |          0 |           -4 |           1 |               0 |               6 |          0 |
|     -3 | unknown  | unknown     |         -1 |           -3 |           1 |               0 |               6 |          0 |
|     -2 | interact | unknown     |         -1 |           -2 |           1 |               0 |               6 |          0 |
|     -1 | none     | none        |          0 |           -1 |           1 |               0 |               6 |          0 |
+--------+----------+-------------+------------+--------------+-------------+-----------------+-----------------+------------+
8 rows in set (0.02 sec)

mysql>
```

10.3 Table *devices* (some sample entries):



```
+--------+-------------+-----------------------------+---------------+------------------------------------------+---------------+-----------------+-------------+
| dev_id | dev_type_id | system_fqdn                 | system_dev_id | location                                 | system_part   | sampling_active | cron_entry  |
+--------+-------------+-----------------------------+---------------+------------------------------------------+---------------+-----------------+-------------+
|      1 |           1 | altix2.lrz-muenchen.de      | 0             | a4k-mig/altix2/cbrick00/core00           | a4k-mig       |               0 | */5 * * * * |
|      2 |           1 | altix2.lrz-muenchen.de      | 1             | a4k-mig/altix2/cbrick00/core01           | a4k-mig       |               0 | */5 * * * * |
|      3 |           1 | altix2.lrz-muenchen.de      | 2             | a4k-mig/altix2/cbrick00/core02           | a4k-mig       |               0 | */5 * * * * |
|     31 |           1 | altix2.lrz-muenchen.de      | 30            | a4k-mig/altix2/cbrick03/core06           | a4k-mig       |               0 | */5 * * * * |
|     32 |           1 | altix2.lrz-muenchen.de      | 31            | a4k-mig/altix2/cbrick03/core07           | a4k-mig       |               0 | */5 * * * * |
|     33 |           2 | altix2.lrz-muenchen.de      | 001c05#0-0    | a4k-mig/altix2/cbrick00/shub0/port0      | a4k-mig       |               0 | */5 * * * * |
|     34 |           2 | altix2.lrz-muenchen.de      | 001c05#0-1    | a4k-mig/altix2/cbrick00/shub0/port1      | a4k-mig       |               0 | */5 * * * * |
|     35 |           2 | altix2.lrz-muenchen.de      | 001c05#4-0    | a4k-mig/altix2/cbrick00/router0/port0    | a4k-mig       |               0 | */5 * * * * |
|    129 |           3 | altix2.lrz-muenchen.de      | lo            | a4k-mig/altix2/network/lo                | a4k-mig       |               0 | */5 * * * * |
|    130 |           3 | altix2.lrz-muenchen.de      | eth0          | a4k-mig/altix2/network/eth0              | a4k-mig       |               0 | */5 * * * * |
|    143 |           1 | altix3.lrz-muenchen.de      | 0             | a4k-mig/altix3/cbrick00/core00           | a4k-mig       |               0 | */5 * * * * |
|    846 |           3 | altix.cos.lrz-muenchen.de   | eth1          | lhr/altix/network/eth1                   | Altix         |               0 | */5 * * * * |
|    847 |           3 | altix.cos.lrz-muenchen.de   | bond0         | lhr/altix/network/bond0                  | Altix         |               0 | */5 * * * * |
|    900 |           1 | lx64i1.cos.lrz-muenchen.de  | 0             | 2way_itanium2/lx64i1/core00              | 2way_itanium2 |               0 | */5 * * * * |
|    901 |           1 | lx64i1.cos.lrz-muenchen.de  | 1             | 2way_itanium2/lx64i1/core01              | 2way_itanium2 |               0 | */5 * * * * |
|   1052 |           1 | lx64i154.cos.lrz-muenchen.de | 2            | 4way_itanium2/lx64i154/core02            | 4way_itanium2 |               0 | */5 * * * * |
|   1102 |           1 | hlrb2i.lrz-muenchen.de      | 0             | hlrb2i/hlrb2i/chassis00/blade04/core0a   | hlrb2i        |               1 | */5 * * * * |
|   1103 |           1 | hlrb2i.lrz-muenchen.de      | 1             | hlrb2i/hlrb2i/chassis00/blade03/core0a   | hlrb2i        |               1 | */5 * * * * |
+--------+-------------+-----------------------------+---------------+------------------------------------------+---------------+-----------------+-------------+
18 rows in set (0.00 sec)

mysql>
```