Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Master's Thesis

Design and Implementation of a Profiling Environment for Trace Based Analysis of Energy Efficiency Benchmarks in High Performance Computing

Name:	Stephan Krempel
Matriculation number:	2666860
Supervisors:	Prof. Dr. Thomas Ludwig
	Julian M. Kunkel
Date:	31 August 2009

Ich versichere, dass ich diese Master-Arbeit selbstständig verfasst, nur die angegebenen Quellen und Hilfsmittel benutzt und die Grundsätze und Empfehlungen "Verantwortung in der Wissenschaft" der Universität Heidelberg beachtet habe.

31. August 2009, ...

Abstract

Energy Efficiency is important since the ever rising energy consumption of information technology becomes increasingly expensive and a large share in earth pollution. Especially in High Performance Computing power saving mechanisms are not available, yet. For developing and verifying such mechanisms it would be conducive to know how the power consumption of a system is related to the utilization of its single components. To analyze this relationship a profiling environment is needed, which is capable of tracing the power consumption of the nodes in a computer cluster together with the cluster components utilization. In this thesis such an environment is developed. We enhance our own XML-based tracing format HDTrace with a binary extension for efficient support of statistics traces and adapt the trace writing library. Then two entirely new libraries are implemented using the extended format, one for power and one for utilization tracing. The power consumption of the nodes is measured by an external power analyzer controlled over the serial interface. Utilization values are provided directly by the Linux operating system. Afterwards the libraries are evaluated for correctness and performance. Finally an analysis of two benchmarks concerning their impact to the system's power consumption demonstrates the use of the libraries. In conclusion an useful environment is now available that will help in further developments of software driven power saving solutions for computer clusters.

Contents

1 Introduction

Today, "Green Thinking" is not only popular in itself, it is popular for very good reasons. Most parts of the world are already highly engineered and the period until the rest will follow is just a matter of time. By now all the machines and devices we are using every day need very high amount of power to run their jobs, power that is mostly produced in a way causing pollution of several kinds. Beside this, the production, shipping and disposal causes further pollution.[1]

In the past few years some benchmarks for measuring electrical energy consumption were developed. However almost all of them are focusing on personal computers, workstations[2, 3], or single components like the CPU[4]. Even the SPEC[1], known for its benchmark suites for industrial use, started to make a power benchmark called SPECpower_ssj2008. This one initially focused on single servers while some multi node support was added lately. More power benchmarks are being developed but not yet available, like the TCP-Energy[2].

But in our days there are not only the personal computers, gaming consoles, mobile phones and similar devices that we use every day, not only the hidden computer systems that control traffic signals, street lighting and even coffee machine. There is a growing number of much bigger computer installations that most people are not aware of when using services provided by them or products developed with their intensive help. Searching for a word with Google[3] or looking up an article in Wikipedia[4] is done by huge server farms. Developing cars with complex security features available today is only possible by doing simulations on large computer clusters. Also the daily weather forecast is based on simulations for which computer clusters are running 24 hours a day, 7 days a week.

These large computer installations need enormous amounts of electrical power. In fact this is not only a matter of earth pollution, it is also an economical and technical one. First, the electrical power is expensive. On of the biggest parts of the cost of operation for computer clusters is the money to be spent for electrical energy. In big installations

[1] SPEC is the Standard Performance Evaluation Corporation (`http://www.spec.org/`)
[2] TCP-Energy by the Transaction Processing Performance Council (`http://www.tpc.org/tpc_energy/default.asp`)
[3] Google Search Engine (`http://www.google.com`)
[4] Wikipedia – The Free Encyclopedia (`http://www.wikipedia.org`)

it is not only the energy needed to run the computers with CPU, memory, network and storage themselves. It is also the effort needed for large cooling systems to dissipate the waste heat. When we take a look at the cost of ownership for such an installation, we can see, that in spite of the very expensive price for facilities, installation, computers, cooling and maintenance, the energy costs still are a very big part.[5]
Second, we have already reached a point, where getting more power to one single place for a computer cluster installation is a real technical problem. The top was reached in 2002 when an own power plant was built to power up the supercomputer Earth Simulator in Yokohama, Japan.[6]

Due to the effort of long battery life for laptops and other mobile devices, the research and development of power saving technologies in this area already started more than ten years ago. The resulting knowledge could be easily transported to normal Desktop PCs when green thinking came up. Even for servers a limited transportation of known technology is possible and realized these days. Most energy saving in this area is achieved by automatically putting the system or at least parts of it into lower power states after a defined period without activity. By using this relatively simple strategy, only the hardware support is crucial. The system is slowed down by itself since waking up components from lower power states on demand can never happen in zero time.

Unfortunately, the situation is entirely different in high performance computing with computer clusters. In a perfect world, a computer cluster would never run out of work and every resource would be used all the time. Of course, in this perfect world there would be no chance to save any power. Even if we are not living in a perfect world, the ambition with computer clusters is always to have no idling resources at all. No program can use all resources at all times, in particular it will never use them all concurrently. Hence, we can presume a power saving capability in the field of cluster computing, too.

As the name implies, in high performance computing the strict focus is on high performance. If we use the same mechanisms as with desktop systems meaning that every node would go to sleep while idling for a short time, this can easily grow from a small to a very large performance impact. This concept of using the past for making decisions effecting the future could in worst case do more bad than good. So what we really need is a crystal ball to look into the future and know when a program will not use a specific component for a certain amount of time and when it will need it again. With this knowledge we could perfectly schedule the power states without any performance impact to the running program.

Of course there is no way to really take a look into the future, but there is someone who can even know the needed information. It's the developer of the program. He knows when the program will perform only calculations not using the network interface, and he knows when the program will do only communication for collecting data or do only I/O for check-pointing and so on. Hence, it would be nice to have the developer telling this information to the system, so it can schedule power saving actions based upon it.

There are several ways this could be done, but all of them need lots of work. First a new infrastructure must be provided or an existing one must be extended to give the programmer the possibility of providing the information to the system. Then the system must be modified to use the information in order to control power saving mechanisms. Finally, the existing programs must be adapted to use the new infrastructure which is perhaps the greatest afford. Before starting such an ambitious project, we have to investigate whether this approach can at least theoretically lead to significant power saving. To do this, a program is needed that can stress different components of typical cluster nodes and measure the power consumption at the same time. For verification of such a program we will need to trace the programs resource usage in conjunction with the consumed power. Therefore, we need a power benchmark that is capable of making conclusions about power consumption of single components.

The goal of this thesis is to develop or enhance, respectively, a tracing environment that meets the requirements mentioned above. The results should be used for a first analysis and in order to evaluate the usability of existing benchmarks for the described task.

First the development of a concept for the work is needed. Therefore, some foundations are introduced in chapter 2. Chapter 3 deals with the design of the new environment as well as the decision to use our own tracing format HDTrace. Important aspects of the enhancement and extension of the implementation of the HDTrace Writing C Library and the implementations of the new Resources Utilization Tracing Library and the Power Tracer are described in chapter 4. After the implementation work is done, some evaluation is presented in chapter 5. Finally, the new environment is used to analyze the existing Benchmarks SPECpower and HPCC in chapter 6. At the end of the thesis, some conclusions are made in chapter 7 and some ideas for future works are given in the last chapter 8.

2 Foundations

This chapter describes already existing developments used as foundation for the following work. Then a short overview of competing approaches is given. Finally a few words about the development environment are written.

2.1 PIOsim and HDTrace

In our research group for parallel and distributed systems, Julian Kunkel is developing the PIOsim Cluster Simulator in the scope of his doctor thesis. This is a software simulator written in Java that simulates a complete computer cluster with focus on parallel I/O. Its mean purpose is intended to be the simulation of parallel applications using MPI and MPIIO for analyzing various different communication and I/O schemes. One of the key ideas for PIOsim is to be able to use traces of real programs as input for the simulator. With this capability it will be possible to replay the program execution within the simulator. On one hand this will allow to verify the simulator, on the other hand hardware characteristics can be changed for instance to get insight how the program would run on a different network topology. In order to achieve this goal, a new trace file format along with the corresponding tracing infrastructure had to be created. The general concept for the new tracing format has already been developed and a Java access library has been implemented. A first version of a C library for creating traces in the new format is already implemented, too.

The new tracing format is called HDTrace, where the HD stands for Heidelberg and also Highly Descriptive. It is completely XML based and therefore well-defined and easy to extend. Even if it is designed as input format for PIOsim, it can be used as general purpose tracing format.

There have been various reasons for not using one of the many existing tracing formats, mainly the insight that they all come with some restrictions that would have made it unnecessarily complex to use them. A detailed discussion of this decision is not within the scope of this thesis.

The Existing HDTrace Format

The initial draft of HDTrace format is specific to tracing parallel programs using MPI and replay them in the simulator. For each thread of each process or MPI rank of a parallel program, respectively, one *trace file* and one *info file* is generated locally on the node the process is running on. The trace file is simply an XML text file. For every event possible exists an XML element type and each recorded event is represented as one XML element of this type. The elements can have several attributes containing further information about the event, for example the mandatory timestamp. Events can be either single events represented by a single tag or state events having a start and end tag. Furthermore, there are some special element types to allow complex structures such as nested and overlapping operations. The info file is used to store structural information that is not necessarily available at program start time. An example for such information are MPI data types that could be created dynamically during the program execution. The info file is later used to merge the traces of the whole program together without the need of parsing all the trace files which can become very large. The position of the single trace and info files is given by the naming of the file. A trace file is always named in the scheme <PROJECT>_<HOSTNAME>_<RANK>_<THREAD>.trc, the respective info file with the extension .info.

For creating the files there is an MPI wrapper using the PMPI interface provided by the MPI library. This wrapper is capable of tracing an MPI program by just linking it before the regular libmpi. To finalize a trace project and prepare it for visualization, the info files are merged by a script named project-description-merger.py into one *project file* called <PROJECT>.proj. This project file can then be loaded with the Sunshot[1] viewer.

2.2 Competing Approaches

There are several other groups engaged in tracing parallel programs. Four German groups working in that field, together started the eeClust project (Energy-Efficient Cluster Computing) in this year. The project is funded under the BMBF[2] call "HPC-Software für skalierbare Parallelrechner" and its goal is to determine relationships between the behavior of parallel programs and the energy consumption of their execution on a compute cluster. Our group is one partner in this project. The others are Dresden University of Technology (TUD/ZIH), ParTec Cluster Competence Center GmbH, and JSC of Forschungszentrum Jülich GmbH.

[1] Sunshot is a viewer for the HDTrace format written in Java by Julian Kunkel. It is a major rewrite of Jumpshot-4.

[2] BMBF (Bundesministerium für Bildung und Forschung) is the German Federal Ministry of Education and Research (`http://www.bmbf.de/en/index.php`)

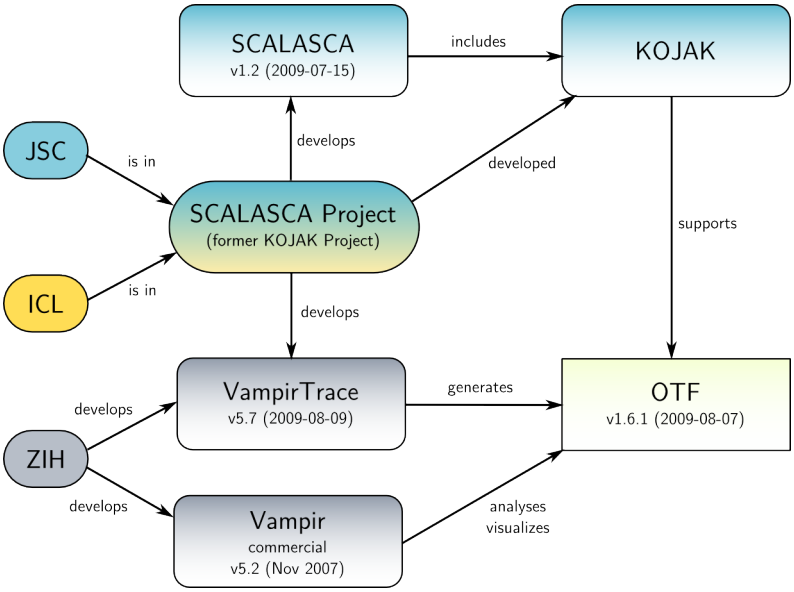


Fig. 2.1: Relevant projects of our eeClust Partners

2.2.1 ZIH: VampirTrace and Vampir using OTF

The Center for Information Services and High Performance Computing (ZIH) in Dresden is working in the field of HPC tracing for many years. Since 1996 they develop the commercial visualization software for very large traces called Vampir.

"Vampir is a graphical analysis framework that provides a large set of different chart representations of event based performance data generated through source code instrumentation. These graphical displays, including state diagrams, statistics, and timelines, can be used by developers to obtain a better understanding of their parallel program's inner working and to subsequently optimize it."[7]

Vampir supports the visualization of several trace formats, amongst others their own old VTF3 (VampirTraceFormat) and the new OTF, the Open Trace Format. OTF is an open format developed by the ZIH "to improve scalability for very large and massively parallel traces"[8]

The ZIH also develops a toolkit and library for trace creation called VampirTrace[9]. It was originally using its own format VTF3. Now it supports OTF, too. Hence, the scientists on the ZIH have available a complete environment for creating and visualizing traces of function calls and performance statistics for large-scale parallel platforms.

At the moment, they are also working at the integration of power consumption data into their traces. This development is currently independent from ours, but they should become coordinated in the scope of the eeClust project.

2.2.2 JSC: KOJAK in SCALASCA

At the Jülich Supercomputing Centre (JSC) from 1998 until 2006 the KOJAK project was running in cooperation with the Innovative Computing Laboratory (ICL) at the University of Tennessee in Knoxville. "KOJAK is a performance-analysis tool for parallel applications supporting the programming models MPI, OpenMP, SHMEM, and combinations thereof. Its functionality addresses the entire analysis process including instrumentation, post-processing of performance data, and result presentation. It is based on the idea of automatically searching event traces of parallel applications for execution patterns indicating inefficient behavior. The patterns are classified by category and their significance is quantified for every program phase and system resource involved. The results are made available to the user in a flexible graphical user interface, where they can be investigated on varying levels of granularity."[10, 11]

In the meanwhile KOJAK is integrated in SCALASCA developed by the same groups.

"SCALASCA is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. It has been specifically designed for use on large-scale systems [...], but is also well-suited for small- and medium-scale HPC platforms."[12] "SCALASCA builds on the idea of searching event traces of parallel applications for execution patterns indicating inefficient behavior. During the search process, SCALASCA classifies detected pattern instances by category and quantifies their significance for every program phase and system resource involved. The results are made available to the user in a flexible graphical user interface, where they can be investigated on varying levels of granularity."[13, Overview]

The experience in automatic trace analysis can also be useful for the subject increasing energy efficiency. As well as different execution pattern indicate inefficient behavior concerning the performance, they can also indicate inefficient behavior concerning the power consumption of the system.

2.2.3 ParTec: ParaStation

"ParTec Cluster Competence Center GmbH is a global leader in cluster management software and support services for High Performance Computing. [...] ParaStationV5 is ParTec's core software product. [...] The ParaStation software architecture includes a runtime environment for parallel jobs coupled with an optimized MPI library that supports a variety of standards-based interconnects. It also includes an extensible monitoring tool, the GridMonitor, which gives administrators a unique insight into all aspect of a cluster's state."[15]

2.3 Development Environment

All the development in our group is primarily made for the GNU/Linux operation system. Even if we are exerted to achieve compatibility with other UNIX like platforms, it is not confirmed for our software to run on another operating system.

In the PIOsim project, to which this thesis is linked, subversion[3] is in use as revision control system. I personally used the Eclipse IDE[4] in combination with Vim[5] for the development within the scope of this thesis.

This document is created using the high-quality typesetting system LaTeX with several additional packages and cooperating software like BibTeX bundled together in the TeX Live Distribution[6].

All the graphics and diagrams are created or edited using Inkscape[7] or The GIMP[8]. The flow charts are created using the yEd Graph Editor[9]

[3]Subversion (SVN) is an open source revision control system (`http://subversion.tigris.org/`)
[4]Eclipse is a free IDE supporting several programming languages (`http://www.eclipse.org/`)
[5]VImproved is an improved version of the well known vi editor (`http://www.vim.org`)
[6]TeX Live a LaTeX Distribution by the TeX user groups (`http://www.tug.org/texlive/`)
[7]Inkscape is an an Open Source vector graphics editor. (`http://www.inkscape.org/`)
[8]The GIMP is the GNU Image Manipulation Program. (`http://www.gimp.org/`)
[9]yEd is a freely available powerful graph editor. (`http://www.yworks.com/en/index.html`)

3 Design

In this chapter the design of the implementation is described, starting with the decision for our own appropriate tracing format. This is followed by a discussion of the enhanced format fitting the new needs. Finally, the design of the two new libraries that build the key element of the thesis is presented.

3.1 Choosing a Tracing Format

The first step when creating a new profiling environment is to choose a suitable trace format. Important criteria beside the capabilities of the format itself is the availability of tools, the usability of the libraries used to write the traces, interoperability issues, and available know-how.

For obvious reasons we want to achieve best interoperability between all current developments in our group. Therefore, extending HDTrace format for the needs of this thesis is the natural choice. In our former trace visualization project PIOviz[16, 17, 18, 19, 20] we had already added statistics to the traces and learned that this provides additional insight in many situations. So having support for tracing of statistical values will be a desirable feature for the new trace format, too. Furthermore the PIOsim program can be extended later to include statistics values in its simulations.

3.2 Extending HDTrace Format

The final goal is to have an environment that allows developers of MPI programs to give hints about future usage of resources to enable the operating system to make better power saving decisions. Moreover, it will be great for evaluation purposes to have traces of MPI function calls and resources utilization as well as power statistics together in one trace project. But for the moment, getting resources utilization and power statistics together is the more important point.

With the existing HDTrace format, described in section 2.1, it is possible to record measured resource utilization and power consumption values as events just as if they were function calls. This has two big disadvantages. First, we would be forced to perform all tracing from within the same program because there can only be one file with the same name. There are some other technical limitations as well. Of course these restrictions could be worked around, but having an own format gives us the great chance to do it properly. The second disadvantage is the large overhead this approach would produce in each trace file, unnecessarily. Statistical values are commonly recorded periodically. Hence, we would write the same XML element over and over having different attribute values only. Therefore we decided to rework and extend the format and developed some modifications of the existing format and an extension for good support of statistics traces.

3.2.1 Programs are Trees

On any modern computer the execution units are organized in a tree structure. When looking on a single host, we have the host as the root node of the tree. The processes running on the host are all children of the host and form the next tree level. The actual execution objects are threads, each one being a child of one process. That is, the threads present the second and deepest level of the tree. Figure 3.1 depicts this.

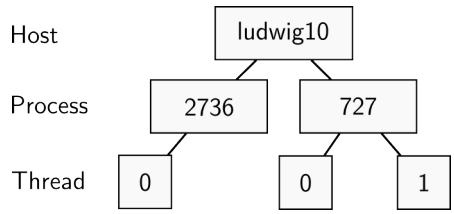


Fig. 3.1: Topology tree of a single computer

When transferring this idea to a computer cluster, there is one more tree level, the cluster itself. The cluster becomes the new root of the tree and the trees of all nodes become subtrees as shown in figure 3.2 on the next page. A parallel program running on the cluster is then represented as the cluster's tree with all subtrees removed that belong to processes which are not part of the program. The shaded nodes in figure 3.2 shows an example tree of a parallel program running on two nodes.

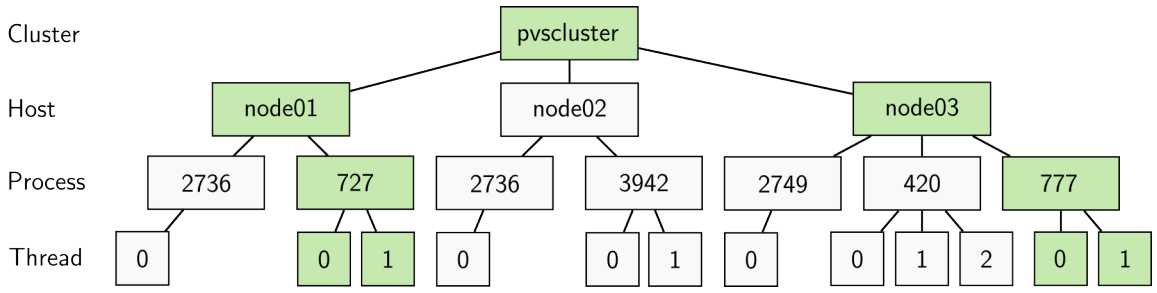


Fig. 3.2: Topology tree of a cluster

3.2.2 HDTopology - A Structure to Bind Them

In the old HDTrace format, all traces are assumed to be on the same level of the tree. This is always the case if we want to trace function calls in a program since each function call is performed by exactly one thread. For statistics, this assumption does not necessarily hold. In fact, even in the most canonical cases it does not. When you think of CPU utilization as the most common statistical value on computers, there is only one such value for each CPU on a single host, not one for each process or thread. So a statistics trace recording CPU utilization has to be created once for each cluster node, not once for each thread as a trace recording function calls. The statistics trace lives on another level of the described tree than the function call trace does.

To handle this difference conveniently, we introduce a very generic structuring method to the HDTrace format called HDTopology. The *topology* of a complete trace project describes a tree structure where each single trace can be assigned to a tree node. As we have several singly assigned trace files that build one whole trace project, this structuring method allows us to describe the associations between the single traces and statistics traces easily. Each single trace simply needs to carry the information about its location within the topology.

The root of the topology is defined always to be the project name. The types of the deeper levels can be freely specified by the user. A topology's structure is fully specified by the *topology level types*. The level type should describe the semantics of the nodes on the level. As a canonical topology structure we could define (Project, Host, Process, Thread).

Each trace is assigned to exactly one node of the topology called the trace's *topology node*. Each such topology node has a label that must be unique at the node's topology level. A node is well-defined by the unique path that leads from the root of the topology to this node. The path is the list of labels of each node in the order they are passed when walking on the edges of the tree to the target node starting from the root node.

An example for a leaf node in a topology with the canonical structure as defined in the last paragraph could be (myProject, node01, rank3, thread0).

In old HDTrace format the information that we now refer to as the ”associated topology node” was given by the file name. Since this is very convenient when looking at the files of an HDTrace project, it is left mostly unchanged. For that reason one restriction persists, each topology node can still have only one trace associated. Why this does not hold for statistics traces is described later.

The topology structure represented by the topology level types is written into the project file in order to be used as labels in trace visualization.

3.2.3 HDStats - Statistics Traces

The formal characteristic of statistics as we use the term here are

- all events have a common structure,
- events vary only in their data values,
- events commonly occur regularly and mostly periodically.

Our design goals for the extension are

- store data efficiently to save space and write time,
- no limitation of the number of statistics traces per topology node.

Efficient Data Storage In some circumstances the chosen period for the statistics can be very small. Then the statistics trace has to store lots of entries in a short time. This can make the trace file very large and is time-consuming to write. To reduce this risk the data should be stored as efficient as possible. Another positive side-effect will be that the performance impact of writing the trace will be lower the smaller the trace will be.

Unlimited Statistics per Topology Node Each statistics trace shall have only one entry structure. That means, all statistics a user may want to collect into one statistics trace must be measured by the same program using the same period length. Since this is an unacceptable restriction it should be possible to have more than one statistics trace for the same topology node, ideally unlimited many traces. Thus one node can have

multiple statistics traces associated with different entry structures, period lengths, and even created by different programs.

We could use the existing event tracing capability to trace statistics, too. However, remember that each event is recorded as full XML element and that all events of a statistics trace have exactly the same structure that would be logged with every event. This would produce lots of meaningless redundant bits and bytes, wasting lots of space and definitely breaking the first design goal. So the main idea in order to meet the goal is to define the structure for all entries once and then write only the varying data values for each entry. For best space and time efficiency the values can then be stored in binary format. Hence, we need a major extension of the HDTrace format named HDStats.

To meet the second design goal, another structuring method for statistics traces has to be introduced for the case that more than one statistics trace should become assigned to the same topology node. Each statistics entry can contain not only one value but a group of values. For example the CPU utilization and memory usage can always be logged together into one entry. Thus, each entry structure actually builds a group and we talk about one statistics file to contain a *statistics group*. This group can be given a name which can then be used to distinguish the statistics files associated to the same topology node from each other.

The HDStats extension shall provide a better format for statistics event tracing and live beside the already existing format for tracing irregular events. In order to save storage space as well as time for writing, the structure of the entries shall be defined only once and the data shall be written in binary format. Since this results in a completely different file format, we define the new *statistics file* type with the extension .stat.

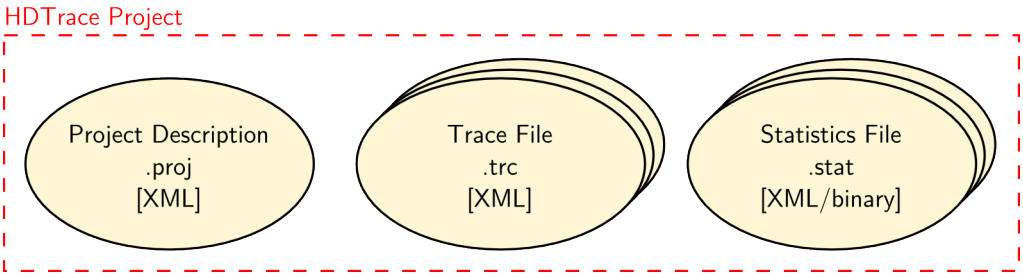
A statistics file has a describing header in XML format followed by binary data. For interpreting the binary data the knowledge of the HDStats format and the information in the header should be sufficient. The name of the statistics group has to be an additional part of the filename to avoid naming conflicts. So the filename of a statistics file is built with the scheme

<PROJECT>_<TOPOLEVEL1>_<TOPOLEVEL2>_..._<GROUPNAME>.stat.

3.3 Summary: The new HDTrace Format

First of all, this is still a work in progress and therefore the HDTrace format definition described here is just a snapshot, not a final or official version. It is even not complete in

this regard, that is, the parts described here are only those ones that are important for this thesis. Some new parallel but independent developments are completely ignored.



An HDTrace project consists of multiple files that are placed in the same directory. The files are associated with each other in such a manner that they contain traces belonging together, mostly recorded at the same time in the same system.

3.3.1 The Topology

The **topology** is an abstraction of how the single trace files belong together. It can always be drawn as a tree graph. The user has to define the **topology structure** by defining the types of the tree levels. The type of the root level (level 0) is always "Project" and cannot be changed. The type of any other level is an alphanumerical string with ASCII characters only.

A node of the topology is called **topology node**. Each such node has a label given by the user. The label must be unique at the topology level the node resides. Also the label is an alphanumerical string with ASCII characters only. A topology node is globally specified by its path. The **path** is the list of consecutive labels of the nodes to cross when walking on the tree graph's edges from the root to the target node. That is, the first element of a node's path is always the project name.
A unique identification string of a node, its **path string**, can be formed by concatenating the node path's elements and separating them by underscores.

3.3.2 The Project File

The central file of an HDTrace project is the **project file**, an XML file named with the project name and the extension .proj. In the project file, a description of the project can be given. The whole topology of the project is defined with all the topology nodes where associated traces exist for in the project. For MPI traces there are informations stored about used communicators and datatypes. For existing statistics traces within

the project the used statistics group names are put into the file. In addition, there is a list of files accessed by MPI applications needed for PIOsim. You can see an example project file in listing 3.1. A formal definition for this file type will be given later in another document.

```
<?xml version="1.0" encoding="UTF-8"?>
<Application name="partdiff-par">
  <Description>PDE solver run (0 2 600 2 2 50)</Description>
  <Topology>
    <Level type="Hostname">
      <Level type="Rank">
        <Level type="Thread" />
      </Level>
    </Level>
    <Node name="node08">
      <Node name="0">
        <Node name="0" />
      </Node>
      <Node name="2">
        <Node name="0" />
      </Node>
    </Node>
    <Node name="node09">
      <Node name="1">
        <Node name="0" />
      </Node>
    </Node>
  </Topology>
  <CommunicatorList>
    <Communicator name="WORLD">
      <Rank global="2" local="2" cid="0" />
      <Rank global="1" local="1" cid="0" />
      <Rank global="0" local="0" cid="0" />
    </Communicator>
    <Communicator name="MPI_ICOMM_WORLD">
      <Rank global="2" local="2" cid="1" />
      <Rank global="1" local="1" cid="1" />
      <Rank global="0" local="0" cid="1" />
    </Communicator>
  </CommunicatorList>
  <Datatypes>
    <Rank name="2" thread="0">
      <NAMED id="0" name="MPI_CHAR"  />
      <NAMED id="1" name="MPI_DOUBLE"  />
      <NAMED id="2" name="MPI_BYTE"  />
    </Rank>
    <Rank name="1" thread="0">
      <NAMED id="0" name="MPI_CHAR"  />
      <NAMED id="1" name="MPI_DOUBLE"  />
      <NAMED id="2" name="MPI_BYTE"  />
    </Rank>
    <Rank name="0" thread="0">
      <NAMED id="0" name="MPI_CHAR"  />
      <NAMED id="1" name="MPI_DOUBLE"  />
      <NAMED id="2" name="MPI_BYTE"  />
    </Rank>
  </Datatypes>
  <ExternalStatistics>
    <Energy />
    <Utilization />
  </ExternalStatistics>
</Application>
```

Listing 3.1: Example HDTrace project file

3.3.3 Trace Files

A **trace file** contains the trace of arbitrary events. It can contain virtually everything as event that can be represented as valid XML element. There can be any number of trace files in a trace project but at most one per topology node.

Each trace file is a well formated XML file containing the event trace for one topology node. Each event is represented by an XML element and has at least the attribute `time` representing the start time of this event. There are two event types supported, single events with only one timestamp and state events with start and end time. Events can be arbitrarily nested. There are special XML element types to handle that. A shortened example of a trace file is given in listing 3.2. As for the project file the formal definition will be given later in another document.

```
<Program timeAdjustment='1247806573'>
[...]
<Sendrecv size='38472' toRank='1' toTag='42' fromRank='1' fromTag='43' cid='0' count='4809'
 sendTid='1' recvTid='1'  time='47.098971' end='47.100749'/>
<Recv fromRank='1' fromTag='44' cid='0'  time='47.869175' end='47.890586'/>
<Recv fromRank='2' fromTag='44' cid='0'  time='47.890595' end='47.942539'/>
<Recv fromRank='3' fromTag='44' cid='0'  time='47.942544' end='48.022254'/>
<Recv fromRank='4' fromTag='44' cid='0'  time='48.022262' end='48.022268'/>
<Recv fromRank='5' fromTag='44' cid='0'  time='48.022272' end='48.022275'/>
<Recv fromRank='6' fromTag='44' cid='0'  time='48.022278' end='48.022281'/>
<Recv fromRank='7' fromTag='44' cid='0'  time='48.022284' end='48.022286'/>
<Send size='8' count='1' tid='1' toRank='1' toTag='45' cid='0'  time='48.022290' end='48.022315'/>
<Send size='8' count='1' tid='1' toRank='2' toTag='45' cid='0'  time='48.022319' end='48.022334'/>
<Send size='8' count='1' tid='1' toRank='3' toTag='45' cid='0'  time='48.022338' end='48.022351'/>
<Send size='8' count='1' tid='1' toRank='4' toTag='45' cid='0'  time='48.022355' end='48.022399'/>
<Send size='8' count='1' tid='1' toRank='5' toTag='45' cid='0'  time='48.022403' end='48.022429'/>
<Send size='8' count='1' tid='1' toRank='6' toTag='45' cid='0'  time='48.022433' end='48.022447'/>
<Send size='8' count='1' tid='1' toRank='7' toTag='45' cid='0'  time='48.022451' end='48.022496'/>
<Recv fromRank='1' fromTag='46' cid='0'  time='48.022503' end='48.221360'/>
<Recv fromRank='2' fromTag='46' cid='0'  time='48.221372' end='48.419424'/>
<Recv fromRank='3' fromTag='46' cid='0'  time='48.419435' end='48.617851'/>
<Recv fromRank='4' fromTag='46' cid='0'  time='48.617861' end='48.735424'/>
<Recv fromRank='5' fromTag='46' cid='0'  time='48.735435' end='48.932930'/>
<Recv fromRank='6' fromTag='46' cid='0'  time='48.932961' end='49.133129'/>
<Recv fromRank='7' fromTag='46' cid='0'  time='49.133140' end='49.330113'/>
<Nested>
  <Sendrecv size='0' toRank='1' toTag='1' fromRank='7' fromTag='1' cid='1' count='0'
   sendTid='2' recvTid='2'  time='49.451536' end='49.451801'/>
  <Sendrecv size='0' toRank='2' toTag='1' fromRank='6' fromTag='1' cid='1' count='0'
   sendTid='2' recvTid='2'  time='49.451810' end='49.451831'/>
  <Sendrecv size='0' toRank='4' toTag='1' fromRank='4' fromTag='1' cid='1' count='0'
   sendTid='2' recvTid='2'  time='49.451835' end='49.452391'/>
</Nested>
<Finalize  time='49.451500' end='49.455727'/>
</Program>
```

Listing 3.2: Shorted example HDTrace trace file

3.3.4 Statistics Files

A **statistics file** contains only one entry type that arbitrarily repeats many times. It is meant for tracing events that occur periodically to produce bar plots over the time. The most common use is to record regularly measured data.

There can be any number of statistics files in a trace project and for each topology node.

Statistics File Structure

File structure:

1. 5 bytes giving the header length in bytes as decimal ASCII integer number
2. a newline character (`'\n'`)
3. Header
4. Binary part

Header structure:

1. Start tag: `<Statistics>`
2. Associated topology node
3. Statistics group definition
4. End tag: `</Statistics>`

Topology node structure:

1. Start tag: `<TopologyNode>`
2. Path of the topology node in nested XML elements of type `<Label>` with the attribute `value` containing the name of the path element.
3. End tag: `</TopologyNode>`

Statistics group definition structure:

1. Start tag: `<Group>` with the following attributes:
   - `name` containing the name of the group
   - `timestampDatatype` containing the type of the time-stamp (currently always the string "`EPOCH`")

- `timeAdjustment` containing an adjustment value in seconds to add to each time-stamp as decimal ASCII floating point number

2. Entry values in correct order given as XML elements of type `<Value>` with the following attributes:
   - `name` containing the name of the value
   - `type` containing the VALUE TYPE
   - `unit` containing the unit of the value
   - `grouping` containing the GROUPING TAG of the value
3. End tag: `</Group>`

VALUE TYPE:

- `INT32` 32 bit integer in network byte order (big endian)
- `INT64` 64 bit integer in network byte order (big endian)
- `FLOAT` 32 bit floating point in network byte order (big endian)
- `DOUBLE` 64 bit floating point in network byte order (big endian)
- `STRING` zero-byte terminated string (not yet implemented)

GROUPING TAG:

The grouping attribute allows the user to specify a tag for each value and by this to define subgroups of comparable values in the statistics group. These subgroups can later be used when visualizing the trace project for example to calculate the absolute minimum of all values with the same tag for better comparison.

BINARY PART:

The binary part contains the actual entries of the statistics group. Each entry has the following structure:

1. 4 Bytes of timestamp: full seconds since epoch
2. 4 Bytes of timestamp: full nanoseconds since last full second since epoch
3. values of the entry as specified in the header

At the very beginning of the binary part and after each stopping and restarting of the trace group, an extra initial timestamp is written for the visualization tool to know at what time to start the first bar. When the trace group is stopped, the special timestamp 0xFFFF is written to indicate that the next 8 bytes are a new initial time-stamp.

Statistics File Example

Listing 3.3 shows an example statistics file. In the binary data section each character represents one byte. `I` is one byte of the initial timestamp and `T` is one of the entry timestamp. The numbers specify the according part of the entry. Linebreaks in the binary part are for readability only and do not exist in the real file.

```
00444
<Statistics>
<TopologyNode>
  <Label value="node06" />
</TopologyNode>
<Group name="Utilization" timestampDatatype="EPOCH" timeAdjustment="-0000000000.000000000">
  <Value name="CPU_TOTAL" type="FLOAT" unit="%" grouping="CPU" />
  <Value name="MEM_USED" type="INT64" unit="B" grouping="MEM" />
  <Value name="NET_IN" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT" type="INT64" unit="B" grouping="NET" />
</Group>
</Statistics>
IIIIIIIIIIIIIII
TTTTTTTTTTTTTTTT00000000111111111111111122222222222222223333333333333333
TTTTTTTTTTTTTTTT00000000111111111111111122222222222222223333333333333333
TTTTTTTTTTTTTTTT00000000111111111111111122222222222222223333333333333333
TTTTTTTTTTTTTTTT00000000111111111111111122222222222222223333333333333333
...
```

Listing 3.3: Example HDTrace statistics file

3.4 The HDTrace Writing C Library API

To use the reworked and extended HDTrace format in a convenient manner, the C library has to be reworked, too. To give the new library a clear structure, we should inherit the three modules from the format definition: HDTopology, HDTrace and HDStats. Of course, separately the HDTopology module is useless so it has to be bundled with each of the other two modules. At the end we will have the libraries libhdTrace and libhdStats.

We use the function naming scheme presented in figure 3.3 on the following page. The prefixes for the HDTrace Writing C Library shall be:

`hdT_`	Main part (topology and tracing)
`hdS_`	Statistics part

That way, when using the standard tracing, there is no difference between topology and tracing functions concerning the function prefix. When using the statistics extension, the user can easily differentiate between extension functions and core functions.

Function Naming Scheme
The naming of all exported functions of the Libraries is done as suggested by the Java Code Conventions: ”Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.”[21, Section 9 "Naming Conventions"]
Since we do not have classes or name spaces in C, we add a prefix specifying the library or library part the function belongs to. That avoids naming conflicts and helps a reader in assigning functions to a library.

Fig. 3.3: Function Naming Scheme

Topology Part

First, some functions are needed to create and destroy a topology and topology node:

- **`hdT_createTopology`** creates a new topology. It takes the project name to use as root level name and the names of the remaining topology levels.
- **`hdT_destroyTopology`** destroys an existing topology and frees all memory used.
- **`hdT_createTopoNode`** creates a topology node. It takes the topology object of the topology the node should be created for and the path to the topology node to be created.
- **`hdT_destroyTopoNode`** destroys a topology node and frees all memory used.

Furthermore, we should provide some functions to get useful informations:

- **`hdT_getTopoDepth`** provides the depth of an existing topology.
- **`hdT_getTopoNodeLevel`** returns the topology level in which a topology node lives.
- **`hdT_getTopoPathString`** provides the path string of a topology node.
- **`hdT_getTopoPathLabel`** returns the label of a node on a given topology node's path. It takes the topology node to investigate and the topology level of the desired node label.

Tracing Part

This part already existed before this thesis and is only slightly modified. Modifications made are the names of public functions, now adapted to the new naming scheme and support for the new topology concept.

Statistics Part

Statistics are structured as so called groups. The values for each statistics group are recorded in one file in binary format. In the header of the file the group is declared together with the topology node that the group belongs to. This header is written in XML format.

First, we need functions to create such a group and defining its structure:

- **hdS_createGroup** creates a new statistics group. It takes the name of the group and the topology node the group shall belong to.
- **hdS_addValue** adds one value to the structure of a not yet committed statistics group. It takes the group where the value shall be added to, the name, type and unit of the new value and the optional grouping identifier for the value.
- **hdS_commitGroup** commits the group meaning it finalizes the group's structure, writes the header to the file and prepares it for writing the binary statistics values.

We also need some functions for writing the actual entries to the created and committed statistics groups:

- **hdS_writeEntry** writes one completely formatted entry to the group with reduced validity checking. It takes the group and the entry to write. The correctness of the given entry except for its length cannot be checked since it is specified as binary object. So it is up to the caller to not break the trace format by invalid entries.
- **hdS_writeInt32Value** writes an `INT32` value to the trace. It takes the group and the 32 bit integer value. Error checks are performed to avoid breaking the format.
- **hdS_writeInt64Value** writes an `INT64` value to the trace. It takes the group and the 64 bit integer value. Error checks are performed to avoid breaking the format.
- **hdS_writeFloatValue** writes an `FLOAT` value to the trace. It takes the group and the single precision (32 bit) floating point value. Error checks are performed to avoid breaking the format.
- **hdS_writeDoubleValue** writes an `DOUBLE` value to the trace. It takes the group and the double precision (64 bit) floating point value. Error checks are performed to avoid breaking the format.

Since we want to be able to do intermittent statistical tracing without creating a new group each time, we need functions to enable or disable the data recording for a group and to check its current state:

- **hdS_enableGroup** enables the recording in a group. It takes the group to enable.
- **hdS_disableGroup** disables the recording in a group. It takes the group to disable.
  When a group is disabled, all calls to the `hdS_*` for this group should be silently ignored.
- **hdS_isEnabled** returns the recording state of a group. It takes the group to test.

Finally, we need a function for finalizing the group and do the cleanup:

- **hdS_finalize** finalizes the group, destroys it and frees all memory. It takes the group to finalize.

Table 3.1 summarizes the API.

`hdT_createTopology`	Create topology.
`hdT_destroyTopology`	Destroy topology.
`hdT_createTopoNode`	Create topology node.
`hdT_destroyTopoNode`	Destroy topology node.
`hdT_getTopoDepth`	Get depth of topology.
`hdT_getTopoNodeLevel`	Get topology level of topology node.
`hdT_getTopoPathString`	Get path string of topology node.
`hdT_getTopoPathLabel`	Get label of topology node (by path).
`hdS_createGroup`	Create statistics group.
`hdS_addValue`	Add value to uncommitted group.
`hdS_commitGroup`	Commit group.
`hdS_writeEntry`	Write complete formated entry to group.
`hdS_writeInt32Value`	Write `INT32` value to group.
`hdS_writeInt64Value`	Write `INT64` value to group.
`hdS_writeFloatValue`	Write `FLOAT` value to group.
`hdS_writeDoubleValue`	Write `DOUBLE` value to group.
`hdS_disableGroup`	Disable group.
`hdS_isEnabled`	Enable group.
`hdS_finalize`	Finalize group.

Tab. 3.1: API of the HDStats Library Module (libhdStats)

3.5 An Environment for Resources Utilization and Power Tracing

Now a trace format is specified that has all capabilities that we need for tracing resources utilization and power consumption, we can design the desired environment. Remember that our actual goal is to analyze the correlation between the utilization of single components and the power consumption of the system.

All modern operating systems provide methods for getting the utilization of different components in real-time. Many utilities like `top` use this capabilities. So getting this information into a trace on a per host base should be possible with moderate effort. The way of our choice is to build a library that can be linked to an arbitrary program making some function calls to trace the resources utilization on the host the program is running on. For example such a program could be a benchmark stressing the host's components. If this is a parallel program it should ensure by itself to create only one trace per host by calling the library functions only once per host. The API of the Resources Utilization Tracing Library is designed in section 3.5.1 on the next page.

Getting a power trace will be harder. A common computer does not have built in power measuring capabilities, at least not for all components. So first of all, the question is where and how to measure the power consumption. The finest picture we could achieve, is to get the real consumption of each single component inside of a host. This is not feasible since we would need special circuit boards for getting this information for important components as CPU, memory and network. When we take a closer look at the problem, it is not even assured, that this method would really give us the desired results. All the components are working together and the impact they have to each other is very complex. For example, the power that a common ethernet network interface card alone needs for sending data is just a portion of the power needed by the system for sending data. The ethernet card needs the main memory where the data is buffered, it needs the bus to transfer the data and perhaps it needs the CPU or DMA controller to control the transfer. So it will be the best to measure the power of each task unit that is independent concerning its power consumption. The smallest unit that fits this needs is commonly one host.

Measuring the power of a complete host is relatively easy. We just need a sophisticated power analyzer which is capable to get the power consumption with high accuracy, frequency and which is able to deliver them in machine readable format to one of our hosts that is running the tracing program. Such power analyzers are available (e.g. ZES ZIMMER LMG450 with four channels used in the tests later) and we can define the measurement setup as shown in figure 3.4 on the following page.

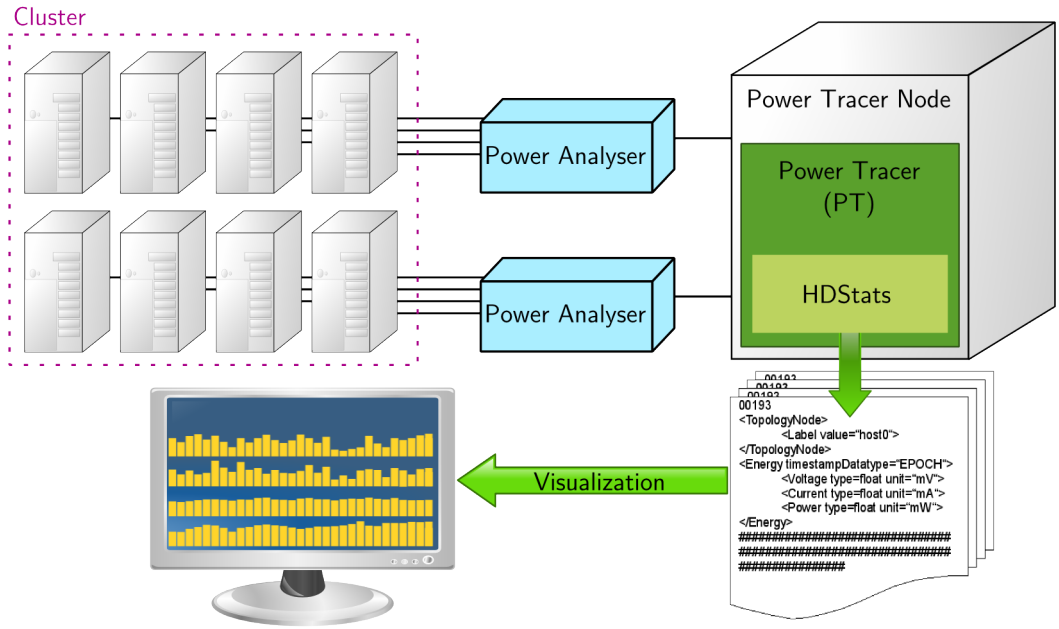


Fig. 3.4: Hardware setup for power measurement

Note that the power analyzer measures four nodes but is only connected to one node. That is, not each host can get its own power consumption and so we cannot have each host creating its own power trace and cannot create the power tracer as a library simply linked to a program running on the host. We have to create an extra program that knows about the power analyzer setup to create the power traces correctly. The Power Tracer is designed in detail in section 3.5.2 on page 32.

3.5.1 Resources Utilization Tracing Library API

This library shall provide the capability to trace as many utilization values as the system provides. In the first place there are the important components CPU, main memory, network interface and hard disk drive. When there are multiple units of one kind, measuring the utilization of each single unit shall be possible as well as getting the aggregated utilization for all units of the same kind. The user shall be able to choose the values included in the trace.

The API should by as simple as possible. First, we need a possibility for the user to specify the desired resources to trace. So we need some kind of object that contains a switch for each kind of data the library is able to trace. We call them sources and define the sources listed in table 3.2 on the following page

Source	Description	Unit
CPU_UTIL	Aggregated CPU utilization of all CPUs	Percent
CPU_UTIL_X	CPU utilization of each single CPU	Percent
MEM_USED	Amount of used Memory	Bytes
MEM_FREE	Amount of free Memory	Bytes
MEM_SHARED	Amount of shared Memory	Bytes
MEM_BUFFER	Amount of buffered Memory	Bytes
MEM_CACHED	Amount of cached Memory	Bytes
NET_IN	Aggregated incoming network traffic for all network interfaces	Bytes
NET_OUT	Aggregated outgoing network traffic for all network interfaces	Bytes
NET_IN_EXT	Aggregated incoming network traffic for all external network interfaces	Bytes
NET_OUT_EXT	Aggregated outgoing network traffic for all external network interfaces	Bytes
NET_IN_X	Incoming network traffic for each single network interface	Bytes
NET_OUT_X	Outgoing network traffic for each single network interface	Bytes
HDD_READ	Amount of data read from harddisk	Bytes
HDD_WRITE	Amount of data written to harddisk	Bytes

Tab. 3.2: Available sources of the Resources Utilization Tracing Library

We also need functions to create and finalize the trace:

- **rut_createTrace** creates a utilization trace. It takes the source configuration to trace.
- **rut_finalizeTrace** finalizes a utilization trace and frees all memory. It takes the trace to finalize.

To inherit the capability of intermittent tracing from the statistics library, we need functions to start and stop tracing:

- **rut_startTrace** starts the tracing in a utilization trace. It takes the trace.
- **rut_stopTrace** stops the tracing in a utilization trace. It takes the trace.

In conclusion, we get the API listed in table 3.3.

`rut_createTrace`	Create trace.
`rut_startTrace`	Start tracing.
`rut_stopTrace`	Stop tracing.
`rut_finalizeTrace`	Finalize trace.

Tab. 3.3: API of the Resources Utilization Tracing Library (libRUT)

3.5.2 Power Tracer

As described at the beginning of section 3.5, for tracing the power consumption it is insufficient to provide just a library as we do for utilization tracing. We need a program that knows about the setup, which channel of the power analyzer is connected to which node's power supply (see figure 3.4 on page 30).

Nevertheless, it is desirable to be able to use the power tracing functionality from other programs than those provided here, too. For example when integrating power tracing into the MPI wrapper for automatic generation of MPI traces mentioned in the last paragraph of section 2.1, this could be needed[1]. Hence, putting all the core power tracing functionality in a library and implementing the actual Power Tracer program as a frontend using this library is a good choice. We distinguish between the Power Tracer Library (libPT) and the Power Tracer.

Configuration

Since the configuration of the library can be very complex, realizing it with many function calls can be very inconvenient. Furthermore, for each program using the library directly it would be necessary to either build a complex configuration system or to recompile the program for each change in the hardware setup. So the best way to handle the configuration is a well-defined configuration file, passed to and read by the library itself.

The configuration file has to contain the general hardware setup of the power analyzer and all traces to create with all relevant information. Therefore the file is divided into sections, one for the general setup and one for each trace to create.

[1] In fact, this work is already done in the meanwhile and works great.

In the general section we need the following values:

- `device` specifying the type of the power analyzer in use.
- `port` specifying where the power analyzer is connected.
- `cycle` specifying the interval length for the tracing.
- `project` specifying the project name.
- `topology` specifying the topology for the project.

In each trace section we need the following values:

- `type` specifying the type of the trace.
- `node` specifying the node to associate with the trace.
- `channel` specifying the channel of the power analyzer to use for the trace.
- `values` specifying which values to trace

Instead of a complex formal definition, the self explaining example in listing 3.4 is given with some comments following.

```
[General]
device=LMG450
port=node06:/dev/ttyUSB0
cycle=100
project=MeinProjekt
topology=Cluster_Host_Process_Thread

[Trace]
type=HDSTATS
node=pvs_node06
channel=1
values=Utrms,Itrms,P

[Trace]
type=HDSTATS
node=pvs_node07
channel=2
values=Utrms,Itrms,P
```

Listing 3.4: Example configuration file for the Power Tracer Library

The value of `port` in line 3 has the format `NODE:DEVICE`. `NODE` is the node which has the power analyzer connected via the RS-232 port. `DEVICE` is the name of the serial device file to use.

The value of `topology` in line 6 is built using the topology level types concatenated with underscores in the same way as the topology node path string is created.

The topology `node` in line 10 is given as the topology node path string and has to match the `topology` defined. Of course, as in the example, the topology can have more levels than the node uses. This is necessary if the traces should be part of a HDTrace project with a larger topology.

`values` (line 12) is a comma separated list of strings, defining the values to trace. The strings can be specific to the used power analyzer.

In this concept for each power analyzer in the system a separate configuration file has to be created and one instance of the Power Tracer or Power Tracer Library is to be started. There are already ideas for another concept more suited to daily use in a computer cluster (compare chapter 8.2). But for initial studies this simple approach should be sufficient.

Library API

With all configuration put into the configuration file, the API of the Power Tracer Library can be small and very similar to the API of the Resources Utilization Tracing Library. That's nice because it make things easier for the user.

We need functions to create and finalize the trace:

- **`pt_createTrace`** creates a power trace. It takes the name of the configuration file.
- **`pt_finalizeTrace`** finalizes a power trace and frees all memory. It takes the trace to finalize.

At this point we also want to keep the capability of intermittent tracing provided by the statistics library. Thus we need the functions to start and stop tracing:

- **`pt_startTrace`** starts the tracing in a power trace. It takes the trace.
- **`pt_stopTrace`** stops the tracing in a power trace. It takes the trace.

We define one additional getter for the name of the host the power analyzer is connected to. This function is needed by parallel programs to decide on which host respectively in which process to call the power tracing functions:

- **`pt_getHostname`** provides the name of the host connected to the power analyzer's data port. This information is read by the library from the configuration file.

In conclusion, we get the API listed in table 3.4.

`pt_createTrace`	Create a power trace object.
`pt_getHostname`	Return the host with the measuring device connected.
`pt_startTracing`	Start the power tracing.
`pt_stopTracing`	Stop the power tracing.
`pt_finalizeTrace`	Finalize and free a power trace object.

Tab. 3.4: API of the Power Tracer Library (libPT)

Power Tracer Frontend

The actual Power Tracer is a program called pt taking at least the configuration file as parameter. It uses the Power Tracer Library to do the power tracing. The program should run non-interactively and use signals for correct termination so it is possible to run it as a daemon in the background if desired by the user.

If pt.cfg is the name of the configuration file, the program call syntax is:

```
./pt -c pt.cfg
```

We now have specified the concepts for the enhanced HDTrace format as an efficient tracing format for statistics traces and the library to write them. Furthermore two other libraries are designed, both using the new tracing statistics traces for different things. One of them records the power consumed by hosts using data from an externally connected power analyzer. The other one traces the utilization of various resources inside the host. Combined we have the concept for an environment to trace resource utilization and power consumption of a computer system together and so be able to identify relationships between them. Now the implementation of the three libraries is pending.

4 Implementation

This chapter describes the implementation of the HDTrace Writing C Library, the Performance Tracing Library and the Power Tracer. Here, only important and complex issues of the implementation are discussed. The complete documentation of the APIs can be found in the appendix and in detail description of the implementation are given in the source code documentations.

4.1 General Concepts

4.1.1 A Whiff of Objects

The programming language C which we use here for all implementations does not natively provide concepts like objects. But if we consider an object just as a set of data, it can be easily represented in C by the language's structure construct. Here, each time when referring to objects, in fact a set of data stored in a C structure is meant. For external usage the real nature of the object is irrelevant, the structure definition is always hidden by a typedef for convenience. Actually, an object pointer always means a pointer to a type, defined to hide the underlying structure with the object's data.

```
struct myObject_s {
  int                counter;
  struct myObject_s  *self;
};

typedef struct myObject_s myObject;
```

Listing 4.1: Example object definition of `myObject`

Mostly the structure is declared together with the typedef in the public header file but defined in an internal header file.

4.1.2 Build System

As it is widely used in the Linux world and beyond, well known and very flexible, we are using **GNU Make** as build system. For preparing the Makefiles we decided to use the **GNU Autotools Suite**[22] for all our C libraries, the tools `autoconf`, `automake` and `libtool` in particular. Their handling is not very intuitive at the first look, but with some time spent to learn and understand the concepts behind, they are mighty helpers especially concerning portability and can do a lot of work for the developer.

`autoconf`[23] is the first tool to use. It creates the well known configure script that can later be used to check a system for various dependencies before compiling the software. As input it takes the file configure.ac containing macros that are translated into shell script tests for the final configure script. configure later uses prepared files named Makefile.in in each directory to create the Makefile files used by `make` to build the sources.

`automake`[24] prepares the Makefile.in files in all directories. It takes its input from Makefile.am files that have to be prepared by hand in each build directory. Once you have understood how such a file is created it is much easier than writing a Makefile by hand. An example Makefile.am sufficient for a single binary program is given in listing 4.2.

```
bin_PROGRAMS = example

example_SOURCES = example.c pt.c tracing.c trace.c conf.c
example_CFLAGS = -I$(INC_DIR) $(HDTWLIB_CFLAGS)
example_LDADD = $(HDTWLIB_LDADD)
example_LDFLAGS = $(HDTWLIB_LDFLAGS) $(HDTWLIB_LIBS) -lpthread
```

Listing 4.2: Example for a complete Makefile.am for a single binary program

`libtool`[25] is a great helper for creating libraries. It is fully integrated into Automake and can do all the wired library packing tasks and can even build dynamical libraries for you. However, one issue with `libtool` is that when building dynamical libraries and executables that use these libraries, the built executables are replaced by wrapper scripts that do all things needed to load libraries dynamically from non standard locations (setting `LD_LIBRARY_PATH` and so on). This might by a nice idea in general, but uses to confuse for example the Eclipse IDE. By only building static libraries this can easily be avoided.
Please see an example Makefile.am for a library using `libtool` in listing 4.3

```
lib_LTLIBRARIES = libexp.la

libexp_la_SOURCES = exp.c tracing.c trace.c conf.c
libexp_la_CFLAGS = -I$(INC_DIR) $(HDTWLIB_CFLAGS)
```

Listing 4.3: Example for a complete Makefile.am for a library using `libtool`

4.1.3 Documentation System

Documentation is a very important part of programming. First of all, it helps you and others to understand the code better and faster and makes maintenance and extension of existing code much easier or even possible at all. Furthermore, it helps yourself to avoid conceptional errors while writing the code. During writing documentation you automatically reflect about what exactly you want the code to do and so identify conceptional entrapment in a time where changes are not yet very costly.

With **doxygen**[26] there exists a documentation system that can be used for source code documentation and general documentation at once. Doxygen is able to extract specially formated comments from the source code files and generate nicely formated documentation in various formats. Thus, producing API documentation is much easier than writing it by hand and even user documentation can be embedded directly into the source files and does not have to be maintained separately.

Doxygen uses a configuration file for setting up the output format and various output parameters that change the appearance of the generated documents. In addition, user specified macros can be defined in that file. Different styles of special comments can be used in the source code to specify documentation to be used by Doxygen. The style chosen is very similar to Javadoc[1], you can see an example in listing 4.4. All possible styles and tags can be found in the Doxygen Manual[27].

```
/**
 * Finalize and free a power trace
 *
 * @param trace  Power trace object
 *
 * @return  Error state
 *
 * @retval PT_SUCCESS  Success
 * @retval PT_EDEVICE  Problem during communication with device
 * @retval PT_EMEMORY  Out of memory
 */
int pt_finalizeTrace(PowerTrace *trace) {
[...]
}
```

Listing 4.4: Example for a complete Makefile.am for a library using `libtool`

[1] Javadoc Tool Homepage: `http://java.sun.com/j2se/javadoc/`

4.1.4 Library Tests

In order to test the correctness of the implementation, some tests are written that check at least the correct usage of all public functions. Testing is one of the best methods to avoid mistakes and helps a lot in debugging, too.

There is at least one test function for each function of the public API. The tests have access to the internal API so they can do checks that are not possible without. Especially, they can access the object structures directly since they know their constitution by importing the internal headers containing the structure definitions.

The sources of all tests can always be found in the tests subdirectory. Testing is fully integrated into the build system with the make target `check`.

4.2 HDTrace Writing C Library

4.2.1 Code Structure

The library can be divided into three parts, topology, tracing, and statistics. The functions of each part are declared in one header file:

Topology part	hdTopology.h
Tracing part	hdTrace.h
Statistics part	hdStats.h

The topology part is always needed, while the hdTrace and hdStats parts can be used each alone or both together. Consequently, two library archives are built, one for the tracing part (libhdTrace) and one for the statistics part (libhdStats), both including the topology objects files. For those programs using all of the HDTrace parts a complete library archive containing every part is provided as libhdTracing.

The error handling is declared globally, so we do not need to implement it multiple times in the different parts. In hdError.h all errno values are defined together with the `hdT_strerror` function and all the error strings. Of course, the error handling is included in all library archives.

4.2.2 Error Handling

In this library the errno concept is used for error handling. This is the same as in system libraries. Each public function needs to have one return state, only, indicating an error. The actual kind of error is given by the value of the thread global variable `errno`. This variable is provided by the system header errno.h. This error reporting concept is chosen for the HDTrace Writing C Library since we often have to pass object pointers back to the user calling a library function and this is more convenient by using return values than by adding an output parameter to each function's parameter list. For such functions returning a pointer, `null` is used as error indicator, for functions returning an integer value, the value -1 is used.

For each valid `errno` value that could be set by a library function a macro is defined for convenient error testing. Similar to the known system function `strerror` there is a function `hdT_strerror` to provide an string describing the error to the user.

4.2.3 API Definition

The API functions are already described in the design section 3.4 "The HDTrace Writing C Library API" on page 25. You can find the exact definition of the function signatures in the API documentation of the library in appendix A "API Documentation of the HDTrace Writing C Library" on page 79.

4.3 Power Tracer

The Power Tracer is a program as well as a library. Actually, the program uses the library and can therefore be seen as frontend providing the libraries function directly at the command line interface. It has additional access to few internal functions to be able to partially override parameters read from the configuration file by the library with command line arguments. The library is using the HDStats format to record power measurements taken from a power analyzer. For this thesis, a ZES ZIMMER LMG450 connected to the RS-232 serial port is used as power analyzer. Currently this device and connector are the only hardware supported by the library. To be prepared for the future, all device specific logic as well as the connector specific logic is individually encapsulated to make later modularization for support of other devices and connector types easy.

The library is structured in four parts:

- *Serial Port Communication* providing an internal programming interface
- *LMG Control and Communication* uses the internal *Serial Port Communication Interface* and proves another internal programming interface
- *Main Power Tracing* uses the two internal interfaces and proves the internal *Power Trace Control Interface*
- The actual public *API* uses the *Power Trace Control Interface*

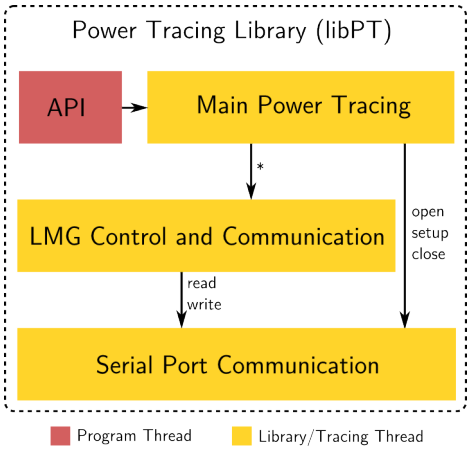


Fig. 4.1: *Structure of the Power Tracing Library Implementation*

4.3.1 Serial Port Communication

Our power analyzer is connected to the RS-232 serial interface of the host running the power tracer. With the TERMIOS API[28], the standard C library comes with support for programming serial terminals, but it is still a low level interface. The serial port can be configured in many ways and it took some time to figure out a working setup to exchange messages with the power analyzer.

First the port of the power analyzer has to be configured with the correct parameters. The values used for the LMG450 are:

ComA	Custom
ComA	8N1
Baudrate	57600
EOS	<lf>
Echo	Off
Protocol	RTS/CTS

In the absence of a real hardware RS-232 serial interface on the development computer, an USB to RS-232 adapter is used. The Linux kernel (2.6.26) handles that adapter connected to an USB port like a standard RS-232, so this should make no difference except the name of the device file which is /dev/ttyUSB0.

The code for configuring the serial port on the host side is shown in listing 4.5. Please read the comments for the description of the single instructions.

Listing 4.5: Code for setting up the serial port (w/o error handling)

```
    /*
     * Get the current options for the port...
     */
    ret = tcgetattr(fd, &options);

    /*
     * Set input flags
     *
     * IGNPAR Ignore framing errors and parity errors.
     * ICRNL  Translate  carriage  return to newline on input
     *
     * Disable all software flow control (IXON, IXOFF, IXANY)
     */
    options.c_iflag     = ( IGNPAR | ICRNL );
    options.c_iflag     &= ~( IXON | IXOFF | IXANY);

    /*
     * Set output flags
     *
     * OPOST  Enable implementation-defined output processing.
     */
    options.c_oflag     = ( OPOST );

    /*
     * Enable the receiver and set local mode...
     */
    options.c_cflag = (CLOCAL | CREAD);

    /*
     * Set data encoding to 8N1
     */
    options.c_cflag &= ~CSIZE;
    options.c_cflag |= CS8;
    options.c_cflag &= ~PARENB;
    options.c_cflag &= ~CSTOPB;

    /*
     * Enable hardware flow control
     */
#ifdef CNEW_RTSCTS
    options.c_cflag |= CNEW_RTSCTS;
#else
#ifdef CRTSCTS
    options.c_cflag |= CRTSCTS;
#endif
#endif
```

```
    /*
     * Set to raw input
     */
    options.c_lflag = 0;
    options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);

    /*
     * Set the baud rates
     */
    ret = cfsetispeed(&options, speed);
    ret = cfsetospeed(&options, speed);

    /*
     * Set the new options for the port...
     */
    ret = tcsetattr(fd, TCSANOW, &options);
}
```

After this setup, it is possible to read/write bytes from/to the serial port using the standard POSIX read/write functions. Nevertheless, the calls to the POSIX functions are encapsulated in own functions so we have a complete internal API to access the serial port providing the functions listed in table 4.1. You can find a detailed description of the functions in the Doxygen generated development documentation or in the source code itself.

`serial_openPort()`	Open serial port.
`serial_setupPort()`	Setup serial port.
`serial_resetPort()`	Reset the port. Discard all buffers.
`serial_sendMessage()`	Send a Message.
`serial_sendBreak()`	Send a BREAK.
`serial_readBytes()`	Read the next n bytes using select with timeout.
`serial_closePort()`	Close serial port.

Tab. 4.1: *Internal Serial Port Communication Interface of the Power Tracer Library*

4.3.2 LMG450 Control and Communication

The LMG450 supports full control over the serial interface using simple ASCII messages. The encoding of the messages is specified by the configured serial protocol. Additionally, the protocol specifies the character used to terminate a string, the EOS (=end of string) symbol. In our case `<lf>` is configured as EOS symbol which is the linefeed character `'\n'` used as the default end-of-line symbol in Unix.

The complete command set of the device can be found in the ZES ZIMMER Programmer's Guide[29]. Actually, one can switch between two command sets or languages that the device understands. The first one is the SCPI[2]. Since these commands are comparatively long, the ZES ZIMMER team defined an alternative language called SHORT. "The main advantage of this SHORT language is that the commands are usually just 4 bytes long. So it is very fast to parse this commands and the execution is much faster."[29, sec. 2.2.7] Excepting the command for switching to SHORT language the Power Tracer Library uses only SHORT commands.

Since we need only a small subset of the functionality provided by the LMG device, we don't need many of the commands, too. All device commands used in the library are listed with a short description in table 4.2.

`:SYSTem:LANGuage SHORT`	Set device to accept SHORT language.
`*CLS`	Reset the event register and clear the error queue.
`*RST`	Reset measuring device
`*IDN?`	Make the device send its ID string
`FRMT ASCII`	Set output format to ASCII mode
`FRMT PACKED`	Set output format to binary mode
`CYCL %f`	Define the measuring cycle length
`INIM`	Forces the instrument to always copy all measuring data into the interface buffer at the end of a cycle.
`UTRMS?;ITRMS?;P?`	Make the device sending Voltage, Currency and Power of all channels
`ACTN;%s`	Define the action script for the continous mode
`CONT ON`	Start continues mode
`CONT OFF`	Stop continues mode
`ERRALL?`	Make the device send all errors in its buffer
`GTL`	Go to local control mode (activates control with the buttons at the device)

Tab. 4.2: *Used commands to control the LMG450*

For the already mentioned modularization, all LMG specific commands are hidden behind another set of functions, the internal LMG Control and Communication Interface. These functions are listed in table 4.3.

[2]**SCPI** are the Standard Commands for Programmable Instruments specified by SCPI Consortium (`http://www.scpiconsortium.org/`)

`LMG_reset()`	Resets everything in LMG.
`LMG_setup()`	Setup the LMG.
`LMG_getIdentity()`	Get identity string from LMG.
`LMG_programAction()`	Program the LMG action script for continues mode.
`LMG_startContinuesMode()`	Start the continues mode.
`LMG_stopContinuesMode()`	Read a message in text (ASCII) format.
`LMG_readTextMessage()`	Read a message in text (ASCII) format.
`LMG_readBinaryMessage()`	Read a message in binary format.
`LMG_getAllErrors()`	Get all errors from LMG.
`LMG_close()`	Close connection to LMG.

Tab. 4.3: *LMG450 Control and Communication Interface Functions*

4.3.3 Main Power Tracing

The library starts its own thread to run the tracing in the background. The internal Power Trace Control Interface is actually a couple of switches and a conditional variable protected by a mutex as you can see in listing 4.6

```
typedef struct {
  int started : 1;
  int terminate : 1;
  pthread_cond_t cond;
  pthread_mutex_t mutex;
} threadControlStruct;
```

Listing 4.6: Internal Power Trace Control Interface for controlling the tracing thread of the library. This interface is used by the API functions.

Let `control` be our instance of `threadControlStruct`. The tracing thread runs in a loop waiting for incoming measurement data at the serial port by using `select` and checking the conditional variable `control.cond` indicating new control instructions given by one of the API functions. When data arrives a new trace entry is created and written using the HDStats API. Control instructions are handled as soon as possible, that is, immediately if the loop is not running and at the beginning of the next loop if it is currently running.

The control of the loop is the most complex but also one of the most important parts of the Power Tracer. Its control flow is shown in figure 4.2 on the following page. The

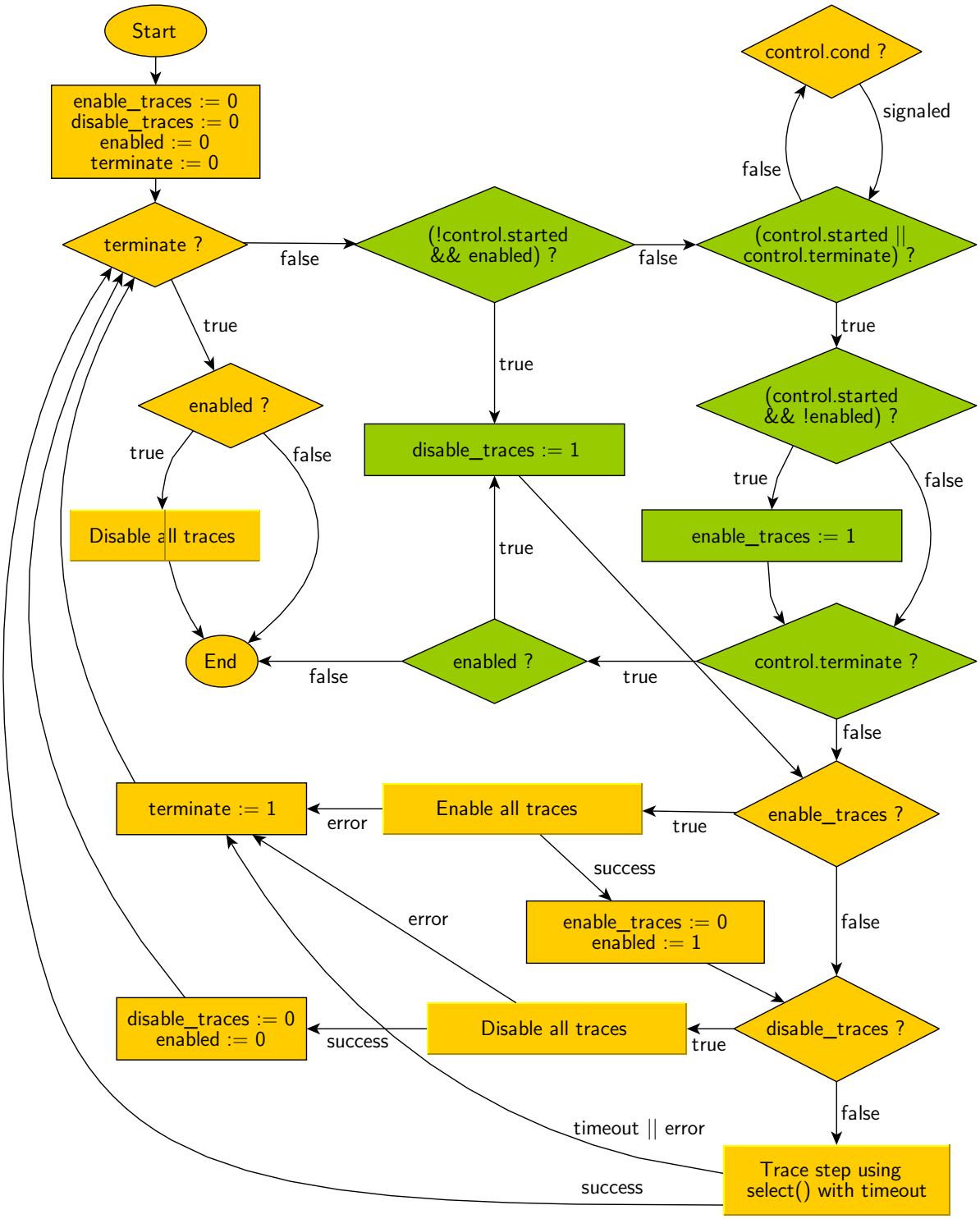
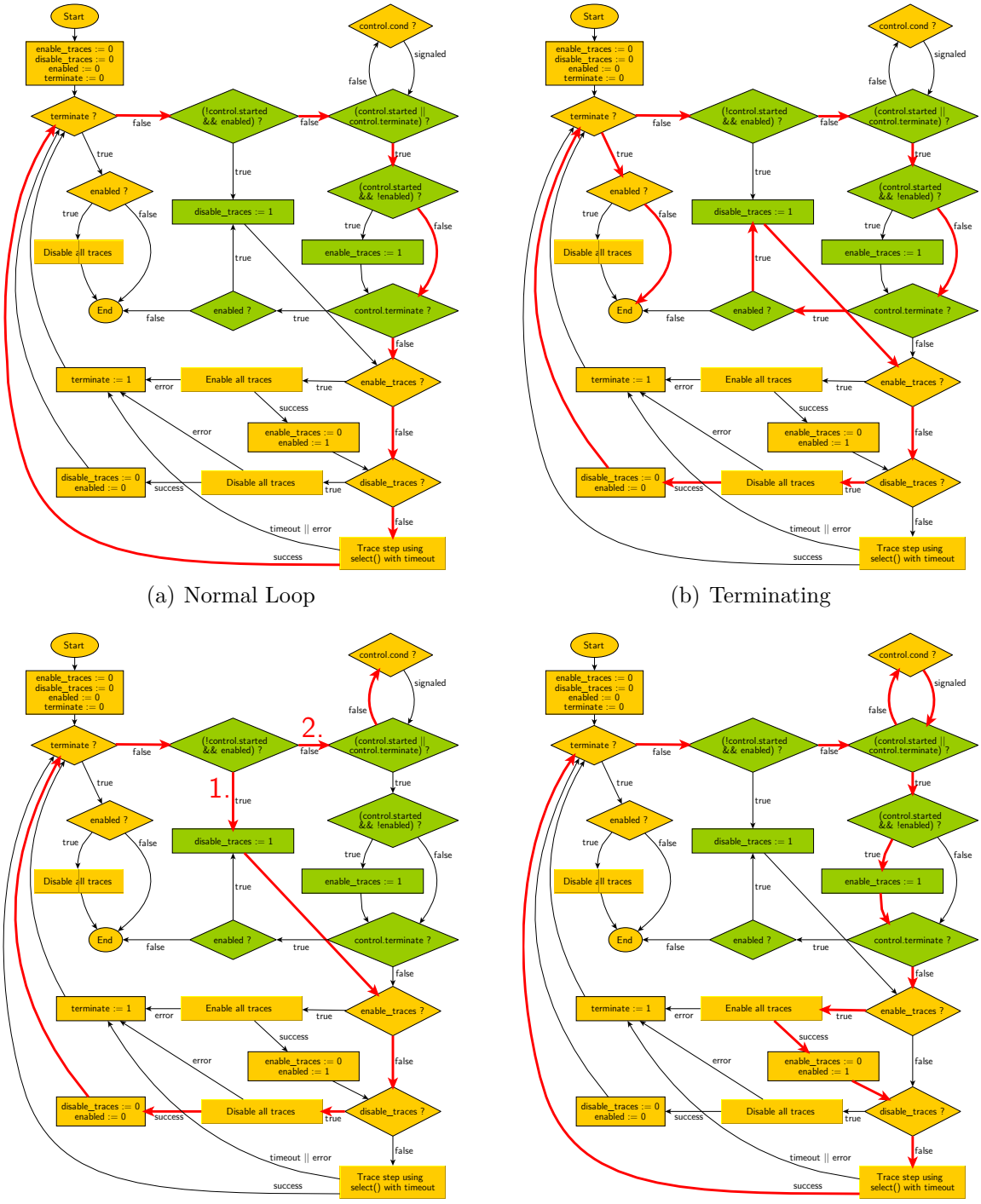


Fig. 4.2: *Controlflow of the Power Tracing Loop*



(a) Normal Loop

(b) Terminating

(c) Stop Tracing

(d) Start Tracing

Fig. 4.3: *Controlflow of the Power Tracing Loop with different paths highlighted*

flow paths of some important cases are highlighted in figure 4.3 on the previous page: 4.3(a) shows a normal tracing loop when tracing is started. This is the consecutive flow for `control.started` set. 4.3(b) shows the flow that terminates the loop when `control.terminate` becomes set. 4.3(c) shows how the tracing is stopped when `control.started` becomes unset. 4.3(d) illustrates how the tracing is started if `control.started` becomes set again.

4.3.4 The external API

In the design section (3.5.2) we have already specified the API of the Power Tracer Library (table 3.4). The user documentation with exact definition of the function signatures can be found in appendix B "API Documentation of the Power Tracer and Library" on page 95. The most detailed documentation also including internal functions and data structures is the Doxygen generated development documentation coming with the sources and the commented source code itself.

4.3.5 Error Handling

The error handling concept of the Power Tracer Library uses integer return values indicating the occurrence and kind of an error. In general, error values are negative integers. For each error there is a preprocessor macro defined to the integer return value of the error in pt.h. In the API documentation the names of all macros for errors that a function can return are given. A list of all errors reported with the return values, the corresponding macro names, and a short description is presented in table 4.4 on the next page.

The library can produce lots of debugging messages. They are structured into four levels shown in the table at the right. Which messages are printed out can be controlled using the environment variable `PT_VERBOSITY`. A message is only printed if the verbosity is greater or equal to its level.

Level	Value
DEBUG	3
INFO	2
WARN	1
ERROR	0

Macro	Value	Description
PT_SUCCESS	0	No error
PT_ESYNTAX	-1	Call syntax error (only for frontend).
PT_ECONFNOTFOUND	-2	Configuration file could not be found.
PT_ECONFINVALID	-3	Configuration read from file is invalid.
PT_EWRONGHOST	-4	Configured hostname does not match actual hostname.
PT_ENOTRACES	-5	No traces found in configuration.
PT_EMEMORY	-6	Out of memory.
PT_EHDLIB	-7	Error in HDTrace library.
PT_EDEVICE	-8	Problem during communication with measurement device.
PT_ETHREAD	-9	Cannot create tracing thread.

Tab. 4.4: *Errors reported by the Power Tracer Library*

4.4 Resources Utilization Tracing Library

The Resources Utilization Tracing Library is for recording consecutively various statistical system information as CPU, Memory, Network, and Harddisk utilization in statistics traces using the HDStats format. Those can then be analyzed together with other traces of for example function calls to gain a deeper look into what happens inside the system.

4.4.1 Getting the System Information

In order to trace the desired information we first have to acquire them. The Linux kernel provides all of them, mostly exported by the kernel as virtual text files in the proc pseudo file system. Instead of parsing the informations from these files, it would be better to use an existing library doing this for us. Beside the saved work this would have the advantage that we don't have to care about possible changes of the form the data is provided by future kernels or perhaps on different systems. Surprisingly, there are not many such libraries, but there exists one that largely fits our requirements. It is called Libgtop[30] and is part of the GNOME project[3]. Libgtop uses Glib[31], a library designed to make platform independent development in C more comfortable and also

[3]GNOME: The Free Software Desktop Project (`http://www.gnome.org/`)

part of the GNOME project. For consistency, the Glib functions and concepts are used for the whole RUT library. The library versions used during development are Glib 2.16.6 and Libgtop 2.22.3.

LibGtop provides some structures for different system components. By using access functions the user can fill these structures with the current values. Some of them contain absolute values, as for example the current memory used, but most of them are only counters and leave it up to the user to build differences if the value for just an interval is needed. For example, the network traffic is just counted in bytes since the interface came up.

CPU Utilization

For each CPU in the system Libgtop provides the time the CPU is idle since the last system start in jiffies. A jiffy is a system dependent small time unit which is normally 1/100th of a second. Fortunately the actual length of a jiffy does not matter, since we want to trace the CPU utilization in percent.

We calculate the CPU utilization in the time interval $]t_{i-1}, t_i]$ with the following formula where $x_{idle}(t_i)$ is the value of the idle counter in jiffies for the CPU at time t_i and $x_{total}(t_i)$ is the total number of jiffies since the system start at time t_i. Libgtop provides $x_{total}(t_i)$, too.

$$u_{CPU}(]t_{i-1}, t_i]) = 1 - \frac{x_{CPU}(t_i) - x_{CPU}(t_{i-1})}{x_{total}(t_i) - x_{total}(t_{i-1})}$$

$u_{CPU}(]t_{i-1}, t_i])$ is the fraction of the time interval $]t_{i-1}, t_i]$ the CPU is not idle, also called utilization. Using that formula for each single CPU and the CPUs aggregated we can record all desired CPU utilization values.

There is a weakness with this concept, when one of the counters overflow in $]t_{i-1}, t_i]$ we will get a wrong value for $u_{CPU}(]t_{i-1}, t_i])$. The counters are provided by Libgtop in 64 bit unsigned integers. But even if we assume, that the system itself only provide the values in 32 bit unsigned integers, the system can run about 500 days before the total counter overflows the first time. Beside the total counter, we use one idle counter for each CPU and the aggregated CPU idle counter. The last-mentioned is calculated by Libgtop and should therefore have true 64 bit. Hence, we use #CPUs counters that could overflow between the definite overflows of the total counter about every 500 days. Performing an overflow protection within each iteration in order to avoid one wrong value occurring that rarely would be disproportional.

Another problem arises, if holding $x_{total}(t_i) = x_{total}(t_{i-1})$. This can happen, if the time period $t_i - t_{i-1}$ is shorter than one jiffy. Here we have a similar situation as above,

to eliminate this problem would cost time in each iteration but we would only benefit if the time period is configured that short that already no meaningful values can be expected. Since the length of one jiffy is usually $\leq 10ms$ [32] we define that as the shortest supported period time for the Resource Utilization Tracing Library.

Memory Utilization

The memory utilization is provided by Libgtop in bytes currently used, free, cached, buffered, shared. So we can just write them into the trace as they are. The only thing the user have to keep in mind is that each tracing entry is just a snapshot, ignoring all changes between the reading points.

Network Utilization

Network utilization again is provided very similar to the CPU idle counting. Here, the bytes transfered in and out are counted separately . We can get only values for each single network interface, no aggregated values are provided, so we have to calculate them by ourself. For usability considerations we calculate not only the aggregated in and out traffic for all network interfaces but also those for the external interfaces only, that is, without local loopback traffic.

The calculation is simpler than for CPU tracing since we want to trace the data in the provided unit directly. For the calculation of network utilization we come to the following formula:

$$in_{NIC}(]t_{i-1}, t_i]) = in_{NIC}(t_i) - in_{NIC}(t_{i-1})$$
$$out_{NIC}(]t_{i-1}, t_i]) = out_{NIC}(t_i) - out_{NIC}(t_{i-1})$$

Harddisk Utilization

The support of hard disk drive read/write amount in the Libgtop is based on mount points. You can get the read and write data only for a mount point, not for a physical device. During development, it turned out to be very complicated to get all mount points and especially to decide which of them are from local disks and which are for example NFS mounts that should be excluded when tracing hard disk drives. Therefore, the user has to configure the mount point to trace using the environment variable `RUT_HDD_MOUNTPOINT`. In most cases this should not be a restriction since the user knows where the data, he is interested in, are read from or written to.

The values are provided in the same fashion as for the network. Consequently, the calculation is entirely the same:

$$read_{HDD}(]t_{i-1}, t_i]) = read_{HDD}(t_i) - read_{HDD}(t_{i-1})$$
$$write_{HDD}(]t_{i-1}, t_i]) = write_{HDD}(t_i) - write_{HDD}(t_{i-1})$$

Efficiency

Of course the library asks only these values that are needed to perform the tracing as configured. For example if the user has chosen not to trace any network statistics at all, no network values are requested by the utilization library from Libgtop.

4.4.2 Tracing control

The tracing control is done exactly in the same way as in the Power Tracer (see section 4.3.3 "Main Power Tracing" on page 45). The API functions start an own tracing thread and control it using two switches `started` and `terminate` protected by a conditional variable and a mutex. The difference is the time base of the tracing loop. In the Power Tracer, the power analyzer is programmed to send data periodically in fix intervals and so only a `select` is called to wait for the next data arriving.

```
/* create timer */
gulong currentTime, waitTime;
GTimer *timer = g_timer_new();

[...] /* thread control variables */

while(1)
{
  [...] /* thread control */

  /* wait for next multiple of interval time */
  currentTime = (gulong) (1000.0 * g_timer_elapsed(timer, NULL));

  waitTime = (gulong) tracingData->interval
      - (currentTime % (gulong) tracingData->interval);

  g_usleep (waitTime * 1000);

  /*  do tracing step */
  doTracingStep(tracingData);
}

/* free timer */
g_timer_destroy(timer);
```

Listing 4.7: Timer for controlling the tracing intervals in RUT library

For the resource tracing we rather following a polling strategy and have to take care about the intervals ourself. Fortunately, the Glib provides a nice timer interface. At the beginning of each step, the wait time until the next tracing step time is calculated. The thread then waits until this time has passed before going on with the operation. Tracing step times are all multiples of the defined time interval since the timer initialization, performed only once before entering the loop. Listing 4.7 on the previous page contains a code snippet of the timer controlled loop.

4.4.3 Error Handling

The error handling concept is exactly the same as of the Power Tracer Library (see section 4.3.5 on page 48). The errors reported by the Resources Utilization Tracing Library are listed in table 4.5. The environment variable to control the verbosity is `RUT_VERBOSITY`.

Macro	Value	Description
`RUT_SUCCESS`	0	No error
`RUT_EMEMORY`	-1	Out of memory.
`RUT_EHDLIB`	-2	Error in HDTrace library.
`RUT_ETHREAD`	-3	Cannot create tracing thread.

Tab. 4.5: *Errors reported by the Resources Utilization Tracing Library*

4.4.4 API Definition

The API functions have already been described in the design section 3.5.1 “Resources Utilization Tracing Library API” on page 30. You can find the exact definition of the function signatures in the API documentation of the library in appendix C “API Documentation of the Resources Utilization Tracing Library” on page 100. Again, the most detailed documentation also including internal functions and data structures is the Doxygen generated development documentation coming with the sources and the commented source code itself.

The implementation work is done an we now have a working environment for tracing power and resource utilization statistics. Next the libraries are evaluated to show the efficiency of the implementations.

5 Evaluation

In this chapter the library implementation is evaluated. Several measurements are made to analyze the correctness and efficiency of the Power Tracer Library, the Resources Utilization Tracing Library, and implicitly the HDStats writing library.

5.1 Visual Inspection for Correctness

Now that we have a complete implementation of the environment for tracing power consumption and resources utilization, it has to be verified that the entire environment is working correctly. We will not formally proof that but do a visual inspection.

5.1.1 PowerTracer

Even a visual inspection of the traces created by the power tracer is not easy to do. First of all we have to rely on the correctness of the data sent by the correctly programmed power analyzer. Fortunately there is a display at the front of the power analyzer (see figure 5.1 on the right) which can show the currently measured values independently from what happens on the serial interface. That means it is possible to just print the data received from the device via the serial connection to the console and manually compare them with the values displayed at the power analyzer. Several such tests revealed that the values are as expected. Hence, we can assume the programming of the device is done correctly.

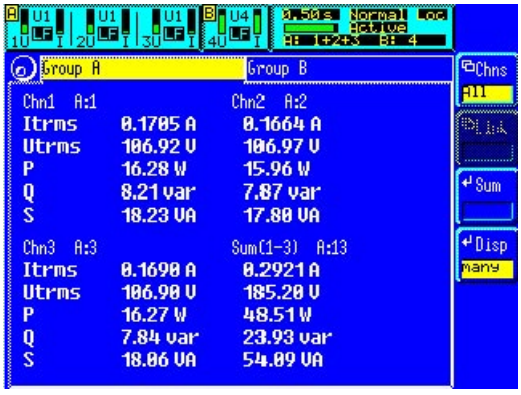


Fig. 5.1: *Display screenshot of the ZES ZIMMER LMG450* [33]

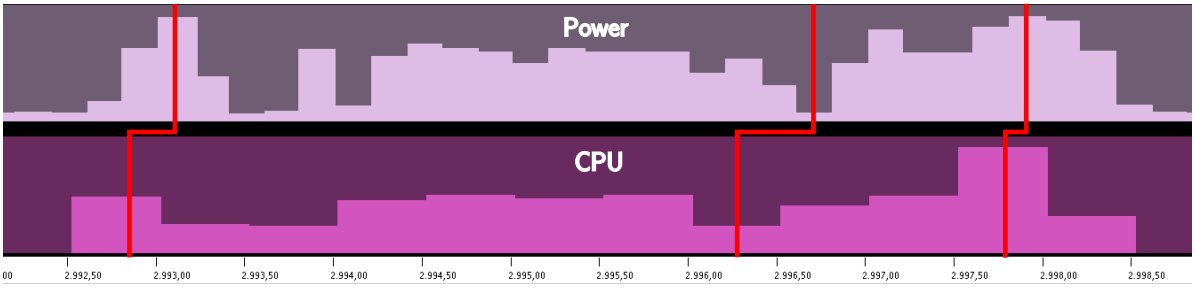


Fig. 5.2: *Delay of the power trace*

At first glance, when looking at the traces, you will see that the power consumption is very dependent on the CPU utilization. When taking a closer look, you will realize that the power consumption is often a bit delayed compared to the CPU utilization. In other words, the power graph lags slightly behind the CPU graph. This phenomenon is pointed up by the drawn lines in figure 5.2. The time of the delay is fluctuating from several milliseconds to half a second. One reason might be the transfer time of the data from the power analyzer to the Power Tracer. Another reason could be the internal buffer of the power analyzer. The data sent are those from the last measuring cycle, not from the current one. Furthermore, there could be some technical reasons, for example the attenuation of the power supply unit. Further investigations in this field should be clearly the subject of a future work.

5.1.2 Resources Utilization Tracing

To get an impression whether the utilization tracing is working correctly, we can trace a simple program that stresses different resources of a system for defined time intervals and intensity. The resulting trace should show the expected resources utilization.

The flow of the program we use is the following:

- $0s$ start stressing first CPU with 100%
- $10s$ start stressing second CPU with 100%
- $20s$ stop stressing first CPU
- $30s$ stop stressing second CPU
- $40s$ allocate 512 MB of memory
- $50s$ allocate another 256 MB of memory
- $60s$ free all memory
- $70s$ start stressing external network interface by sending 512 MB

80*s* start stressing external network interface by receiving 512 MB

90*s* start stressing local hard disk drive by writing 512 MB

100*s* allocating all memory for cleaning cached data

110*s* start stressing local hard disk drive by reading 512 MB

120*s* terminate program

Since there are standard shell programs capable of producing most of the desired effects, it is the easiest way to implement the program as a shell script. Our script is named verifyRUT.sh. It uses the programs awk, cmalloc[1], nc, and dd successively in order to stress the CPU, memory, network and harddisk. You can find the script's code in appendix D on page 107.

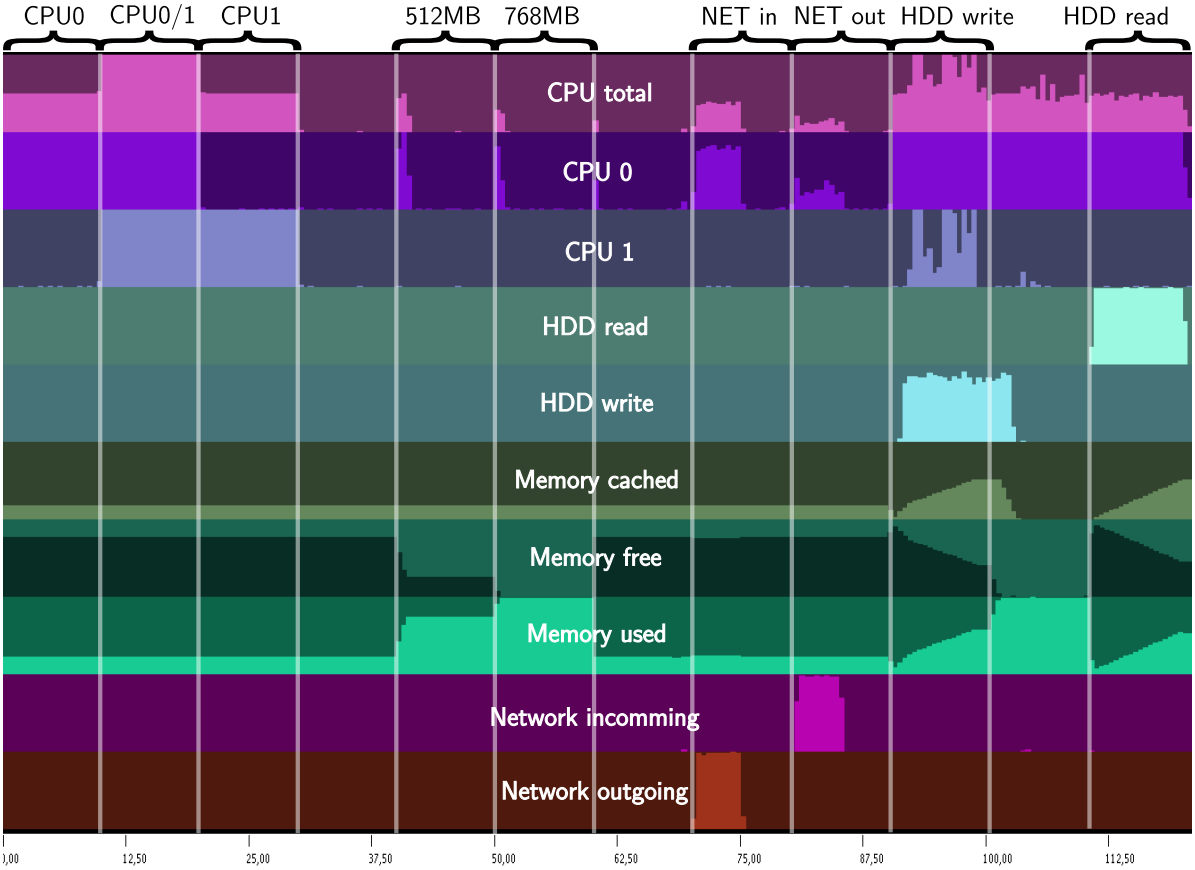


Fig. 5.3: *Utilization Trace of a verifyRUT.sh run*

Please have a look at the trace in figure 5.3 showing the utilization trace of a run of verifyRUT.sh on one node of the PVS Cluster (see figure 5.4 on the next page). As you can see, the trace shows every resource utilization as expected. Note, it is absolutely

[1] cmalloc is a small C program just allocating a given amount of memory

plausible that the CPU shows some utilization also while the other components are stressed. As well, the free memory is reduced while writing to the disk due to the operating systems data cache becomes filled. Actually we can recognize, that the disk write operation could not finish within the estimated 10 seconds so it goes into the cache cleaning phase.

With Sunshot, also the correctness of the data values for the network and hard disk usage could be verified. In conclusion the utilization library seems to work as desired.

The PVS Cluster
PVS Cluster is the research cluster in our group. It consists of one control node named *master* and nine worker nodes named *node01* to *node09*. Five of the worker nodes (*node01-node05*) are also I/O test nodes having a two-disk raid system each. All worker nodes do have a local hard disk but they usually are running their operating system via NFSv3. The user home directories are mounted using NFSv3, too.
For this thesis, only the non-I/O nodes *node06* to *node09* are used. They have the following configuration each:

- CPUs: 2x Intel XEON (Prestonia) 2GHz, 512KB Cache (HT disabled)
- Memory: 1GB RAM / 3GB swap space
- Network: 1Gbit Ethernet
- Harddisk: P-ATA UDMA100 (mounted to /tmp and used as swap space)
- Operating System: Ubuntu Linux 8.04 (SMP-Kernel 2.6.30, PXE/NFS)

Fig. 5.4: *Hardware/Software configuration of the PVS Research Cluster*

5.2 Performance Impact by Tracing

One of the problems with measurements in general is the influence of the measuring procedure to the measured system and thus to the measurement itself. In our context, this means the tracing produces additional utilization of resources that would not happen if we would not trace. Therefore, it is very important to keep the influence to the measured system very small, in our case to do the tracing very efficient and produce as little additional utilization as possible. While writing the libraries, this fact had always been in mind and efficient programming was done whenever possible. Nevertheless, an evaluation of the results is necessary to make a qualitative statement.

It is not possible to see the impact of the tracing directly since we cannot get the power consumption and resources utilization of a non-tracing program run without tracing them. Instead we can make an indirect test by comparing the runtime of a resource intensive program with and without tracing.

Most likely, tracing will have its highest impact on the CPU, so we focus here on this bottleneck. The tests are made using two programs, the well known HPCC benchmark and a simple self-written synthetic benchmarks called CPUstress. The HPCC benchmark simulates typical load situations for parallel programs. CPUstress just pushes the CPU utilization to 100% all the time by counting up the loop variables in two nested `for` loops.

We use the PVS Cluster described in figure 5.4 on the preceding page for all of our tests. The exact configuration of the HPCC benchmark is the same as used later in section 6.2 "HPCC Benchmark" on page 70 and is briefly described there.

5.2.1 Power Tracer

For the power tracing, it holds: Higher utilization of the resources produces higher power consumption of the whole system. Even though, the described problem is not so important for our Power Tracer since its design allows to put it on a separate host and so eliminate all direct impact to the measured systems.

	no tracing	**RUT500ms**	**RUT500ms/PT200ms**
Run 1	599 s	601 s	599 s
Run 2	597 s	603 s	602 s
Run 3	602 s	603 s	603 s
Run 4	589 s	593 s	613 s
Run 5	592 s	597 s	596 s
$\oslash$	595, 8 s	599, 4 s	602, 6 s
σ	5, 26 s	4, 34 s	6, 43 s
c_v	0.88%	0.72%	1.07%
Impact	–	0.60%	1.14%

Tab. 5.1: *Runtime comparison of the HPCC benchmark with or without utilization and power tracing.* Five runs are shown with arithmetic mean ($\oslash$), standard deviation (σ), coefficient of variation (c_v), and the relative tracing impact.

Yet, in some cases it might be important as we also provide the Power Tracer Library to be used directly. Hence, we make one test to get an idea of the libraries performance impact. In table 5.1 on the preceding page you can see the runtime values of the HPCC benchmark. The first column is without tracing. The second column is with only utilization tracing using a moderate tracing interval of 500 ms. The last column is with additional power tracing in the same node using the Power Tracer Library directly with a tracing interval of 200 ms.

You can see that tracing with the interval lengths used here produces only a very small performance impact of only about one percent. Therefore we can mark the libraries as usable. Even if the used power device supports measuring cycles down to a length of 50 ms, using such a small interval makes not much sense as long as the technical problems discovered and described in section 5.1.1 are not yet resolved. The influence of the interval length used for utilization tracing is analyzed in the next section.

5.2.2 Resources Utilization Tracing

To analyze the influence of the tracing interval length on the performance impact of the utilization tracing the HPCC benchmark is traced with a few different interval lengths. The results are listed in table 5.2.

Interval	**no tracing**	**500 ms**	**100 ms**	**20 ms**
Run 1	599 s	601 s	609 s	660 s
Run 2	597 s	603 s	613 s	655 s
Run 3	602 s	603 s	613 s	660 s
Run 4	589 s	593 s	607 s	657 s
Run 5	592 s	597 s	609 s	655 s
$\oslash$	595.8 s	599.4 s	610.2 s	657.4 s
σ	5.26 s	4.34 s	2.68 s	2.51 s
c_v	0.88%	0.72%	0.44%	0.38%
Impact	–	0.60%	2.42%	10.34%

Tab. 5.2: *Runtime comparison of the HPCC benchmark with or without utilization tracing using different tracing intervals.* Five runs are shown with arithmetic mean ($\oslash$), standard deviation (σ), coefficient of variation (c_v) and the relative tracing impact.

We can see a strong influence of the tracing interval to the performance impact. The 2.42% with 100 ms steps are just about acceptable but more than 10.34% is not acceptable at all. So we can make the guess that when tracing all values provided, with the current implementation of libRUT the tracing interval should not be smaller than 100 ms. To verify the guess, we make another test using the described CPUstress program. To get a more detailed picture we increase the number of tested intervals. Please take a look at the results in table 5.3 and in figure 5.5.

Interval	**no tracing**	**500 ms**	**200 ms**	**100 ms**	**50 ms**	**20 ms**	**10 ms**
Run 1	402.2 s	406.2 s	405.5 s	408.6 s	415.6 s	436.3 s	476.4 s
Run 2	402.5 s	405.5 s	408.0 s	408.6 s	415.3 s	436.2 s	476.3 s
Run 3	402.1 s	404.0 s	405.7 s	408.9 s	415.5 s	436.2 s	475.8 s
Run 4	402.0 s	403.8 s	405.9 s	408.5 s	415.4 s	436.1 s	475.8 s
Run 5	402.3 s	404.2 s	405.6 s	409.2 s	415.4 s	435.8 s	476.1 s
$\oslash$	402.2 s	404.7 s	406.1 s	408.8 s	415.4 s	436.1 s	476.1 s
σ	0.21 s	1.07 s	1.05 s	0.27 s	0.14 s	0.20 s	0.30 s
c_v	0.05%	0.26%	0.26%	0.07%	0.03%	0.04%	0.06%
Impact	0.00%	0.63%	0.97%	1.62%	3.29%	8.43%	18.36%

Tab. 5.3: *Runtime comparison of the CPUstress benchmark with or without utilization tracing using different tracing intervals.* Five runs are shown with arithmetic mean ($\oslash$), standard deviation (σ), coefficient of variation (c_v) and the relative tracing impact.

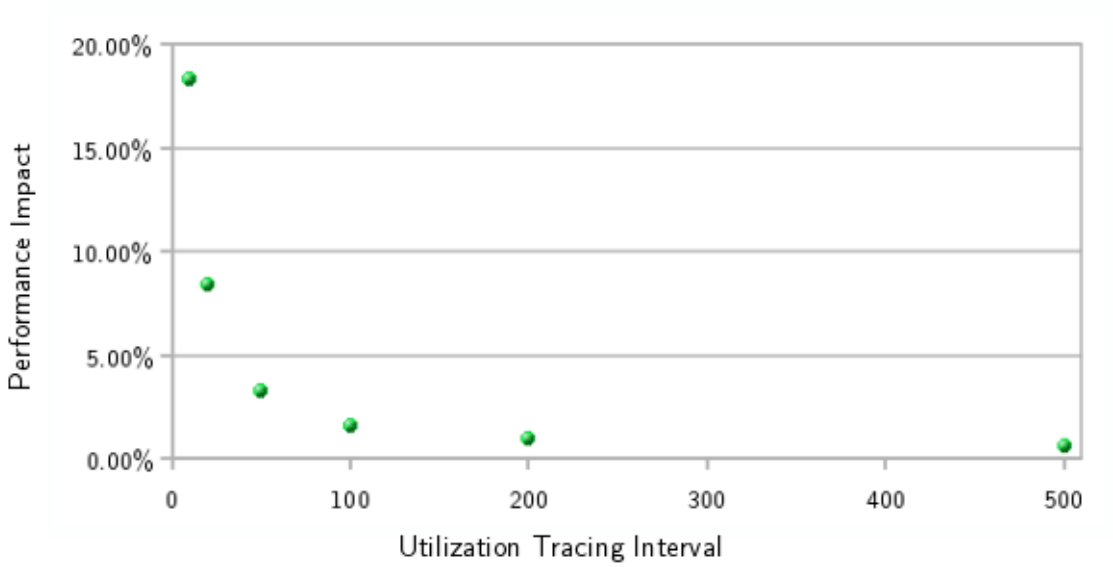


Fig. 5.5: *Mean value diagram of runtime comparison of the CPUstress benchmark* (compare table 5.3)

Here as well, we can identify 100 ms as a threshold to the reasonable interval length. Decreasing the interval length below this value results in a highly increased performance impact. In general we instruct the user not to choose an interval smaller than necessary for the desired purpose to keep the performance impact caused by the tracing as small as possible. This is a wise advice from another point of view, too: The smaller the tracing interval is chosen, the larger the traces will be.

During the evaluation, be found the library working correctly and efficiently to a satisfactory degree. Solely, the tracing intervals should not be chosen to short. Anyway, the environment is ready for analyzing some benchmarks in the next chapter.

6 Analyzing Existing Benchmarks

In this Chapter the developed libraries are used to do some first investigation of existing and popular benchmarks. We take a look at the SPECpower benchmark as one of very few power benchmarks already established today and at the HPCC benchmark suite as a widely accepted benchmark in the field of high performance computing.

6.1 SPECpower_ssj2008 Benchmark

The SPECpower_ssj2008 benchmark is developed by the Standard Performance Evaluation Corporation (SPEC)[1]. "SPECpower_ssj2008 is the first industry-standard SPEC benchmark that evaluates the power and performance characteristics of volume server class and multi-node class computers"[34]. Unfortunately support for multi-node class computers that could have been of high interest in the subject of this thesis was added initially in version 1.10. The version available for the tests was only 1.00. Hence, everything following is related to this now obsolete version 1.00.

6.1.1 Description of the Benchmark

SPECpower_ssj2008 consists of three logical components, the Workload (SSJ), the power and temperature daemon (PTD) and the control and collect system (CCS). "These modules work together in real-time to collect server power consumption and performance data by exercising the server under test (SUT) with a predefined workload"[35]. The base of the benchmark is the assumption that "power consumption varies as a function of [transactions] throughput"[35].

[1] The SPEC "is a non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers." It is well known as the creator of several performance benchmark suites."[`http://www.spec.org/`]

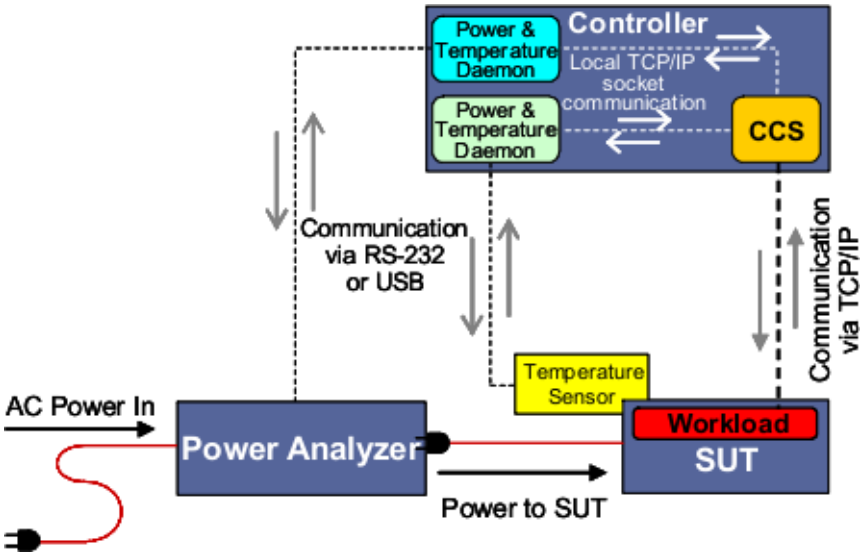


Fig. 6.1: *SPECpower_ssj2008 scheme diagram* [36, page 5]

Workload (SSJ)

As the name already implies, the component called workload is the part responsible for generating the workload on the server under test. "It is a Java implementation simulating a standard three-tier client-server architecture designed to exercise several components of the SUT"[35]. The three tiers are the client, the businessmodel and the warehouse. Virtually the client sends requests for transactions to the warehouse using the businessmodel and such provokes load on the server under test. It is supported to run the benchmark with more than one warehouse instance, but in the standard configuration that is used for the tests, there is only one warehouse and one client. Therefore having multiple instances is not regarded here.

Power and Temperature Daemon (PTD)

The power and temperature daemon is the only part not written in Java. Is is responsible for getting the measurement values from the power analyzer and temperature sensors connected via USB oder RS-232 serial bus. For testing, it comes with two dummy devices, one for power and one for temperature. By using these dummy devices, it is possible to run the benchmark without having a power analyzer or temperature sensors connected. The daemon has to run two times, once for the power and once for the temperature. For our purpose the environment conditions don't matter for which reason we always use the temperature daemon in dummy mode.

Control and Collect System (CCS)

The control and collect system is written in Java, too. It is connected to all three other components via TCP, so each of the components could be on different systems as long as a TCP network connection is available between them. The CCS starts the benchmark and collects the data from the other three components in real-time. At the end of the run it generates the output for the user.

Workflow of the Benchmark

The concept of the SPECpower_ssj2008 benchmark is to generate different stress levels on the SUT and measure the average power consumption on each level. Therefore, the benchmark is running in several phases. Each phase is driven with a preferably constant stress level. The single phases are always divided by short intermediate idle times.

The workload to stress the SUT is produced by performing operations in the virtual warehouse. As first step of each run, a number of calibration phases are performed. In those, the SUT is driven to the highest stress level possible and the CPU utilization is measured as well as the operations throughput the server could handle. These values are set to be 100% performance and all target stress levels are calculated on that base. The assumption made is that the utilization is straight proportional to number of operations performed in the virtual warehouse during a fix time interval. So the benchmark controls the stress levels by controlling the number of operations.

The number of calibration phases, the stress levels benchmarked, and the duration of the levels are all configurable. We use here the standard configuration, that is three calibration phases with the first ignored to get rid of warmup effects, 11 stress levels (100% to idle in 10% steps) and a duration of 240 seconds for each phase. The intermediate idle time is 10 seconds.

6.1.2 Test Environment

The Benchmark on its Own

In the first test described here, the recommended setup for the benchmark is used. One of the nodes of our cluster is the server under test which only runs the SSJ workload program. Another node is used as CCS, running also the two instances of PTD. The temperature daemon is running in dummy mode, while the power daemon is connected to the power analyzer where all the energy consumed by the SUT node is measured.

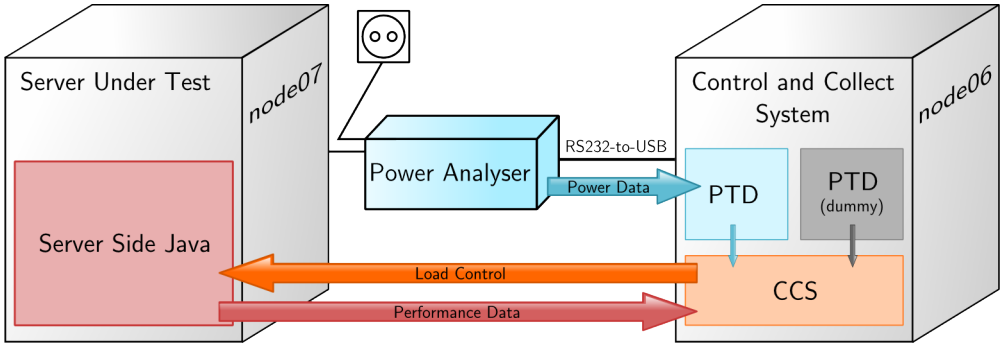


Fig. 6.2: *Schematic diagram of the SPECpower_ssj2008 test setup without our environment*

The SUT and the CCS are both identical nodes in the PVS Cluster described in figure 5.4 on page 57. The SUT is running on `node08` and the CCS on `node06`. The power supply of `node08` is connected to the first channel of a ZES ZIMMER LMG450 power analyzer. The data interface of the power analyzer is connected to the USB port of `node06` using an USB-to-RS232 adapter. You can see a schematic diagram of the test setup in figure 6.2

Using Our Environment

The second test uses our own environment. The setup is illustrated in figure 6.3. The CCS now is running on `node07` with both instances of the PTD in dummy mode. The power analyzer is still connected to `node06`.The power analyzer is still connected to

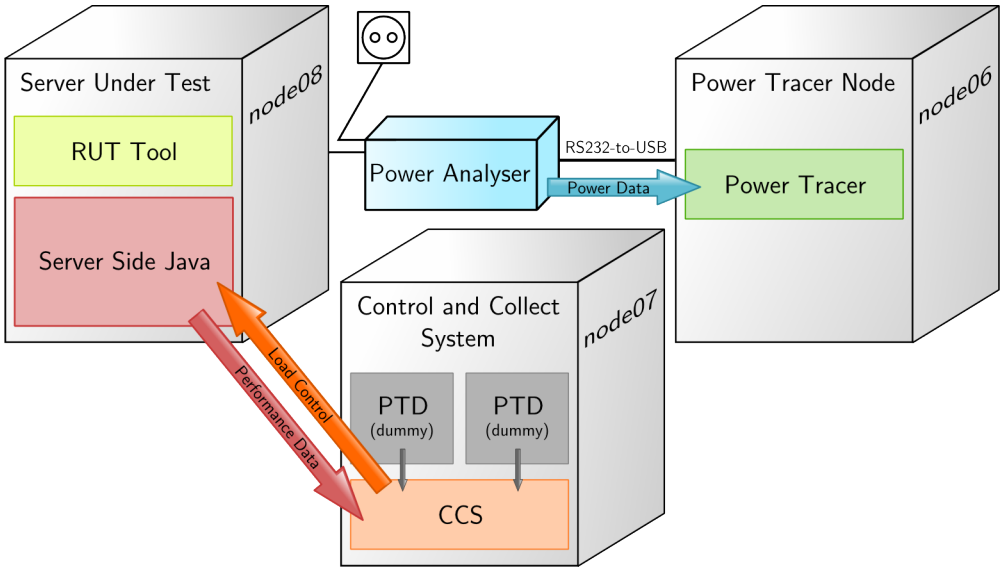


Fig. 6.3: *Schematic diagram of the SPECpower_ssj2008 test setup with our environment*

node06. It is only running our Power Tracer for tracing the power of node08 which is again the SUT. Additionally, on node08 a little tool is running using libRUT for tracing the resources utilization on the SUT. The power tracing is performed with 200 ms. The utilization tracing is using a 500 ms interval time for minimizing the influence to the SUT as much as reasonably possible.

6.1.3 Test Results and Interpretation

The Benchmark on its Own

The first results are those of a normal SPECpower_ssj2008 benchmark run using its default configuration with the test setup presented in figure 6.2 on the previous page. The main output of the run you can see in figure 6.4. It is a table showing the ten load steps with the target and the actual load. If the difference in only one step extends a limit (+2%, −2.5% for the 100% and 90% target loads, ±2% for the 80% though 10% target loads), the entire run is declared as invalid. Invalidity of a benchmark run mainly denies the acceptance of submission to the official SPEC results database and is of no high interest for us. Actually we cannot get a valid run since we don't have an environment temperature sensor connected to the system. The other values in the table are the power measurements and the performance to power ratio, both shown in the diagram, too.

Performance			Power	Performance to Power Ratio
Target Load	Actual Load	ssj_ops	Average Power (W)	
100%	99,8%	7.424	189	39,2
90%	92,1%	6.851	185	37,0
80%	80,0%	5.951	178	33,4
70%	69,4%	5.161	172	30,1
60%	62,2%	4.627	167	27,7
50%	49,9%	3.715	159	23,4
40%	41,2%	3.064	153	20,0
30%	29,3%	2.181	145	15,0
20%	20,0%	1.487	139	10,7
10%	9,9%	733	132	5,56
Active Idle		0	125	0
		∑ssj_ops / ∑power =		23,6

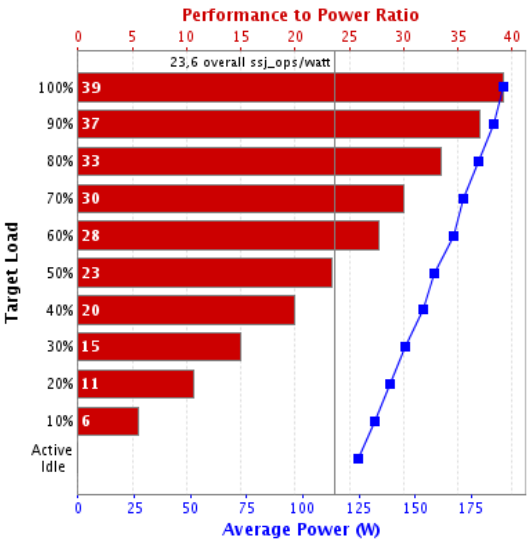


Fig. 6.4: *Results of a standard run of SPECpower_ssj2008 1.00 with power analyzer connected* The server under test is one of the PVS Cluster nodes.

The performance to power ratio is a metric indicating how power efficient the system is

working at a specific load level. The higher this value is, the more performance you will get per watt[2] consumed. It is not surprising, that the best ratio is achieved at 100% load. There is a base amount of power that you always need for having the system running, primarily idling. Every piece of load you put on the system will decrease the relative portion of the base energy requirement and so increase the performance to power ratio. The unit for the ratio is *Performance per Watt* and SPEC uses its Server Side Java operations (ssj ops) as unit for the performance.

Using Our Environment

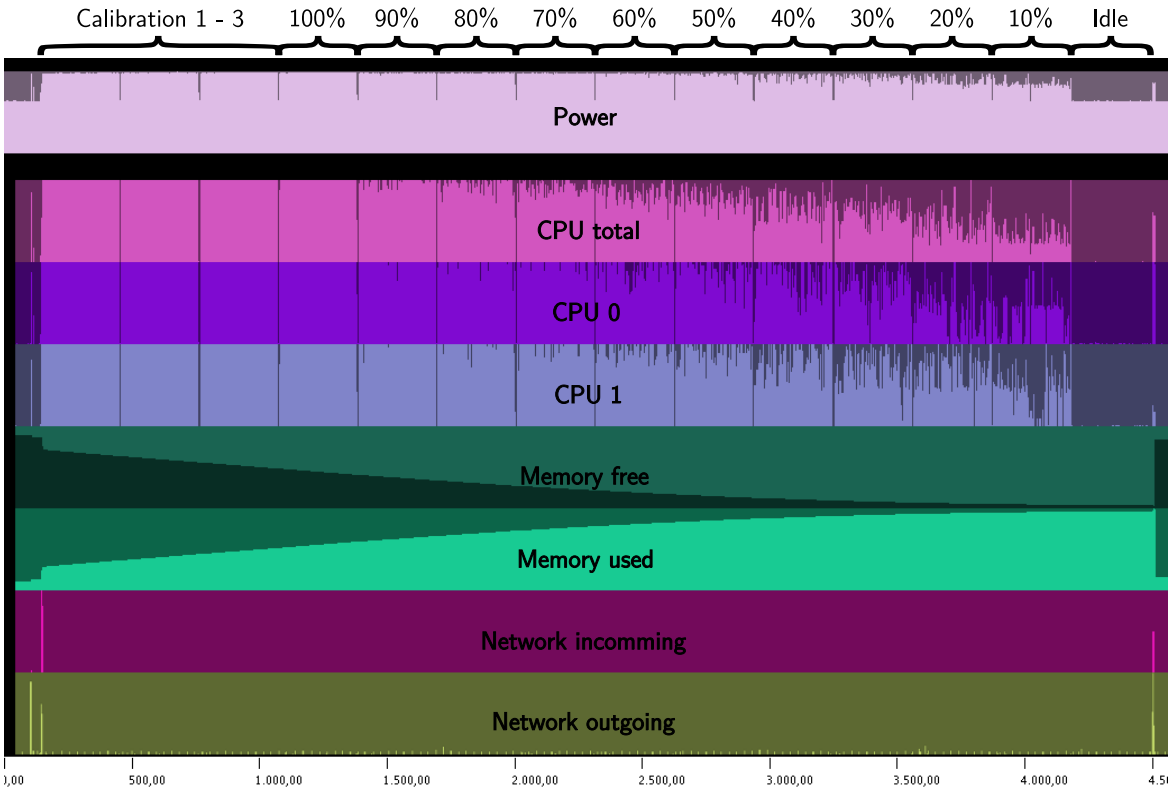
Now we try to verify the results presented by the benchmark itself using our new tracing environment in the setup illustrated in figure 6.3 on page 65. The SPEC's power daemon is put into dummy mode. Instead the power analyzer is used by our Power Tracer, running on its own node, to trace the power consumption of the server under test. A small standalone program using libRUT traces the utilization on the SUT node.

Performance		
Target Load	Actual Load	ssj_ops
100%	100,4%	7.453
90%	87,4%	6.483
80%	80,2%	5.952
70%	70,1%	5.203
60%	59,2%	4.392
50%	49,8%	3.696
40%	39,2%	2.911
30%	31,7%	2.351
20%	19,5%	1.450
10%	10,6%	787
	Active Idle	0

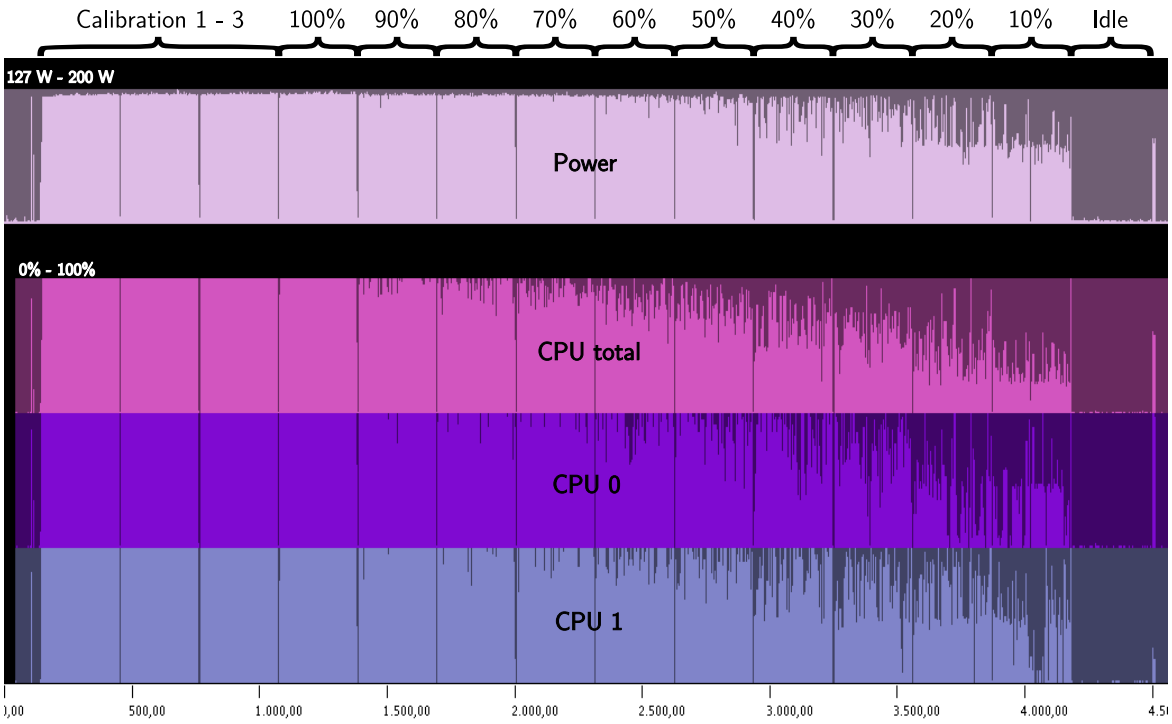
Fig. 6.5: *Results of a standard run of SPECpower_ssj2008 without power analyzer connected* The power analyzer is used by the Power Tracer. The server under test is one of the PVS Cluster nodes.

In figure 6.5 you can see the benchmark's own output of the traced run. The visualization of the trace project containing power and utilization trace is represented in figure 6.6(a) on the next page. Due to the idle phases separating the consecutive load phases they can be easily identified. In the figure they are additionally labeled. Beside the different CPU load intervals, we can also identify a constantly increasing memory usage that we cannot explain at this time. At the beginning an finalizing of the benchmark, some short two way network activity is observable. Presumably this is the setup of the SUT at the beginning and the collection of the final results at the end. Some low outgoing network activity we can see during the whole runtime which we assume to be the real-time data collection by the CCS mentioned in the user manual.

[2]Watt [W] is the SI unit for electrical power



(a) Overview



(b) CPUs and power in detail

Fig. 6.6: *Power and utilization trace of a of a standard SPECpower_ssj2008 run*

Since we are especially interested in the relation of the CPU load and the consumed power, we now take a closer look to this aspect using figure 6.6(b) on the preceding page showing the same trace zoomed in and having only the CPU and power lines visible. We can see that the CPU load in the single intervals is far away from being constant. For getting a nice visual stairs effect, perhaps larger tracing intervals could help or the viewer had to support ad hoc merging of adjacent values to smoothen the jitters.

But the Sunshot viewer has another useful feature already implemented. It is capable to calculate the arithmetical mean of a specified interval, therefore we can get the mean values of each load phase. They are listed in table 6.1. In the **scaled** column here are the achieved CPU utilization levels scaled to the highest measured level, that is the achieved utilization for 100% target. This is how the benchmark calculates its actual performance levels. **error** is the difference from the archived scaled level to the desired level. We compare these values with those reported by the benchmark (see figure 6.5 on page 67) and find some significant differences. While the actual load values reported by the benchmark are all together very close to the target values with a maximum error of 7.9% and an average error of 0.79%, the values calculated from our trace show differences from actual to target load up to 4% and an average error of 1.45%. This discrepancy indicates the need of more tests. Unfortunately the time with access to the power meter was not long enough for further testing, so we have to delay that to a another opportunity.

target	CPU	CPU0	CPU1	Power	scaled	error
100.00%	93.70%	93.90%	93.50%	193.9 W	100.00%	0.00%
90.00%	80.50%	79.40%	81.60%	186.2 W	85.91%	−4.09%
80.00%	74.20%	74.80%	73.50%	182.4 W	79.19%	−0.81%
70.00%	64.30%	66.20%	62.30%	176.1 W	68.62%	−1.38%
60.00%	53.70%	53.10%	54.30%	168.9 W	57.31%	−2.69%
50.00%	44.80%	44.60%	45.10%	162.8 W	47.81%	−2.19%
40.00%	35.30%	36.60%	33.90%	155.8 W	37.67%	−2.33%
30.00%	28.00%	28.50%	27.50%	150.6 W	29.88%	−0.12%
20.00%	18.50%	12.60%	24.40%	143.3 W	19.74%	−0.26%
10.00%	10.00%	9.10%	10.90%	136.6 W	10.67%	0.67%
0.00%	0.10%	0.10%	0.10%	128.9 W	0.11%	0.11%

Tab. 6.1: *Mean values for CPU utilization and power consumption from a default run of the SPECpower_ssj2008 benchmark* The values are calculated by Sunshot from the trace shown in figure 6.6(b) on the preceding page.

In conclusion, it seems to be hard work to develop a method that can steadily stress a system at a defined load level. The SPEC people have already reached some achievement in this point so a detailed analysis of their methods could bring helpful insights. Nevertheless, at least in our test their methods did not work perfectly. Hence, it could be necessary to further enhance them.

6.2 HPCC Benchmark

6.2.1 Description of the Benchmark

The High Performance Computing Challenge Benchmark[37] is actually a benchmark suite combining seven different tests into one consistent environment. It is developed in cooperation of several partners from research and industry in the scope of the DARPA HPCS[3] project coordinated by Jack Dongarra at the Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville.

The motivation for the benchmark suite is that "modern standard [parallel computer] architecture produces a "steep" multi-layered memory hierarchy"[38, slide 11] reaching from registers, and cache over the local memory and remote memory to the disk. In the HPCC benchmark suite, "each benchmark focuses on a different part of the memory hierarchy"[38, slide 12]. Please see the memory hierarchy and benchmark focuses in figure 6.7.

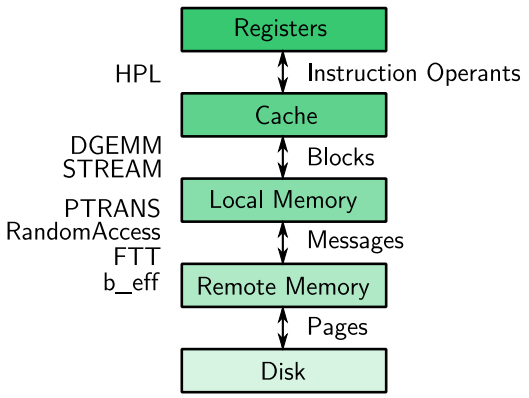


Fig. 6.7: *Memory hierarchy levels and test focuses* [38]

The single benchmarks are:

1. HPL - High Performance Linpack
"HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers"[39].

2. DGEMM - Double-Precision General Matrix Multiply "The DGEMM benchmark measures the floating point performance of a processor (or core). The code is cache friendly, thus memory bandwidth has little affect on results and the results obtained should be reasonably close to the theoretical peak performance of the processor."[40]

[3]DARPA High Productivity Computing Systems (`http://www.highproductivity.org/`)

3. STREAM - Sustainable Memory Bandwidth "The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels"[41].

4. PTRANS - Parallel Matrix Transpose "PTRANS [...] exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network."[37]

5. RandomAccess "RandomAccess measures the rate of integer random updates of memory"[37].

6. FFT - Fast Fourier Transform "FFT measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT)."[37]

7. b_{eff} - Effective Bandwidth "The effective bandwidth b_{eff} measures the accumulated bandwidth of the communication network of parallel and/or distributed computing systems. Several message sizes, communication patterns and methods are used. The algorithm uses an average to take into account that short and long messages are transferred with different bandwidth values in real applications."[42]

6.2.2 Test Environment

We use the HPCC benchmark version 1.3.1. This is the current version released in December, 2008. The test runs are performed on the PVS research cluster described in figure 5.4 on page 57. For running the HPCC benchmark suite an implementation of the Message Passing Interface (MPI) is needed. We use the free MPICH2, version 1.0.8p1 released in March, 2009. The test runs are made using four cluster nodes (`node06`–`node09`) and hpcc is started with eight ranks, thus we have one rank per CPU. The default configuration is slightly modified to achieve a longer runtime for a better ability to identify the different test phases and to minimize the portion of potential fade in and fade out effects at phase changes. We use the input file coming with the HPCC benchmark sources and increase the problem size `Ns` from the default 1000 to 10000.

For the runs with tracing, the Power Tracer is executed on `node06` tracing the power consumption of all four used nodes. For the utilization tracing, on each node one instance of the same RUT tool as used for the SPECpower analysis is started from `node06` with ssh. The setup is illustrated in figure 6.8 on the next page.

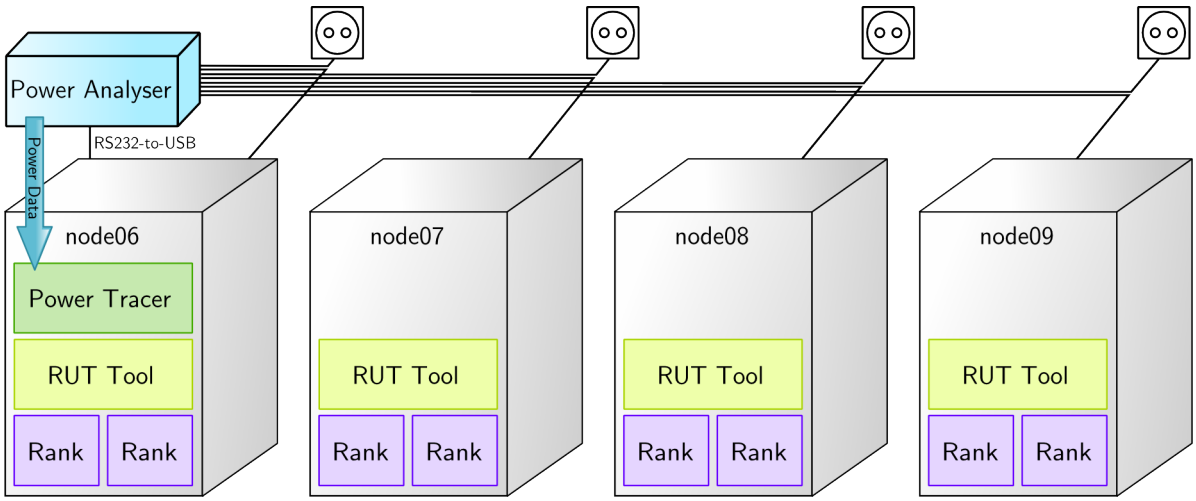


Fig. 6.8: *Schematic diagram of the HPCC test setup with our environment*

6.2.3 Test Results and Interpretation

The HPCC benchmark generates an output text file called hpccoutf.txt containing the results of all tests. Since we are not interested in performance analysis here, the only relevant data in this file are the border times between the single tests. The trace project of a complete HPCC run using the setup and configuration described in the last section is visualized in figure 6.9 on the following page. During the run, only 2 MB of data have been written to the local disk, probably not even by the benchmark but by a system process. Therefore we will completely disregard the hard disk traces. The border times from the results file are drawn in the figure and the phases are labeled. As you can see, some of the tests have multiple stages so there are actually 13 phases instead of the seven expected.

In the bare trace visualization, the phases were not as simple to identify as for the SPECpower_ssj2008 benchmark, but with the border lines drawn in we can discover different resources utilization profiles in the different phases. Especially, it emerges that not all phases are using all available CPUs. Let us take a look at the single phases:

In **phase 1** PTRANS is running. The resources utilization is discontinuous. Even if some memory is used and the network shows some activity, the power consumptions seems to follow only the CPU utilization. Due to the discontinuity this test is of low interest for our purposes.

Phase 2 is the Linpack benchmark. Surprisingly, only one CPU is used per node. In addition, the utilization of the one CPU is obviously not constant at its maximum. Hence, it seems that HPL is not suitable for stressing the CPU to its maximum, at least not with the parameters used here.

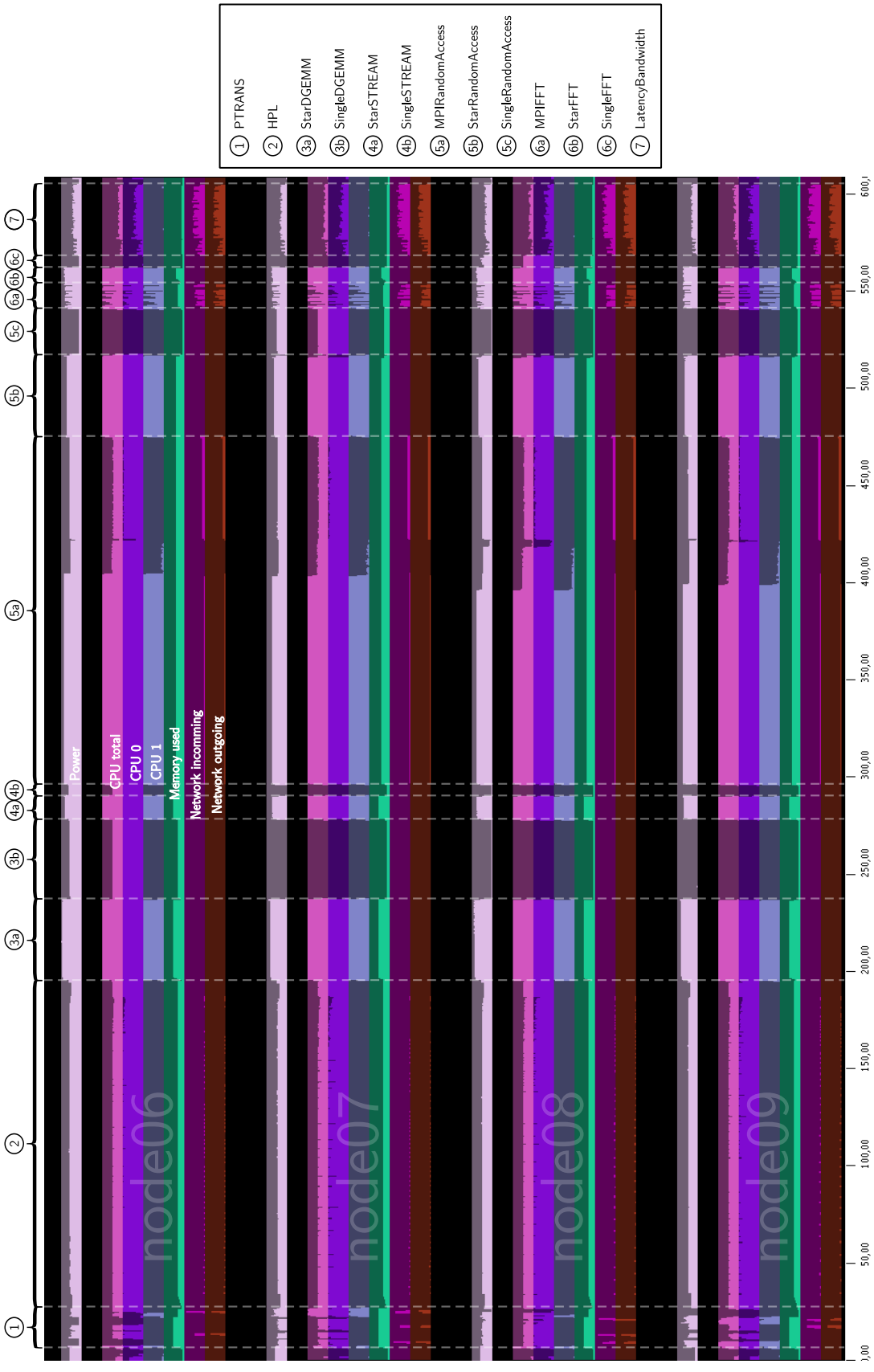


Fig. 6.9: *Power and utilization trace from a run of the HPCC benchmark on the PVS Cluster* The run uses four nodes and eight processes. All the power graphs are scaled to the range [122 W, 220 W].

On the contrary, **Phase 3a** and **3b** could be of use for that purpose. These are the phases running DGAMM with multiple and single processes per node. They seem to produce a constant maximal utilization for two or one CPU on each system, respectively. The only problem could be the simultaneous utilization of some memory that would better be avoided for measuring only the CPU's influence to the power consumption.

Phase 4a and **4b** running the STREAM benchmark shows exactly the same resources utilization as the phases 3a+b, but with 10% (220 W to 200 W) lower power consumption. Further investigation would be interesting to identify the cause for this difference. This is not only an interpolation issue. A deeper analysis using the capabilities of Sunshot as zooming, histograms and mean value calculation engaged no significant difference in the traced resources utilization between phases 3a and 4a or 3b and 4b, respectively. Perhaps there are other relevant utilization values not yet traced in our environment.

Phase 5a running the MPIRandomAccess test shows two subphases. The first one has nearly the same utilization profile as phase 4a with lower memory usage and an additional slight network activity. With the network usage eliminated it could be a better alternative for stressing the CPU than 4a. The second subphase is uninteresting since no resource is stressed to its maximum. With **phase 5b** we have exactly the described case of the first subphase of 5a without the network utilization. Until now from all HPCC tests this is the best CPU-only stress test. **Phase 5c** is just the same as phase 5b with only one process running.

The first FFT phase labeled **6a** again is very discontinuous in resources utilization thus not of interest for us. **Phase 6b** has nearly the same profile as phase 5b, but showing discontinuities in power consumption not explainable with only the information within the trace. If we later think about using this test for our purposes, it has to be investigated whether this is a persistent effect and if so where it is coming from. Again, **phase 6c** is just like phase 6b with only one process running and of no further interest.

The last **phase 7** is running the LatencyBandwidth test. It shows moderate CPU utilization and discontinuous network utilization. Configured in the way used here, this is not what we are looking for. But the discontinuity is not very distinct, so perhaps this test could be more useful for our purpose running it with other parameters.

We summarize, that barely one of the tests in the HPCC benchmark suite can directly be used for our purpose of investigating the impact of a single component's utilization to the system's power consumption. But some of the tests could become suitable with slight modifications. Candidates for this are especially the RandomAccess test used in phase 5b and maybe the LatencyBandwidth test b_{eff} in phase 7. Furthermore we have found some hints, that the currently traces utilization values could be insufficient. To clarify this, further comparisons between the DGRAMM and STREAM phases should be done. Both of these topics could be subject for a future work.

7 Summary and Conclusion

The work for this thesis was entirely driven by the idea of saving power in high performance clusters by knowing about future program behavior. The first conclusion made was the need of a tracing environment for analyzing the potential of the approach. This tracing environment should be able to profile the power consumption of a system together with the utilization of several system resources. It is the precondition for the development of a benchmark that stresses different system components at different load levels and measures the power consumed in the meanwhile. Purpose of such a benchmark would be to analyze correlations between using different components and the system's consumed power for making conclusions at the power saving capability of the system. Goal of the thesis was to create the tracing environment and make some first analysis of existing benchmarks.

At the beginning we had only these ideas and the very new HDTrace tracing format not yet suitable for implementing them. We enhanced the format by the topology concept and HDStats, an extension for efficient tracing of statistical values. We designed two entirely new libraries, one for power tracing and one for resources utilization tracing.

The three new libraries are implemented using reasonable techniques as multi-threading, automatic build system, integrated tests and good documentation. They have been evaluated for correctness and efficiency and could be marked as well usable.

Afterwards the two benchmarks SPECpower_ssj2008 and HPCC have been analyzed with the help of our new tracing environment. It emerged that SPECpower_ssj2008 has a reasonably working concepts for CPU load control but further testing would be necessary to decide whether it is suitable for our needs. The tests used by HPCC in the default configuration are mostly unusable for our purposes. Anyhow, some of them have shown desired characteristics so further testing with different parameters could be promising.

In conclusion, we have developed a useful profiling environment which is a first achievement. We could demonstrate their help with analyzing benchmark concepts concerning power consumption in computer clusters. Even if they could still get improved, they build a good foundation to proceed with our approach. However, there is still a long way to go until the High Performance Computing caravan can finally enter the Green IT valley.

8 Future Work

8.1 Work in Progress

Ongoing Development of the HDTrace format

The HDTrace format under heavy development. New concepts are integrated and existing concepts become successively enhanced. For example, in the meantime a relations concept has been integrated for tracking connections between different events.

MPI Wrapper for Seamless Statistics Tracing

The existing MPI wrapper mentioned in section 2.1 has already been adapted to the format modifications and extended by the functionality provided by our two new libraries. We are now able to trace power and utilization statistics together with the MPI function calls for an arbitrary MPI program. We just have to compile it with a modified mpicc linking the wrapper and necessary libraries to the program and then start it as usual.

Simulation of Power Consumption in PIOsim

In parallel to this thesis, Timo Minartz already has used the new environment for his own thesis about *Model and simulation of power consumption and power saving potential of energy efficient cluster hardware*. He used the Power Tracer Library and the Resources Utilization Library to analyze the power consumption of different system components to integrate power simulation functionality into the PIOsim cluster simulator.

8.2 Ideas for Enhancing the Libraries

Buffering in HDStats

In the current implementation of HDStats, each entry is written directly into the statistics trace file once is is complete. Depending on the entry size this leads to many short writes. Using a buffer to collect a number of entries and write them all at once could decrease the CPU usage of the library.

Split Configuration for the Power Tracer

It would be great to provide the power tracing capability also to the regular users of a cluster. In the current configuration concept for the Power Tracer, each user would have to know the hardware configuration, that is, which node is measured by which channel of which power analyzer connected to which node. For this purpose it would be nice to have an abstraction layer for the power tracing infrastructure. The idea is, that the administrator configures the power analyzers setup somewhere and a user simply has to specify which values of which nodes he wants to trace for which time interval. After the interval has passed, he will just get the traces.

8.3 Interesting Further Investigations

Technical Power Analyzing Details

In section 5.1.1 on page 54 we have seen, that the power trace has always a delay of about $50ms$ to the utilization trace. We assumed, that this is due to the time needed to collect and transfer the measurement values to the Power Tracer but there could also be other reasons. In addition such effect as the attenuation of the power supply unit could influence the accuracy of the data. To be able to interpret the traces correctly, a deeper insight to this area is necessary. Therefore it would be expedient to make further investigations, ideally in cooperation with electrical engineers.

Verification of the SPECpower_ssj2008 Load Levels

The tests section 6.1.3 on page 66 revealed a discrepancy between the actual load levels as measured by the benchmark itself and those extracted from our traces. This should be investigated by further testing to discover in which concept there is an error, or whether

there are just different measuring methods used. In the second case a comparison of the methods would be interesting.

Trying Different Parameters for Interesting HPCC Tests

We found two of the tests in the HPCC benchmark suite interesting for our purposes in section 6.2. That is the RandomAccess test for CPU stressing and the Latency-Bandwidth test for stressing the network. Those should be analyzed with other input parameters. We would like to so see whether it is possible to make them fit better for our needs of producing constant stress levels for a defined period of time.

8.4 An Idea for the Farther Future

Automatic Generation of Resources Utilization Profiles

The introduction describes the far goal of giving the developer of a program the possibility to pass information about the program's future behavior to the power management system. Another or supplementary approach could be to automatically create a resources utilization profile for a program during one run, and use the gained information in later runs to make power saving decisions.

A API Documentation of the HDTrace Writing C Library

A.1 Module Documentation

A.1.1 HDTrace Topology

Detailed Description

The topology of a complete trace project describes a tree structure where each single trace can be assigned to a tree node.

As we have several singly assigned trace files that build one whole trace project, this structuring method allows us to describe the associations between the single traces.

The root of the topology is defined to be the project name. The types of the deeper levels can be freely specified. A topology's structure is fully specified by the topology level types. The level type should describe the semantics of the nodes on the level.

An example of a topology structure is (Project, Host, Process, Thread).

Each trace is assigned to exactly one node of the topology called the trace's topology node. Each such topology node has a label that must be unique at the node's topology level. A node is well-defined by the unique path that leads from the root of the topology to this node. The path is the list of labels of each node in the order they are passed when walking on the edges of the tree to the target node starting from the root node.

An example for a leaf node in the example topology above could be (myProject, node01, rank3, thread0). The path string and unique id of that node would be `myProject_node01_rank3_thread0`. The node in this path on the first level were (myProject, node01) with the path string `myProject_node01`.

Note:

We count the topology levels starting with the root level as number zero.

The topology structure represented by the topology level types is written into the project file in order to be used as labels in trace visualization.

Note:

The node labels as well as the topology types are restricted to strings with alphanumerical ASCII characters only.

Typedefs

- typedef struct _hdTopology hdTopology
- typedef struct _hdTopoNode hdTopoNode

Functions

- hdTopology * hdT_createTopology (const char *project, const char **levels, int nlevels)
- int hdT_getTopoDepth (const hdTopology *topology)
- int hdT_destroyTopology (hdTopology *topology)
- hdTopoNode * hdT_createTopoNode (hdTopology *topology, const char **path, int length)
- int hdT_getTopoNodeLevel (const hdTopoNode *node)
- const char * hdT_getTopoPathString (const hdTopoNode *node)
- const char * hdT_getTopoPathLabel (const hdTopoNode *node, int level)
- int hdT_destroyTopoNode (hdTopoNode *node)

Typedef Documentation

hdTopology

Type for using topology objects.

Use hdT_createTopology to get one of this objects.

hdTopoNode

Type for using topology node objects.

Use hdT_createTopoNode to get one of this objects.

Function Documentation

hdTopology* hdT_createTopology (const char * *project*, const char ** *levels*, int *nlevels*)

Create new topology.

The topology is simply specified by the names of the tree levels.

The first one (level 0) is the name of the root node and also the name of the HDTrace project. The second one is the name of level 1, the third name belongs to level 2 and so on.

In HDTrace these names do have only descriptive character and are not used by the library beside writing them to the Trace Info. Other tools like the Project Description Merger could use this information for consistency checks.

This function generates and returns a hdTopology object that you have to destroy when no longer needed by passing it to hdT_destroyTopology.

Example usage:

```
const char *levels[] = {"Host", "Process", "Thread"};
hdTopology *myTopology = hdT_createTopology("myProject", levels, 3);
```

Parameters:

project Name of the HDTrace project

levels Array of names

nlevels Number of levels the topology has beside root level

Returns:

HDTrace topology object

Return values:

topology on success

NULL on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_MALLOC

See also:

hdT_destroyTopology

hdTopoNode* hdT_createTopoNode (hdTopology * *topology*, const char ** *path*, int *length*)

Create new topology node.

The topology is the semantical structure of HDTrace files. It is meant to map a hierarchical structure to the trace files.

Actually the topology is a tree structure, so the place of each node in the topology is exactly specified by the path to reach this node from the root node. The root node is always the starting point, so it is omitted here when passing the path to this function.

This function takes such a path and gives you a hdTopoNode object representing the node you reach walking the given path. This could be an inner as well as a leaf node of the topology tree.

With the path for creating nodes, you implicitly label the path's nodes on each involved level of the topology tree. And so the tree is automatically extended with each hdTopoNode object created.

A good example is the typical structure of a parallel program: Hosts, Processes, Threads.

Each thread has its own control flow an so it might be a good idea to create one trace for each thread. So you will choose to have the threads as leaf nodes of your tracing topology tree. Of course you are free to use only two levels and so produce traces per process.

The node labels on the path are used to generate the canonical filenames for the traces.

Example usages:

```
const char *path[] = {hostname, pid, thread_id};
hdTopoNode *myTopoNode = hdT_createTopoNode(myTopology, path, 3);


const char *path[] = {hostname, pid};
hdTopoNode *myTopoNode = hdT_createTopoNode(myTopology, path, 2);
```

Parameters:

topology Topology this node should belong to

path Array of pointers to the node labels on the path.

length Length of the path.

Returns:

HDTrace topology node object

Return values:

node on success

NULL on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_MALLOC

See also:

hdT_destroyTopoNode

int hdT_destroyTopology (hdTopology * *topology*)

Destroy topology object.

Parameters:

topology Topology object to destroy

Return values:

0 on success

-1 on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

See also:

hdT_createTopology

int hdT_destroyTopoNode (hdTopoNode * *node*)

Destroy topology node.

Parameters:

node Topology node to destroy

Return values:

0 on success

-1 on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

See also:

hdT_createTopoNode

int hdT_getTopoDepth (const hdTopology * *topology*)

Get the depth of a topology.

Returns the number of levels in the topology including root level (level 0).

Parameters:

topology Topology to use

Returns:

The number of topology levels.

Return values:

>0 on success

-1 on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

int hdT_getTopoNodeLevel (const hdTopoNode * *node*)

Get the topology tree level of a node.

Returns the level where the *node* take place in its topology.

Parameters:

node Topology node to use

Returns:

The number of the topology level *node* lives in.

Return values:

>0 on success

-1 on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

const char* hdT_getTopoPathLabel (const hdTopoNode * *node*, int *level*)

Get the label of one node at the passed *node's* path.

Searches the path of the given *node* for the topology node at the given *level* and return its label.

For example, create a node like this:

```
const char *path[] = {host0, process0, thread0};
hdTopoNode *myTopoNode = hdT_createTopoNode(myTopology, path, 3);
```

then the following call will return `"process0"`:

```
hdT_getTopoPathLabel(myTopoNode, 2);
```

Parameters:

node Topology node to use

level Level to get node label for (>0)

Returns:

Label of the node sitting at *level* on *node's* path

Return values:

String on success

NULL on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

const char* hdT_getTopoPathString (const hdTopoNode * *node*)

Get the path string of a topology node.

Returns a string representation of the node's path.

For example: `host1_process1_thread1`

Parameters:

node Topology node to use

Returns:

String representing the node's path

Return values:

String on success

NULL on error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT

A.1.2 HDTrace Statistics Writing Library

Detailed Description

In HDTrace format statistics are a special kind of trace for periodically occurring data. One canonical usage is to trace utilization data like CPU load, memory usage, used amounts of network or I/O bandwidth.

The statistics are categorized in so called groups. Each group is associated with the tracing topology at any level and each group's data is written binary to its own data file.

A statistics group consists of entries, each marked with a timestamp. One entry consists of a tuple of values, each with one of several well defined types. In one group, all entries have exactly the same number of values with exactly the same types and so exactly the same length, except when using values of string type (not yet implemented).

The HDTrace statistics group data file is self-descriptive. At the very beginning there is a header in XML describing the values and types each entry in the following binary part of the file contains.

Library Usage

Outline of creating HDTrace Statistics Groups using this library:

1. Create the group first (hdS_createGroup, called once)
2. Add all value info (hdS_addValue, called several times)
3. Commit the group (hdS_commitGroup, called once)
4. Use the group to trace data (**hdS_write**..., called many times)
5. Finalize the group (hdS_finalize, called once)

Writing Values Write order of the values must be exactly the same as they were registered!

At the beginning of each entry, the current time is taken and a timestamp is written for the new entry. This is always done, when the first **hdS_write**∗ function for an entry is called. A call to hdS_writeEntry is always the first such call for an entry.

The timestamp is 4 byte integer seconds and 4 byte integer nanoseconds since epoch (Jan 01 1970). Function **gettimeofday** is used to get the time and the returned microseconds are transferred to nanoseconds.

File Content The file written for each statistics group is constructed in three parts.

1. The first part is only 6 bytes an contains the length of the header, the second part, as string in decimal notation with 5 numbers and leading zeros followed by a newline character '\n'.
2. The second part is the header. It describes the binary data, the third part, and is written in correct XML notation.
3. The third part is where the actual trace data are written to in entries like described in "Writing Values".

Example:

```
00362
<Group name="Utilization" timestampDatatype="EPOCH"
    timeAdjustment="-0000000000.000000000">
  <Value name="CPU_TOTAL" type="FLOAT" unit="%" grouping="CPU" />
  <Value name="MEM_USED" type="INT64" unit="B" grouping="MEM" />
  <Value name="NET_IN" type="INT64" unit="B" grouping="NET" />
  <Value name="NET_OUT" type="INT64" unit="B" grouping="NET" />
</Group>
BINARYBINARYBINARYBINARYBINAY.......
```

362 since '\n' and spaces also count of cause.

Defines

- #define order_bytes32ip(x)
- #define order_bytes64ip(x)
- #define order_bytes32fp(x)
- #define order_bytes64fp(x)

Typedefs

- typedef enum _hdStatsValueType hdStatsValueType
- typedef struct _hdStatsGroup hdStatsGroup

Enumerations

- enum _hdStatsValueType {
  INT32, INT64, FLOAT, DOUBLE,
  STRING }

Functions

- hdStatsGroup * hdS_createGroup (const char *groupName, hdTopoNode *topoNode, int topoLevel)
- int hdS_addValue (hdStatsGroup *group, const char *name, hdStatsValueType type, const char *unit, const char *grouping)
- int hdS_commitGroup (hdStatsGroup *group)
- int hdS_enableGroup (hdStatsGroup *group)
- int hdS_disableGroup (hdStatsGroup *group)
- int hdS_writeEntry (hdStatsGroup *group, void *entry, size_t entryLength)
- int hdS_writeInt32Value (hdStatsGroup *group, int32_t value)
- int hdS_writeInt64Value (hdStatsGroup *group, int64_t value)
- int hdS_writeFloatValue (hdStatsGroup *group, float value)
- int hdS_writeDoubleValue (hdStatsGroup *group, double value)
- int hdS_writeString (hdStatsGroup *group, const char *str)
- int hdS_finalize (hdStatsGroup *group)

Define Documentation

#define order_bytes32fp(x)

Conversation of a 32 bit floating point number (float) to network byte order.

The argument has to be a pointer to the float and becomes converted in place.

#define order_bytes32ip(x)

Conversation of a 32 bit integer to network byte order.

The argument has to be a pointer to the integer and becomes converted in place.

#define order_bytes64fp(x)

Conversation of a 64 bit floating point number (double) to network byte order.

The argument has to be a pointer to the double and becomes converted in place.

#define order_bytes64ip(x)

Conversation of a 64 bit integer to network byte order.

The argument has to be a pointer to the integer and becomes converted in place.

Typedef Documentation

hdStatsGroup

Type to use for statistics groups.

hdStatsValueType

Type to use for value types for statistics groups.

Enumeration Type Documentation

enum _hdStatsValueType

Enumeration of value types for statistics groups.

Enumerator:

INT32 32 bit integer

INT64 64 bit integer

FLOAT single precision floating point

DOUBLE double precision floating point

STRING String.

Function Documentation

int hdS_addValue (hdStatsGroup * *group*, const char * *name*, hdStatsValueType *type*, const char * *unit*, const char * *grouping*)

Add a new value to the entry structure of a statistics group.

By multiple calls of this function you can specify the structure of an entry to the group. This is only possible as long as the group is not committed. After committing the group by calling hdS_commitGroup any modification to the entry structure of a group is impossible.

Each call of this function adds a new value to the end of the entry structure of the group.

Parameters:

group Statistics group to modify

name Name of the new value (Not more than HDS_MAX_VALUE_NAME_LENGTH characters including XML escapes done automatically)

type Type of the new value

unit Unit string of the new value. (NULL or not more than HDS_MAX_UNIT_NAME_LENGTH characters including XML escapes done automatically)

grouping Grouping string of the new value (NULL or not more than HDS_MAX_GROUPING_NAME_LENGTH characters including XML escapes done automatically)

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_BUFFER_OVERFLOW
- HDS_ERR_GROUP_COMMIT_STATE

int hdS_commitGroup (hdStatsGroup * *group*)

Commit statistics group, closes initialization step.

Calling this function for a statistics group closes its initialization and writes the descriptive header to the group's trace file.

The group is not enabled automatically, so before any passed values or entries are recorded by the group, you have to call hdS_enableGroup.

Parameters:

group Statistics group to commit

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HDS_ERR_GROUP_COMMIT_STATE

hdStatsGroup* hdS_createGroup (const char * *groupName*, hdTopoNode * *topoNode*, int *topoLevel*)

Create a new statistics group.

Creates and opens the file for a new statistics group. The filename is built using the rules for HDTrace statistics files an the given topology and level.

For example:

```
const char *levels[] = {"Host","Process"};
hdTopology *myTopology = hdT_createTopology("MyProject", levels, 2);
const char *path[] = {"host0","process0"};
hdTopoNode *myTopoNode = hdT_createTopoNode(myTopology, path, 2);
hdStatsGroup *myGroup = hdS_createGroup("MyGroup", myTopoNode, 1);
```

creates a file named `MyProject_host0_MyGroup.stat`

Parameters:

groupName Name of the new statistics group (Not more than HDS_MAX_GROUP_-NAME_LENGTH characters including XML escapes done automatically)

topoNode Topology node to use

topoLevel Topology level the group shell belong to

Return values:

Statistics group on success

NULL error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_MALLOC
- HD_ERR_BUFFER_OVERFLOW
- HD_ERR_CREATE_FILE

See also:

hdT_createTopoNode, hdT_createTopology

int hdS_disableGroup (hdStatsGroup * *group*)

Disable statistics group.

This function does not set errno! So it can easier be used as reaction of errors.

Parameters:

group Statistics group to disable

Returns:

Error state and type

Return values:

1 Success, was already disabled

0 Success, is now disabled

-1 Error: *group* is not committed

-2 Error: *group* is NULL

See also:

hdS_enableGroup

int hdS_enableGroup (hdStatsGroup * *group*)

Enable statistics group.

This function does not set errno!

Parameters:

group Statistics group to enable

Returns:

Error state and type

Return values:

1 Success, was already enabled

0 Success, is now enabled

-1 Error: *group* is not committed

-2 Error: *group* is NULL

See also:

hdS_disableGroup

int hdS_finalize (hdStatsGroup * *group*)

Finalizes a statistics group.

This must be the last hdS_* function called in a program for each statistics group.

Parameters:

group Statistics group to use.

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HDS_ERR_GROUP_COMMIT_STATE

int hdS_writeDoubleValue (hdStatsGroup * *group*, double *value*)

Writes 8 byte double as next value to a statistics group.

Checks if the next value in current entry is of type DOUBLE and append it to the group buffer if so.

Parameters:

group Statistics Group

value DOUBLE value to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_TRACE_DISABLED
- HDS_ERR_GROUP_COMMIT_STATE
- HDS_ERR_ENTRY_STATE

int hdS_writeEntry (hdStatsGroup * *group*, void * *entry*, size_t *entryLength*)

Writes a complete entry to a statistics group.

Attention:

Do only use this function if you really know what you are doing.
No byte order conversation is done. No check for consistency with the specified entry structure beside length check is done. In groups containing string values even the length check is omitted since it is not possible.

Parameters:

group Statistics group to use

entry Pointer to the entry to write

entryLength Length of the entry to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_TRACE_DISABLED
- HDS_ERR_GROUP_COMMIT_STATE
- HDS_ERR_UNEXPECTED_ARGVALUE
- HDS_ERR_ENTRY_STATE

int hdS_writeFloatValue (hdStatsGroup * *group*, float *value*)

Writes 4 byte float as next value to a statistics group.

Checks if the next value in current entry is of type FLOAT and append it to the group buffer if so.

Parameters:

group Statistics Group

value FLOAT value to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_TRACE_DISABLED
- HDS_ERR_GROUP_COMMIT_STATE
- HDS_ERR_ENTRY_STATE

int hdS_writeInt32Value (hdStatsGroup * *group*, int32_t *value*)

Writes 4 byte integer as next value to a statistics group.

Checks if the next value in current entry is of type INT32 and append it to the group buffer if so.

Parameters:

group Statistics Group

value INT32 value to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_TRACE_DISABLED
- HDS_ERR_GROUP_COMMIT_STATE
- HDS_ERR_ENTRY_STATE

int hdS_writeInt64Value (hdStatsGroup * *group*, int64_t *value*)

Writes 8 byte integer as next value to a statistics group.

Checks if the next value in current entry is of type INT64 and append it to the group buffer if so.

Parameters:

group Statistics Group

value INT64 value to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HD_ERR_TRACE_DISABLED
- HDS_ERR_GROUP_COMMIT_STATE
- HDS_ERR_ENTRY_STATE

int hdS_writeString (hdStatsGroup * *group*, const char * *str*)

Writes string as the next value to a statistics group.

This function is not yet implemented.

Parameters:

group Statistics group to use

str STRING value to write

Returns:

Error state

Return values:

0 Success

-1 Error, setting errno

Values of errno:

- HD_ERR_INVALID_ARGUMENT
- HDS_ERR_GROUP_COMMIT_STATE

A.1.3 HDTrace Errors

Enumerations

- enum hdCommonError {

  HD_ERR_INVALID_ARGUMENT, HD_ERR_MALLOC, HD_ERR_BUFFER_OVERFLOW, HD_ERR_GET_TIME,

  HD_ERR_CREATE_FILE, HD_ERR_WRITE_FILE, HD_ERR_CLOSE_FILE, HD_ERR_TIMEOUT,

  HD_ERR_TRACE_DISABLED, HD_ERR_INVALID_CONTEXT, HD_ERR_UNKNOWN }
- enum hdStatsError { HDS_ERR_GROUP_COMMIT_STATE, HDS_ERR_UNEXPECTED_ARGVALUE, HDS_ERR_ENTRY_STATE }

Functions

- char * hdT_strerror (int errno)

Enumeration Type Documentation

enum hdCommonError

Enumeration for common errors in function.

hdT_* as well as hdS_* functions can set errno to one of these values in case of an error

Enumerator:

HD_ERR_INVALID_ARGUMENT Invalid argument.

HD_ERR_MALLOC Error while memory allocation.

HD_ERR_BUFFER_OVERFLOW Error due to buffer overflow.

HD_ERR_GET_TIME Error while getting system time.

HD_ERR_CREATE_FILE Error while creating a file.

HD_ERR_WRITE_FILE Error while writing a file.

HD_ERR_CLOSE_FILE Error while closing a file.

HD_ERR_TIMEOUT Timeout occurred.

HD_ERR_TRACE_DISABLED Trace is disabled.

HD_ERR_UNKNOWN Error with unknown cause.

enum hdStatsError

Enumeration for errors in statistics functions.

hdS_* functions can set errno to one of these values in case of an error

Enumerator:

HDS_ERR_GROUP_COMMIT_STATE Statistics group's commit state is not the needed.

HDS_ERR_UNEXPECTED_ARGVALUE One of the arguments has an unexpected value.

HDS_ERR_ENTRY_STATE State of the current entry is wrong for requested action.

Function Documentation

char* hdT_strerror (int *errno*)

Return a string describing the error represented by errno.

Parameters:

errno errno value to get the string for

Returns:

Error describing string or null for unknown errno value

B API Documentation of the Power Tracer and Library

B.1 Module Documentation

B.1.1 Power Tracer Library

Detailed Description

The Power Tracer Library (libPT) is the part of the Power Tracer providing the power tracing functionality directly to other programs. It is able to trace the power measured by a power analyzer connected.

Currently only the ZES ZIMMER LMG450 power analyzer connected via RS-232 serial port is supported.

The library uses the HDTraceWritingCLibrary to produce statistics traces as specified by the HDTrace format. The traces can be embedded in an HDTrace project and visualized in Sunshot, the native viewer for HDTrace format.

Library Usage

Outline of creating Power Traces using this library:

1. Create a configuration file matching the setup of the power analyser and containing the traces you want to create.
2. Create a PowerTrace object (pt_createTrace)
3. Start tracing (pt_startTracing)
4. Stop tracing (pt_stopTracing)
5. Finalize the trace and destroy the PowerTrace object (pt_finalizeTrace)

Creating the configuration file The configuration file has to contain the general hardware setup of the power analyzer and all traces to create with all relevant information. Therefore the file is divided into sections, one for the general setup and one for each trace to create.

In the general section we need the following values:

- `device` specifying the type of the power analyzer in use.
- `port` specifying where the power analyzer is connected.
- `cycle` specifying the period time for the tracing.
- `project` specifying the project name.
- `topology` specifying the topology for the project.

In each trace section we need the following values:

- `type` specifying the type of the trace.

- `node` specifying the node to associate with the trace.
- `channel` specifying the channel of the power analyzer to use for the trace.
- `values` specifying which values to trace

Example:

```
[General]
device=LMG450
port=node06:/dev/ttyUSB0
cycle=100
project=MeinProjekt
topology=Cluster_Host_Process_Thread

[Trace]
type=HDSTATS
node=pvs_node06
channel=1
values=Utrms,Itrms,P

[Trace]
type=HDSTATS
node=pvs_node07
channel=2
values=Utrms,Itrms,P
```

- `port` has the format NODE:DEVICE. NODE is the node which has the power analyzer connected via the RS-232 port. DEVICE is the name of the serial device file to use.
- `topology` is built using the topology level types concatenated with underscores in the same way as the topology node path string is created.
- `node` is given as the topology node path string and has to match the topology defined. Of course, as in the example, the topology can have more levels than the node uses.
- `values` is a comma separated list of strings, defining the values to trace. The strings can be specific to the used power analyzer. Currently the three shown here are the only supported values.

Creating a PowerTrace object With the configuration file created you are ready to create a PowerTrace object:

```
PowerTrace *myPowerTrace;
pt_createTrace("pt.cfg", NULL, &myPowerTrace);
```

Note:

Passing NULL to the second argument will use the topology defined in the configuration file.

Start and stop tracing The tracing does not start before you tell it to by calling pt_-startTracing.

You can start and stop tracing multiple times by calling pt_startTracing and pt_stopTracing. When calling pt_stopTracing the current tracing period is finished and no new period is started. Calling pt_startTracing immediately instructs the power analyzer to send data. The exact start time depend on the arrival time of the new data.

Environment Variables

There in an environment variables used by libPT: `PT_VERBOSITY`

`PT_VERBOSITY` can be set to a number in the range -1 to 3. The default is 0 only showing error messaged. 1 enables warnings, 2 enables info messages and 3 enables all debugging output. -1 makes the library absolutely silent, even in case of a fatal error. This value affects only the messages printed to `stderr` not the behavior of the functions.

Defines

- #define PT_SUCCESS
- #define PT_ECONFNOTFOUND
- #define PT_ECONFINVALID
- #define PT_ENOTRACES
- #define PT_EMEMORY
- #define PT_EHDLIB
- #define PT_EDEVICE
- #define PT_ETHREAD

Typedefs

- typedef struct powertrace_s PowerTrace

Functions

- int pt_createTrace (const char *configfile, hdTopology *topology, PowerTrace **trace)
- char * pt_getHostname (PowerTrace *trace)
- int pt_startTracing (PowerTrace *trace)
- int pt_stopTracing (PowerTrace *trace)
- int pt_finalizeTrace (PowerTrace *trace)

Define Documentation

#define PT_ECONFINVALID

Configuration read from file is invalid.

#define PT_ECONFNOTFOUND

Could not find configuration file.

#define PT_EDEVICE

Problem during communication with measurement device.

#define PT_EHDLIB

Error in HDTrace library.

#define PT_EMEMORY

Out of memory.

#define PT_ENOTRACES

No traces found in configuration.

#define PT_ETHREAD

Cannot create tracing thread.

#define PT_SUCCESS

Success.

Typedef Documentation

PowerTrace

Type definition of power trace object.

Function Documentation

int pt_createTrace (const char * *configfile*, hdTopology * *topology*, PowerTrace ** *trace*)

Create a power trace using the passed configuration file.

Use this function to create a new power trace. It will setup a new statistics trace with the configuration read from the file.

For the format of the configuration file, please take a look at the example in Library Usage

The tracing will not start until pt_startTracing is called for the PowerTrace object returned by this function.

Parameters:

configfile Name of the configuration file

topology Topology to override default or config file choice (NULL not to override)

trace Location to store the PowerTrace pointer (OUTPUT)

Returns:

Error state

Return values:

PT_SUCCESS Success

PT_ECONFNOTFOUND Could not find configuration file

PT_EMEMORY Out of memory

PT_ECONFINVALID Configuration read from file is invalid

PT_ENOTRACES No traces found in configuration

PT_EHDLIB Error in HDTrace library

PT_ETHREAD Cannot create tracing thread

int pt_finalizeTrace (PowerTrace * *trace*)

Finalize and free a power trace.

Parameters:

trace Power trace object

Returns:

Error state

Return values:

PT_SUCCESS Success
PT_EDEVICE Problem during communication with measurement device
PT_EMEMORY Out of memory

char* pt_getHostname (PowerTrace * *trace*)

Return the hostname with the measuring device connected if specified in config file.

Parameters:

trace Power trace object

Returns:

Hostname of NULL if none specified in config file

int pt_startTracing (PowerTrace * *trace*)

Start the power tracing.

Parameters:

trace Power trace object

Returns:

Indicate if tracing state changed

Return values:

0 Tracing is now started (was stopped before)
1 Tracing is started (as it was already before)
2 Parameter is NULL

int pt_stopTracing (PowerTrace * *trace*)

Stop the power tracing.

Parameters:

trace Power trace object

Returns:

Indicate if tracing state changed

Return values:

0 Tracing is now stopped (was started before)
1 Tracing is stopped (as it was already before)
2 Parameter is NULL

C API Documentation of the Resources Utilization Tracing Library

C.1 Module Documentation

C.1.1 Resources Utilization Tracing Library

Detailed Description

The Resources Utilization Tracing Library (libRUT) provides an easy way to produce traces of several utilization statistics available in a common UNIX system. It utilizes the Gtop library to get the statistics and the HDTraceWritingCLibrary to produce statistics traces as specified by the HDTrace format. The traces can be embedded in an HDTrace project and visualized in Sunshot, the native viewer for HDTrace format.

Library Usage

Outline of creating Resources Utilization Traces using this library:

1. Define sources to trace (rutSources, **RUTSRC_SET_**$*$ macros)
2. Create a UtilTrace object (rut_createTrace)
3. Start tracing (rut_startTrace)
4. Stop tracing (rut_stopTrace)
5. Finalize the trace and destroy the UtilTrace object (rut_finalizeTrace)

Defining sources After creating a rutSources object

```
rutSources sources;
```

you can either set every source you want to trace by hand

```
sources.CPU_UTIL = 1;
```

or you can use the provided macros to set a whole source group at once

```
RUTSRC_SET_CPU(sources)
```

or to simply set all sources available

```
RUTSRC_SET_ALL(sources)
```

Creating a UtilTrace object To create a UtilTrace object, you first need to create an *hdTopology* and an *hdTopoNode* object to define the place of your trace in a HDTrace project.

Refer to *hdTopology* section in *HDTraceWritingCLibrary* documentation for further information about this topic.

For the simplest case, you can use the following code fragment:

```
const char *levels[] = {"Host"};
hdTopology *myTopology = hdT_createTopology("MyProject", levels, 1);
const char *path[] = {"host0"};
hdTopoNode *myTopoNode = hdT_createTopoNode(myTopology, path, 1);
```

Note:

rut_finalizeTrace will not destroy the *hdTopology* and *hdTopoNode* objects for you, so if you don't need it for other purposes, you have to do this by calling *hdT_destroyTopoNode* and *hdT_destroyTopology*.

Now that you have an *hdTopology* object, an *hdTopoNode* object and a rutSources object with all sources set that you want to be included in the trace, you can create the UtilTrace object:

```
UtilTrace *myUtilTrace;
rut_createTrace(myTopology, myTopoNode, 1, mySources, 500, &myUtilTrace);
```

Note:

You cannot change the sources of an already created UtilTrace object

Start and stop tracing The tracing does not start before you tell it to by calling rut_startTrace.

You can start and stop tracing multiple times by calling rut_startTrace and rut_stopTrace. When calling rut_stopTrace the current tracing period is finished and no new period is started but the period timer is not reset. So after a subsequent rut_startTrace, the tracing will first wait until the current (virtual) period would end and take the next values at the beginning of the next period.

Environment Variables

There are two environment variables used by libRUT: `RUT_VERBOSITY` and `RUT_HDD_MOUNTPOINT`

`RUT_VERBOSITY` can be set to a number in the range -1 to 3. The default is 0 only showing error messaged. 1 enables warnings, 2 enables info messages and 3 enables all debugging output. -1 makes the library absolutely silence, even in case of a fatal error. This value affects only the messages printed to `stderr` not the behavior of the functions.

`RUT_HDD_MOUNTPOINT` specifies the mountpoint of the partition to be traces when rutSources::HDD_READ or rutSources::HDD_WRITE is enabled. Currently you can trace only one partition at the same time.

Note: The mount mountpoint must be specified without a trailing '/'

Data Structures

- struct rutSources_s

  Bit field for sources to trace.

Defines

- #define RUT_MAX_NUM_CPUS
- #define RUT_MAX_NUM_NETIFS
- #define RUT_MAX_STATS_VALUES
- #define RUTSRC_UNSET_ALL(sources)
- #define RUTSRC_SET_ALL(sources)
- #define RUTSRC_SET_CPU(sources)
- #define RUTSRC_SET_MEM(sources)
- #define RUTSRC_SET_NET(sources)
- #define RUTSRC_SET_HDD(sources)
- #define RUT_SUCCESS
- #define RUT_EMEMORY
- #define RUT_EHDLIB
- #define RUT_ETHREAD

Typedefs

- typedef struct UtilTrace_s UtilTrace
- typedef struct rutSources_s rutSources

Functions

- int rut_createTrace (hdTopoNode *topoNode, int topoLevel, rutSources sources, int interval, UtilTrace **trace)
- int rut_startTrace (UtilTrace *trace)
- int rut_stopTrace (UtilTrace *trace)
- int rut_finalizeTrace (UtilTrace *trace)

Define Documentation

#define RUT_EHDLIB

Error in HDTrace library.

#define RUT_EMEMORY

Out of memory.

#define RUT_ETHREAD

Cannot create tracing thread.

#define RUT_MAX_NUM_CPUS

Maximum number of CPUs supported.

#define RUT_MAX_NUM_NETIFS

Maximum number of network interfaces supported.

#define RUT_MAX_STATS_VALUES

Maximum number of statistics values traceable.

#define RUT_SUCCESS

Success.

#define RUTSRC_SET_ALL(sources)

Macro for setting all available sources.

#define RUTSRC_SET_CPU(sources)

Macro for enabling tracing of all CPU statistics at once.

#define RUTSRC_SET_HDD(sources)

Macro for enabling tracing of all hard disk statistics at once.

#define RUTSRC_SET_MEM(sources)

Macro for enabling tracing of all memory statistics at once.

#define RUTSRC_SET_NET(sources)

Macro for enabling tracing of all NET statistics at once.

#define RUTSRC_UNSET_ALL(sources)

Macro for cleaning all available sources.

Typedef Documentation

rutSources

Type definition of tracing sources bit field.

UtilTrace

Type definition of utilization trace object.

Function Documentation

int rut_createTrace (hdTopoNode * *topoNode*, int *topoLevel*, rutSources *sources*, int *interval*, UtilTrace ** *trace*)

Create utilization trace.

Use this function to create a new utilization trace. It will setup a new statistics trace with the values specified in *sources* to trace.

The tracing will not start until rut_startTrace is called for the UtilTrace object returned by this function.

Parameters:

topoNode The new trace will we associated to this node or one of its ancestor nodes (those on the nodes path)

topoLevel level of the node to associate the new trace to

sources Sources to trace

interval Interval to trace in milliseconds. Should be at least 100ms for acceptable performance impact.

trace Location to store the UtilTrace pointer (OUTPUT)

Returns:

Error state

Return values:

RUT_SUCCESS Success

RUT_EMEMORY Out of memory

RUT_EHDLIB Error in HDTrace library

RUT_ETHREAD Cannot create tracing thread

int rut_finalizeTrace (UtilTrace * *trace*)

Finalize utilization trace object.

Finalizes the trace. This destroys the UtilTrace object an frees all memory.

Parameters:

trace Utilization trace to finalize

Returns:

Error state

Return values:

RUT_SUCCESS Success

RUT_EMEMORY Out of memory

int rut_startTrace (UtilTrace * *trace*)

Start utilization tracing.

Parameters:

trace Trace object to work on

Returns:

Indicate if tracing state changed

Return values:

0 Tracing is now started (was stopped before)

1 Tracing is started (as it was already before)

2 *trace* is NULL

int rut_stopTrace (UtilTrace * *trace*)

Stop utilization tracing.

Parameters:

trace Trace object to work on

Returns:

Indicate if tracing state changed

Return values:

0 Tracing is now stopped (was started before)

1 Tracing is stopped (as it was already before)

2 *trace* is NULL

C.2 Data Structure Documentation

C.2.1 rutSources_s Struct Reference

Detailed Description

Bit field for sources to trace.

Data Fields

- unsigned int CPU_UTIL: 1
- unsigned int CPU_UTIL_X: 1
- unsigned int MEM_USED: 1
- unsigned int MEM_FREE: 1
- unsigned int MEM_SHARED: 1
- unsigned int MEM_BUFFER: 1
- unsigned int MEM_CACHED: 1
- unsigned int NET_IN_X: 1
- unsigned int NET_OUT_X: 1
- unsigned int NET_IN_EXT: 1
- unsigned int NET_OUT_EXT: 1
- unsigned int NET_IN: 1
- unsigned int NET_OUT: 1
- unsigned int HDD_READ: 1
- unsigned int HDD_WRITE: 1

Field Documentation

unsigned int rutSources_s::CPU_UTIL

aggregated utilization of all CPUs

unsigned int rutSources_s::CPU_UTIL_X

CPU utilization for each single CPU.

unsigned int rutSources_s::MEM_USED
amount of main memory used

unsigned int rutSources_s::MEM_FREE
amount of free main memory

unsigned int rutSources_s::MEM_SHARED
amount of shared main memory

unsigned int rutSources_s::MEM_BUFFER
amount of main memory used as buffer

unsigned int rutSources_s::MEM_CACHED
amount of main memory cached

unsigned int rutSources_s::NET_IN_X
incoming traffic of each single network interface

unsigned int rutSources_s::NET_OUT_X
outgoing traffic of each single network interface

unsigned int rutSources_s::NET_IN_EXT
aggregated incoming traffic of external network interfaces

unsigned int rutSources_s::NET_OUT_EXT
aggregated outgoing traffic of external network interfaces

unsigned int rutSources_s::NET_IN
aggregated incoming traffic of all network interfaces

unsigned int rutSources_s::NET_OUT
aggregated outgoing traffic of all network interfaces

unsigned int rutSources_s::HDD_READ
amount of data read from hard disk drives

unsigned int rutSources_s::HDD_WRITE
amount of data written to hard disk drives

D Sources of the Shell Script verifyRUT

```
#!/bin/bash

set -b

REMOTEHOST=node08

echo [ 0s] start stressing first CPU with 100%
taskset 0x1 awk 'BEGIN { i = 0; while(1) i++; }' &

sleep 10
echo [10s] start stressing second CPU with 100\%
taskset 0x2 awk 'BEGIN { i = 0; while(1) i++; }' &

sleep 10
echo [20s] stop stressing first CPU
kill %?0x1

sleep 10
echo [30s] stop stressing second CPU
kill %?0x2

sleep 10
echo [40s] allocate 512MB of memory
./cmalloc 512 &

sleep 10
echo [50s] allocate another 256MB of memory
./cmalloc 256 &

sleep 10
echo [60s] free all memory
jobs
killall ./cmalloc

sleep 9
```

```
ssh -n $REMOTEHOST "nc -l -p 7666 > /dev/null" &
sleep 1
echo [70s] start stressing external network interface by sending 512MB
dd bs=16M count=32 if=/dev/zero 2> /dev/null | nc $REMOTEHOST 7666 &

sleep 9
killall -SIGINT nc
nc -l -p 7666 > /dev/null &
sleep 1
echo [80s] start stressing external network interface by receiving 512MB
ssh -n $REMOTEHOST "dd bs=16M count=32 if=/dev/zero 2> /dev/null | nc
    $HOSTNAME 7666" &

sleep 10
echo [90s] start stressing local hard disk drive by writing 512MB
killall -SIGINT nc
dd if=/dev/zero of=/tmp/verifyRUT.tmp bs=16M count=32 2> /dev/null &

sleep 10
echo [100s] allocate all memory to clean cached data
./cmalloc 1024 &

sleep 10
killall -SIGINT cmalloc
echo [110s] start stressing local hard disk drive by reading 512MB
dd if=/tmp/verifyRUT.tmp of=/dev/null bs=16M 2> /dev/null &

sleep 10
echo [120s] terminate program

rm -f /tmp/verifyRUT.tmp
```

Listings

List of Figures

List of Tables

References

[1] Mujtaba Talebi. *Computer Power Consumption Benchmarking for Green Computing*. Master's thesis, Department of Computing Sciences, Villanova University, April 2008.
http://www.csc.villanova.edu/~tway/publications/talebi_thesis_2008_green_computing.pdf

[2] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan and Christos Kozyrakis. *JouleSort: A Balanced Energy-Efficiency Benchmark*. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*. Beijing, China, June 2007.
http://csl.stanford.edu/%7Echristos/publications/2007.jsort.sigmod.pdf

[3] M. Marcu, M. Vladutiu, H. Moldovan and M. Popa. *Thermal Benchmark and Power Benchmark Software*. 2006.
http://hal.archives-ouvertes.fr/hal-00171376/en/

[4] *EnergyBench Benchmark Software*. Internet.
http://www.eembc.org/benchmark/power_sl.php

[5] Wu-Chun Feng. *The Importance of Being Low Power in High-Performance Computing*. Cyberinfrastructure Technology Watch Quarterly (CTWatch Quarterly), 1: pp. 12–20, August 2005.
http://sss.lanl.gov/pubs/CTWatch-Quarterly-Excerpt.pdf

[6] Barry A. Cipra. *SC2002: A Terable Time for Supercomputing*. SIAM News, 2/36, March 2003.
http://www.siam.org/pdf/news/297.pdf

[7] Hartmut Mix. *TUD - ZIH - Vampir - Visualization and Analysis of Parallel Applications*. Internet.
http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/vampir

[8] Hartmut Mix. *TUD - ZIH - Open Trace Format (OTF)*. Internet.
http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/otf

[9] Matthias Jurenz. *TUD - ZIH - VampirTrace*. Internet.
`http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/vampirtrace/index_html/document_view?cl=en`

[10] *KOJAK*. Internet.
`http://www.fz-juelich.de/jsc/kojak/`

[11] *KOJAK*. Internet.
`http://icl.cs.utk.edu/kojak/`

[12] *SCALASCA*. Internet.
`http://www.fz-juelich.de/jsc/scalasca/`

[13] *SCALASCA*. Internet.
`http://icl.cs.utk.edu/scalasca`

[14] Zoltán Szebenyi, Brian J. N. Wylie and Felix Wolf. *SCALASCA Parallel Performance Analyses of SPEC MPI2007 Applications*. vol. 5119 of *LNCS*, (pp. 99–123). Springer, 2008.

[15] *ParTec Cluster Competence Center*. Internet.
`http://www.parastation.com/`

[16] Thomas Ludwig, Stephan Krempel, Julian M. Kunkel, Frank Panse and Dulip Withanage. *Tracing the MPI-IO Calls' Disk Accesses*. In *PVM/MPI* (edited by Bernd Mohr, Jesper Larsson Träff, Joachim Worringen and Jack Dongarra), vol. 4192 of *Lecture Notes in Computer Science*, (pp. 322–330). Springer, 2006. ISBN 3-540-39110-X.

[17] Thomas Ludwig, Stephan Krempel, Michael Kuhn, Julian M. Kunkel and Christian Lohse. *Analysis of the MPI-IO Optimization Levels with the PIOViz Jumpshot Enhancement*. In *PVM/MPI* (edited by Franck Cappello, Thomas Hérault and Jack Dongarra), vol. 4757 of *Lecture Notes in Computer Science*, (pp. 213–222). Springer, 2007. ISBN 978-3-540-75415-2.

[18] Julian M. Kunkel and Thomas Ludwig. *Bottleneck Detection in Parallel File Systems with Trace-Based Performance Monitoring*. In *Euro-Par* (edited by Emilio Luque, Tomàs Margalef and Domingo Benitez), vol. 5168 of *Lecture Notes in Computer Science*, (pp. 212–221). Springer, 2008. ISBN 978-3-540-85450-0.

[19] Julian M. Kunkel, Yuichi Tsujita, Olga Mordvinova and Thomas Ludwig. *Tracing Internal Communication in MPI and MPI-I/O*, 2009. Submitted for PDCAT 2009.

[20] Yuichi Tsujita, Julian Martin Kunkel, Stephan Krempel and Thomas Ludwig. *Tracing Performance of MPI-I/O with PVFS2: A Case Study of Optimization*, 2009. Accepted for ParCo'09.

[21] Sun Microsystems Inc. *Code Conventions for the Java™ Programming Language*. Internet, April 1999.

http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[22] Tom Tromey Gary V. Vaughan, Ben Elliston and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. Internet. http://sources.redhat.com/autobook/autobook/autobook_toc.html

[23] Free Software Foundation. *GNU Autoconf - Creating Automatic Configuration Scripts*. Internet. http://www.gnu.org/software/autoconf/

[24] Free Software Foundation. *Automake - GNU Project - Free Software Foundation (FSF)*. Internet. http://www.gnu.org/software/automake/

[25] Free Software Foundation. *GNU Libtool - The GNU Portable Library Tool*. Internet. http://www.gnu.org/software/libtool/

[26] Dimitri van Heesch. *Doxygen*. Internet. http://www.stack.nl/~dimitri/doxygen/

[27] Dimitri van Heesch. *Doxygen Manual*, 2009. http://www.stack.nl/~dimitri/doxygen/manual.html

[28] Linux man-pages project. *TERMIOS(3) - manual page for the TERMIOS interface*, 2007.

[29] ZES ZIMMER Electronic Systems GmbH, Tabaksmühlenweg 30, D-61440 Oberursel (Taunus), FRG. *Programmer's Guide*, March 2004.

[30] The GNOME Project. *Libgtop Reference Manual for libgtop 2.22.3*. http://library.gnome.org/devel/libgtop/2.22/

[31] The GNOME Project. *GLib Reference Manual for GLib 2.16.6*. http://library.gnome.org/devel/glib/2.16/

[32] Linux man-pages project. *time(7) - overview of time and timers*, 2008.

[33] ZES ZIMMER Electronic Systems GmbH Oberursel/Germany. *ZES ZIMMER Electronic Systems GmbH - Power Analyzing and Net Quality Analysis*. Internet. http://www.zes.com/index_e.html

[34] Standard Performance Evaluation Corporation. *SPECpower_ssj2008*. Internet. http://www.spec.org/power_ssj2008/

[35] Standard Performance Evaluation Corporation. *SPEC – Power and Performance - User Guide - SPECpower_ssj2008 V1.00*, December 2007. http://www.spec.org/power_ssj2008/docs/SPECpower_ssj2008-User_Guide.pdf

[36] Standard Performance Evaluation Corporation. *SPEC – Power and Performance - Design Overview - SPECpower_ssj2008 V1.01*, August 2008.

http://www.spec.org/power_ssj2008/docs/SPECpower_ssj2008-Design_overview.pdf

[37] *HPC Challenge Benchmark*. Internet. http://icl.cs.utk.edu/hpcc/index.html

[38] Piotr Luszczek, David Bailey, Jack Dongarra, Jeremy Kepner, Robert Lucas, Rolf Rabenseifner and Daisuke Takahashi. *The HPC Challenge (HPCC) Benchmark Suite*. Internet, November 2006. Presentation Slides. http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/sc06_hpcc.pdf

[39] A. Petitet, R. C. Whaley, J. Dongarra and A. Cleary. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. Internet, September 2008. http://www.netlib.org/benchmark/hpl/

[40] *DGEMM Benchmark*. Internet. https://computecanada.org/page/138/EN

[41] *STREAM Benchmark*. Internet. https://computecanada.org/page/141/EN

[42] Rolf Rabenseifner and Gerrit Schulz. *Effective Bandwidth (b_eff) Benchmark*. Internet. https://fs.hlrs.de/projects/par/mpi//b_eff/