Comput Sci Res Dev
DOI 10.1007/s00450-011-0193-x

SPECIAL ISSUE PAPER

Towards an energy-aware scientific I/O interface

Stretching the ADIOS interface to foster performance analysis and energy awareness

Julian M. Kunkel · Timo Minartz · Michael Kuhn · Thomas Ludwig

© Springer-Verlag 2011

Abstract Intelligently switching energy saving modes of CPUs, NICs and disks is mandatory to reduce the energy consumption.

Hardware and operating system have a limited perspective of future performance demands, thus automatic control is suboptimal. However, it is tedious for a developer to control the hardware by himself.

In this paper we propose an extension of an existing I/O interface which on the one hand is easy to use and on the other hand could steer energy saving modes more efficiently. Furthermore, the proposed modifications are beneficial for performance analysis and provide even more information to the I/O library to improve performance.

When a user annotates the program with the proposed interface, I/O, communication and computation phases are labeled by the developer. Run-time behavior is then characterized for each phase, this knowledge could be then exploited by the new library.

Keywords Scientific I/O API · Energy efficiency · ADIOS · Performance analysis · Performance optimization

1 Introduction

Newer hardware devices offer sophisticated power management with various states—each with different energy and performance characteristics. Aim of those energy saving modes is to reduce the energy footprint. Some of these capabilities are hidden within the device while others are disclosed to the operating system via an interface. For instance, when a CPU is idle in modern processors the microarchitecture reduces the voltage and frequency automatically or turns off parts of the electronic by putting it into a so-called *C-State*. In the same fashion storage and communication subsystems can be put into energy saving modes.

It is mandatory to switch the states intelligently, because switching between two states takes some time—while changing CPU states is fast, changing the state of a hard disk and network device is in the order of seconds. Usually, the operating system manages the state of deep sleep of CPU, network and communication devices to ensure they are available to process their work when needed. As the operating system does not know the future workload, often historic knowledge about the utilization is extrapolated to the future. For example, a disk is spun down when it was idle for 10 minutes. In *High Performance Computing* (HPC) even small interruptions, for example, by changing energy saving modes of a CPU, can cause noise which could hinder synchronization with 1000 other processors, thus those hardware capabilities are often disabled.

Developers on the other hand, can predict the activity of their program more accurately. This especially true in HPC environments, where in most cases only one application runs on one node. Therefore, if the developer indicates the future activity to the operating system, the operating system can control those devices in an efficient manner.

However, developers do not see the need to instrument the code to trigger specific hardware energy modes. Therefore, we leverage an existing I/O interface, requiring less code rewrite by the developers. Also, the I/O interface will benefit by those extensions because it can perform back-

J.M. Kunkel (✉) · T. Minartz · M. Kuhn
Department of Informatics, University of Hamburg, Hamburg, Germany
e-mail: kunkel@dkrz.de

T. Ludwig
DKRZ GmbH & Department of Informatics, University of Hamburg, Hamburg, Germany

Published online: 31 August 2011

ground operations more efficiently. In detail, we propose to combine the *CIAO* API into the existing *ADIOS* interface.[1] In brief, ADIOS replaces existing interfaces aiming to improve I/O performance and usability. It is already used in large scale scientific applications. Small modifications to the ADIOS API enable us to control the energy modes of the devices automatically. Additionally, those interfaces also strengthen performance analysis.

When a user annotates the program with the proposed CIAO interface, I/O, communication and computation phases are labeled by the developer. On the one hand this information is exploited by ADIOS to optimize the I/O—for example, by performing write-behind—, on the other hand it could assist performance analysis—that is, users could assess phases individually. Lastly the interface could announce expected utilization to the OS which in turn could control the hardware accordingly.

Seeing all the potential gains by such an API developers could be convinced to instrument their code accordingly.

The structure of this paper is as follows. In Sect. 2, related work and the state of the art are discussed. The original ADIOS interface is presented in detail in Sect. 2.1. In Sect. 3, the proposed interface extension is introduced. Benefits for trace analysis tools and energy-saving mechanisms are discussed in Sects. 4 and 5. In Sect. 6, the paper is concluded and ideas for future work are presented.

2 Related work

As the aspects covered by this paper are threefold, related work on controlling energy saving modes, performance analysis and the parallel I/O interface ADIOS is provided.

Our project—called Energy-Efficient Cluster Computing [14][2]—aims at making high performance computing more efficient with respect to economic and ecological aspects. Its basic idea is to determine relationships between the behavior of parallel programs and their impact on the energy consumption of the underlying compute cluster. Strategies will be developed to reduce the energy consumption with as little impact as possible on program performance. In addition to measuring and analyzing program behavior our tools will be enhanced to record energy-related metrics as well. Based on this new energy efficiency analysis, the users can insert energy control calls into their applications which will allow the operating system and the cluster job scheduler to control the cluster hardware in an energy-efficient way. This paper is supplementary to the eeClust project, the introduced API will control the hardware with the services provided by eeClust.

Next, related work for performance analysis of parallel application is briefly mentioned. The localization of an performance issue on a existing system is a process in which a hypothesis is supported by measurement and theoretic considerations—in praxis due to the complexity of the software mostly measurements are conducted. Measurements are performed by executing the program while monitoring run-time behavior of the application and the system.

Popular post-mortem performance analysis tools are *TAU* [17], *Vampir* [9] and *Scalasca* [4]. All of those tools provide several ways to assist a developer to assess application behavior. Typically, a GUI visualizes the actual behavior of the individual processes over time, or it summarizes system metrics for each function in a so called profile. *Periscope* [5] and *PerfExpert* [1] are automatic tools which perform online scans of performance properties—appropriate metrics are measured and evaluated directly, ultimately locating the bottlenecks to some extent. Many performance analysis tools allow to group a sequence of instructions together in a phase with a user-defined label. For example, with TAU profiles can be created per phase, thus the user can analyze phases of different activity separately.

These profiles or trace files can further be examined in terms of energy efficiency. Free et al. [3] divide the trace files into blocks whereat a block is a set of executed statements demarcated by MPI operations and memory pressure[3] changes. Two adjacent blocks are merged into a phase if their corresponding memory pressure is within the same threshold. They execute each phase with different DVFS settings and select the right setting based on a user-weighted energy-time trade off. Hotta et al. [6] use the same approach, but they use the EDP (Energy Delay Product) as the metric for selecting the right setting. Further they instrument their application manually into phases.

To identify the (low utilization) phases it is also possible to monitor performance counters [2], to analyze the current MIPS of the processor [7] or to perform an interval based workload characterization based on processor stall cycles due to off-chip activities [8]. There are many further approaches for phase-detection, but this is out of the scope of this work.

2.1 ADIOS interface

The *Adaptable IO System* (ADIOS) [10, 11] provides an abstract I/O API and library, which decouples application logic from the actual I/O setting. Several best practices are realized within the ADIOS library to increase usability and

[1] Although, CIAO can be considered an extension of ADIOS, for clarification we use the term CIAO to refer to it.

[2] eeClust—http://www.eeclust.de/.

[3] Memory pressure changes are indicated by L3 cache misses.

performance, for instance aggressive write-behind is performed, and MPI collectives transfer file information to decrease the burden on metadata servers.

By using the API the developer specifies the variables and attributes which should be accessed in an XML file, a tool generates C or Fortran code to call the library. Each write call is annotated in ADIOS with names which can be referred to in the XML file. The amount of data accessed, datatype[4] and further attributes are defined in the XML.

The I/O interface and parameters for file access to perform the actual I/O are selected in the XML file, too. Available modules include NetCDF, HDF5, MPI (collective or independent), POSIX and several asynchronous staging modules. Settings can be defined without changing code, for example, the buffer size can be altered. An advantage of the decoupling of the underlying I/O procedure is that the best fitting implementation can be selected for a group of files—on one system the POSIX interface shows best performance, while on another system the MPI module is advantageous. It is also possible to specify the NULL method which discards I/O.

Moreover, data could be forwarded to a visualization system—even multiple I/O methods can be selected to visualize and store data at the same time. Similar to *SIONlib*, the system is capable to either write a shared file or to split logical I/O into several file system objects, therefore, the new *BP* file format is proposed. ADIOS ships tools to edit and convert BP files into HDF5 and NetCDF files.

The API provides functions to the programmer to indicate when the computation starts or ends, or where the scientific application main loop occurs (`adios_end_iteration()`) to indicate the speed of an iteration. On the one hand, this enables efficient write-back of data to the servers without disturbing application communication, on the other hand the pace in which data is created and written back is announced to the library. Concluding, ADIOS provides a completely new API in which the programmer is forced to deal with I/O related aspects consciously—but due to the XML system, optimizations are possible without source code modifications.

In Listing 1 an MPI example is sketched in which an iterative algorithm loops through computation, communication (here `MPI_Barrier()`) and then writes the computed results into the file "testfile.bp"—the data types are defined in the group "fullData". Accessed data is a 3-dimensional matrix, the dimensions are defined in the NX, NY and NZ variables respectively. The write calls to store one iteration of the data are automatically generated from the XML file by a tool (see Listing 2). Once every 5 iterations a checkpoint is written which contains also the whole data—for simplicity the same data is written in this example, in a more realistic example the checkpoint would contain all variables needed to restart the application.

Listing 3 shows the XML file which defines two I/O groups, the "fullData" group contains a time-series and the variables NX, NY, NZ and the matrix. An attribute describing the data in more detail is also given. Datatypes of each variable and the corresponding names in the file and the C code (`gwrite` attribute) are defined. ADIOS is capable to automatically generate histograms from the data—the histogram is generated for each stored group, in our case for each timestep of the full matrix, this can be done by specifying the analysis tag in the XML file, again without recompiling the application. For each group the I/O method is defined, here both, the "checkpoint" and "fullData" group use the MPI backend, we could replace that with POSIX or NULL to discard all I/O. In the last tag the size of the write-behind buffer is given.

[4]Elementary datatypes and arrays of arbitrary dimension are supported.

Listing 1 Sketched ADIOS code

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include "mpi.h"
 4 #include "adios.h"
 5
 6 int main (int argc, char** argv) {
 7     int         rank, size, i, j, k, t;
 8     int         NX = 10, NY = 10, NZ = 100;
 9
10     double      matrix[NX][NY][NZ];
11
12     MPI_Comm    comm = MPI_COMM_WORLD;
13
14     int         adios_err;
15     uint64_t    adios_groupsize, adios_totalsize;
16     int64_t     adios_handle;
17
18     MPI_Init(&argc, &argv);
19     MPI_Comm_rank(comm, &rank);
20
21     adios_init("example.xml");
22
23     for (t = 0; t < 10 ; t++) {
24         adios_start_calculation();
25         /* computation */
26         adios_stop_calculation();
27
28         /* MPI communication */
29
30         adios_open(&adios_handle, "fullData", "testfile.bp", t == 0 ? "w":
               ↪ "a", &comm);
31 #include "gwrite_fullData.ch"
32         adios_close(adios_handle);
33
34         if (t == 5) {
35             adios_open(&adios_handle, "checkpoint", "testfile.bp", "a",
                   ↪ &comm);
36 #include "gwrite_checkpoint.ch"
37             adios_close(adios_handle);
38         }
39
40         /* indicate progress for write-behind */
41         adios_end_iteration();
42     }
43
44     adios_finalize(rank);
45
46     MPI_Finalize();
47
48     return 0;
49 }
```

Listing 2 ADIOS example code—`gwrite_fullData.ch`

```
1 adios_groupsize = 4 \
2                 + 4 \
3                 + 4 \
4                 + 8 * (NX) * (NY) * (NZ);
5 adios_group_size (adios_handle, adios_groupsize, &adios_totalsize);
6 adios_write (adios_handle, "NX", &NX);
7 adios_write (adios_handle, "NY", &NY);
8 adios_write (adios_handle, "NZ", &NZ);
9 adios_write (adios_handle, "matrix_data", matrix);
```

Listing 3 ADIOS example code XML file

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="fullData" coordination-communicator="comm"
          ↪ time-index="iteration">
        <var name="NX" type="integer"/>
        <var name="NY" type="integer"/>
        <var name="NZ" type="integer"/>
        <attribute name="description" path="/fullData" value="Global array
              ↪ of memory data" type="string"/>
        <var name="matrix_data" gwrite="matrix"  type="double"
              ↪ dimensions="iteration,NX,NY,NZ"/>
    </adios-group>

    <analysis adios-group="fullData" var="matrix_data" min="0"
          ↪ max="3000000" count="30"/>


    <adios-group name="checkpoint" coordination-communicator="comm">
        <var name="NX" type="integer"/>
        <var name="NY" type="integer"/>
        <var name="NZ" type="integer"/>
        <var name="matrix_data" gwrite="matrix"  type="double"
              ↪ dimensions="NX,NY,NZ"/>
    </adios-group>

    <method group="fullData"    method="MPI"/>
    <method group="checkpoint"  method="MPI"/>

    <buffer size-MB="80" allocate-time="now"/>
</adios-config>
```

3 CIAO interface

The CIAO interface and library extends ADIOS in some important aspects, specifically computation phases and communication phases are now annotated by the user. Further it stretches the concept of so-called phases into named phases. A phase is a sequence of code with one goal specified by its label, repeated invocations of the same phase should show similar characteristics in respect to computation, I/O and communication. Depending on the bottleneck, phases are classified into computation bound, I/O bound or communication bound. CIAO also should maintain information to characterize the phase, that is, the demand on CPU, network and I/O resources and the estimated length of the phase. More information about the characterization of phases is found in Sect. 3.2.

The original ADIOS only provides the `adios_end_iteration()` function to indicate the end of an iteration. Obviously, this only works for application with very regular iterations. Doing pre-processing, post-processing or checkpointing every n iterations can not be handled in this way. Introducing phases allows the library to make better predictions by delivering more information about the application's structure. For example, the library could detect that every n-th iteration a checkpoint is written. This could be used to do write-behind over the next n iterations if no communication is happening at the same time.

Calculation and I/O is associated with exactly one phase. The calculation phase is indicated with the new function `ciao_start_calculation()`, which takes the phase name as its only argument. The I/O phase is indicated implicitly by using the group name provided in the appropriate `ciao_open()` call. The `ciao_open()` and `ciao_close()` functions are just thin wrappers around the ADIOS counterparts.

Because the end of the iteration can be detected by using the calculation phase, this also makes it possible to remove the need for `adios_end_iteration()`. That is, whenever a previously seen calculation phase is entered again, the iteration has ended. With the calculation and I/O phases potential communication phases are implicitly derived.

Listing 4 shows the application from Listing 1 modified to use the CIAO interface. In lines 3–5, `ciao_open()` and `ciao_close()` are used read the input data. `ciao_start_calculation()` and `ciao_end_calculation()` is used in lines 7–9 to indicate that some form of calculation is happening during the `pre-processing` phase. In line 12, `adios_start_calculation()` is replaced by `ciao_start_calculation()` to signify the start of the `iteration` phase. This phase is then ended by `ciao_end_calculation()` in line 14. Communication phases are indicated by `ciao_start_communication()` and end respectively. By using the additional knowledge provided by the `ciao_open()` calls in lines 20 and 25 it is possible to handle these I/O operations more efficiently. The code blocks on lines 31–33 and 35–37 work analogous to the input and pre-processing blocks.

Listing 4 CIAO example code

```
adios_init("example.xml");

ciao_open(...);
/* read input */
ciao_close(...);

ciao_start_calculation("pre-processing");
/* pre-process input */
ciao_end_calculation();

for (t = 0; t < 10 ; t++) {
    ciao_start_calculation("iteration");
    /* computation */
    ciao_end_calculation();

    ciao_start_communication("exchange-neighbor");
    /* communication */
    ciao_end_communication();

    ciao_open(&adios_handle, "fullData", "testfile.bp", t == 0 ? "w": "a",
          ↪ &comm);
#include "gwrite_fullData.ch"
    ciao_close(adios_handle);

    if (t == 5) {
        ciao_open(&adios_handle, "checkpoint", "testfile.bp", "a", &comm);
#include "gwrite_checkpoint.ch"
        ciao_close(adios_handle);
    }
}

ciao_start_calculation("post-processing");
/* post-process output */
ciao_end_calculation();

ciao_open(...);
/* write output */
ciao_close(...);

adios_finalize(rank);
```

3.1 Triggered activity by the library

The relation between the phases and potentially triggered activity of CIAO is shown in Table 1.

During an I/O phase data is either read or written, in case data is read, then the actual data must be requested from the (parallel) file system. ADIOS aims to cache write operations to enable write-behind during the iterative computation phase—during a computation bound phase the network

Table 1 Process phases and triggered activity

Phase bottleneck	I/O activity	Network activity	Potential energy savings
Computation	–	Write-behind to I/O servers	I/O and NIC
Communication	–	–	I/O and CPU
Input/Output	Access data and/or buffer data	Read data if necessary	CPU and NIC

is not utilized, thus the data can be staged to the I/O servers. In case the buffer does not suffice to keep all data, then it is forced to actually write data in the I/O phase, thus communication to the servers is needed.

When a communication phase starts, that is, no computation happens, then all background activity must pause until the communication phase completes—this ensures that communication bandwidth is available to potential communication activity.

CIAO exploits this information by triggering energy modes of (un)required devices for a phase. During communication and I/O phases usually the CPU is not utilized to a high extent, thus, the frequency could be reduced via DVFS. ADIOS aggressively caches data for write-behind and tries to write-out data during computation phases, once the data is staged on the I/O servers NICs and I/O devices could be put into an energy saving mode (often energy saving modes are also referred to as ACPI Device Power States).

For devices such as disks and NICs of some network technology which require seconds to change states it is important to estimate the benefit before the state change is triggered. Therefore, the time until the device is needed again must be approximated and indicated by CIAO. More details about how CIAO fosters energy efficiency are provided in Sect. 5.

3.2 Characterization of phases

In order to control the devices appropriately characteristics of phases must be available in CIAO. Two kind of phases can be distinguished: a *regular phase* shows very predictable characteristics and varies only slightly, thus it can be estimated easily by using little *historic knowledge*, for example just using the last characteristics might be a very good approximation. However, *irregular phases* reveal major differences in their characteristics depending on the state of the program. For instance, this could be caused by repartitioning of the workloads in finite element methods or by load-balanced applications.

There are two methods to tackle this issue, either the user provides hints to the characteristics or CIAO detects characteristics by itself. The former requires the user to carefully indicate bottlenecks and thus bears a burden to the user. The latter is problematic for irregular patterns, because there is a magnitude of potential patterns. Thus, we propose that the user can provide hints to CIAO via the XML file to indicate the estimation module which is used for approximation. Those hints could be embedded into the ADIOS XML as illustrated in Listing 5—for each phase the estimation method is selected. General characteristics of the estimation are specified as attributes of the *estimation* tag. To estimate valuable switches CIAO must not understand the cost for switching between states of the hardware, however, this information must be available (for example in the eeClust daemons). Thus, CIAO just tells the daemons an estimate for the duration in which a resource is not used and how much performance is needed, then the daemon switches the device states if profitable. It is possible to compute the threshold within CIAO, though.

Several modules are envisioned, to give an impression: a *NULL* module which does not trigger any background operation and prohibits to change into energy saving states. This avoids estimates in case almost random behavior is expected. The *MIN* module measures the duration and uses the minimum time observed during the runtime as an estimate for the future duration. Thus, MIN is useful for regular patterns and it tries not to overestimate characteristics. With *HISTORIC* statistics are stored into a file (or database) and are reused for subsequent invocations of the program, similar to profile-guided-optimization. This is very useful if an application is run repeatedly with similar parameters.

With the `debug` attribute, statistics or debugging information of the phases can be printed at program termination, it is also possible that a module warns the user if the assumption provided in the XML file do not hold, that is, the historic knowledge does not match the observations.

Fast transitions between phases bear a problem to the interface, as the I/O and energy hints change rapidly which causes overhead and prohibits switching of energy metrics. This could also be mitigated by wrapping the medium-grained phases into larger phases including the bottlenecks.[5] In our example for instance, both, the checkpointing and iterative write-out could be wrapped into `ciao_start_io()` and `ciao_end_io()`, this way the indicated energy hints could be reduced. Also, characteristics for the projected sequence of phases could be estimated guided by the transitions between phases. Those inter-phase statistics could

[5] Remember that it is allowed to perform little communication and/or computation in all phases.

be handled by additional modules, a *STOCHASTIC* module for instance could estimate the probability for transitions to other phases and if the phases transits to another phase in more cases than specified in the threshold, then the transition is performed and lengthens the current phase.

There is much literature available in which the estimation of application runtimes is investigated with historic knowledge, for instance in [15, 18]. Also, machine learning has been applied to provide better estimates of future invocations. Several approaches exist to merge phases together, however, in contrast to our proposed extension they are all built either on artificially introduced "phases" or onto new interfaces, and are not part of an approach which is especially designed to improve I/O performance. Thus, all those mechanisms could be integrated into CIAO.

Listing 5 CIAO XML snippet specifying phase estimators

```
1  <adios-config host-language="C">
2     ...
3     <buffer size-MB="80" allocate-time="now"/>
4     ...
5
6     <estimation debug="statistics">
7       <inter-phase method="STOCHASTIC" accept-threshold="95%">
8       <phase name="iteration"       method="MIN"/>
9       <phase name="post-processing" method="HISTORIC"/>
10    <estimation/>
11 </adios-config>
```

4 Benefit for analysis tools

The concept of phases allows usability improvements in trace analysis tools like Sunshot, Vampir or gprof. The phases can be integrated into the traces and then used by the respective tool to present a more user-friendly output. For example, Fig. 1 shows the visualization of an application as produced by the tool Sunshot. As can be seen, the vertical lines show the beginning (and end) of each iteration. Additionally, a checkpointing phase can be observed. This information can be added automatically using the knowledge of the application's phases as recorded by the CIAO interface.

It is also possible to use the additional information to reduce noise when analyzing traces. Because the length and structure of each phase is known, (mostly) identical iterations can be skipped, showing only deviant ones to the end-user. For example, this can be useful when analyzing load imbalances, because it allows the user to concentrate on the interesting parts of the trace without having to manually find them.

In tools which analyze code metrics such as time, hardware counters or energy metrics the instrumentation of phases enables to aggregate those metrics within certain phases. Further, phases could be analyzed with statistical methods. For instance profiling tools like gprof could visualize the time spent in each phase (see Listing 6).

Some performance analysis tools already offer the capability to identify repeated phases and cluster data, yet

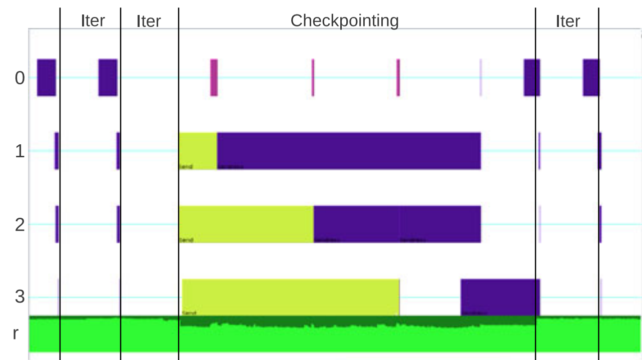

Fig. 1 Tracing MPI activity and node power consumption

they usually do this by looking at the hardware counters and/or function invocation order. The named phases introduced with CIAO are provided by the user and thus allow clustering on a higher abstraction with semantics defined by the user, yet the user must not instrument the library just for the purpose of performance analysis.

Listing 6 Proposed gprof output

```
1  Phase profile
2
3   Overview
4     %     cumulative   self              self
5    time     seconds   seconds    calls   s/call  phase name
6   60.08      48.06     48.06      10      4.80   iteration
7   20.02      64.08     16.02       2      8.01   checkpoint
8   10.04      72.11      8.03      10      0.80   exchange-neighbor
9    8.02      78.53      6.42       1      6.42   post-processing
10   1.80      79.97      1.44       1      1.44   pre-processing
11   0.04      80.00      0.03       -      0.03   [unlabeled]
12
13   Phase "iteration"
14     %   cumulative   self              self     total
15    time   seconds   seconds    calls   s/call   s/call  name
16   69.37     33.33     33.33      10    18.62    33.33  calculateValues
17   30.63     48.06     14.72      10    14.72    14.72  calculateOffset
18
19   Phase "checkpoint"
20     %   cumulative   self              self     total
21    time   seconds   seconds    calls   s/call   s/call  name
22  100.00     20.02     20.02       2    20.02    10.01  writeMatrix
23     ...
```

5 An interface fostering energy efficiency

From an energy efficiency point of view the additional information provided by the developer can be used to make better decisions when setting hardware device states. For example, spinning up and down hard disk drives takes a considerable amount of time. Their energy consumption is also higher during this period. Thus, it only makes sense to do this if they are not used for an extended period of time.

Using the knowledge of the application's phases, it is possible to predict whether possible actions are beneficial or not. A major goal is to identify the phases where switching to low power modes is profitable—for example in case of device idleness, memory-boundness or busy-waiting. The energy consumption to turn into another energy saving mode and to transit into the required mode for the next phase must be computed and compared with the savings during

a phase—if the latter dominates, then switching energy saving mode is advisable. In brief, the minimal power consumption without reducing the time to complete the operation is searched—components only switch to a lower energy saving mode if the component utilization is below a certain threshold and can be woken up before more performance is required.

The decision whether a component switches to another energy saving mode depends on different factors: The duration of the phase, the power saving potential of the state change (P_{diff}, difference of power consumption between power saving states), the duration of the state change (t_{change}, which is the time to switch to the better energy saving mode of this phase and the energy saving mode required by the next phase) and the energy of the state change (E_{change}, the sum of both changes). Note that in case the saving mode of the next state can not be anticipated, then the maximum performance state must be chosen. With these values the minimal duration of the phase t_{phase} can be calculated, for which switching energy saving mode is advicable (see 1). In case the selected mode matches the requirements of the phase, that is, if CIAO manages to estimate the performance demand correctly, then those transitions do not increase wall-clock time. Some more details about changing states are available in [13].

$$t_{phase} = \frac{E_{change}}{P_{diff}} + t_{change} \quad (1)$$

The identification of phases is partly addressed by the developer, who indicates the areas of demand with CIAO, which estimates the duration and characteristics of those phases. If CIAO detects that during an I/O phase only buffered writes happen, then the disk can still be off during that phase, while an I/O phase which performs read operations will need an active I/O subsystem. Also, with the knowledge of the phases it is possible to reduce performance of the components according to the demand, for example, the CPU frequency while checkpointing [12].

With profund knowledge about the future phases (as discussed in Sect. 3.2) characteristics of those could be incorporated into the calculation. As this would lengthen the time in which devices are not required with maximum performance this knowledge would enable to control devices even better.

5.1 Saving energy by controlling hardware

To identify the general power saving potential, we measured the hardware of our power-aware cluster in multiple operating and idle states. Our cluster consists of five dual socket Intel Nehalem (Xeon X5560, 4 cores + Hyperthreading) and five dual socket AMD Magny-Cours (Opteron 6168, 12 cores) computing nodes. Each of the processors is DVFS

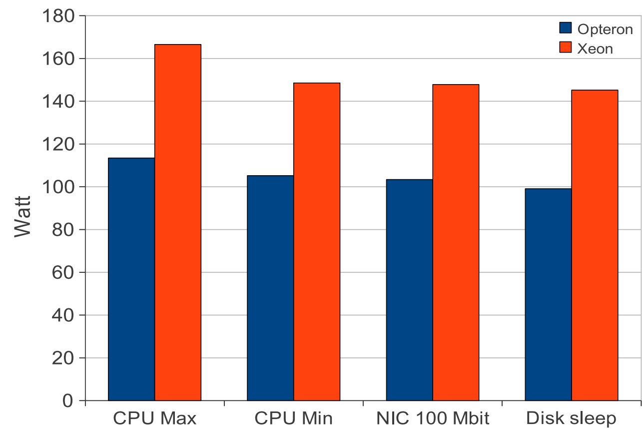

Fig. 2 Idle power consumption for Opteron and Xeon nodes depending on hardware device states

enabled and supports multiple performance states (P-States) and idle states (C-States). Further the hard disks[6] and the network interface cards[7] of each node support transitions to low power or reduced performance states (D-States). Additionally, the cluster has two I/O subsystems with five disks, one with HDDs and one with SSDs. The computation nodes and the I/O nodes are connected with Gigabit Ethernet. To measure the power consumption of the hardware, each node and the Gigabit switch are connected to ZES LMG450 high precision power meters with a accuracy of about 0.1%. The power consumption of each node is stored in a database on the head node, to whom all power meters are connected via serial ports.

Figure 2 visualizes the measured power savings for our specific hardware for an idle node [12]. For the Opteron nodes, we can save up to about 11% power while we can save about 18% with the Xeon nodes. Switching the device state of network card and disk result in a decreased power consumption of about 6% compared to using only frequency scaling. Adjusting the processor frequency only seems to be promising in phases of load imbalance, MPI communication, I/O or memory-boundness, because the performance decreases faster than the energy-efficiency increases. This should be even more the case for the network card, because the power consumption decreases to 40% while the speed decreases to 10% (when switching from 1000 Mbit to 100 Mbit, switching to 10 Mbit is even worse, see Table 3). Entering the sleep state of the disk makes sense only if the disk is idle for a longer period of time. Unfortunately, there are only two sleep states (standby and sleep) supported, yet. But when entering the sleep mode, the disk can reduce its power consumption to about 18% (see Table 2).

[6] http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369b.pdf.

[7] http://download.intel.com/design/network/datashts/82574.pdf.

Table 2 Seagate Barracuda ST3500418AS power consumption

Mode	Power (W)
Idle	5.0
Operating	6.57
Standby	0.79
Sleep	0.79

Table 3 Intel 82574 NIC power consumption

Speed (Mbit/s)	Power active (mW)	Power idle (mW)
1000	878	642
100	351	190
10	416	167
no link	–	44

Table 4 Transition times

Transition	Time (milliseconds)
CPU P-State	0.01
NIC Speed	4000
Disk Spinup	8500

Table 4 shows the transition times of the manageable devices. Each P-State transition of the processor takes about 10000 nanoseconds as estimated by common operating systems. A speed change of the network card results in a transition time of about 4 seconds, independent of the concrete transition. The transition time for the disk is even higher, about 8.5 seconds.

Taking the device state power consumption and the transition time into account, energy savings with slight performance degradation is possible for HPC applications as shown in recent works [3, 6, 13, 16].

6 Conclusion and future work

In summary, it is safe to say that high-level interfaces like ADIOS and CIAO offer great possibilities to provide even better support for trace analysis and energy saving while requiring very little additional work from the application developers. A huge portion of the necessary information can be inferred from the normal ADIOS and CIAO calls. With the extensions to mark computation and communication dominant phases background I/O activity could be scheduled even more efficiently than with ADIOS.

A future goal is to modify the ADIOS interface to support the proposed CIAO extensions. The preliminary work described in this paper is just a first step towards integration. Proposed extension would make it possible to use energy-saving mechanisms like those developed by the eeClust project without additional instrumentation. Further research to automatically detect phase transitions and for phase characterization can be inferred from existing projects and integrated into the system.

Acknowledgements This work has been partially funded by the BMBF (German Federal Ministry of Education and Research) under grant 01|H08008E within the call: “HPC-Software für skalierbare Parallelrechner.”

References

1. Burtscher M, Kim BD, Diamond J, McCalpin J, Koesterke L, Browne J (2010) Perfexpert: An easy-to-use performance diagnosis tool for HPC applications. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, SC ’10. IEEE Computer Society, Washington, DC, pp 1–11. doi:10.1109/SC.2010.41
2. Freeh V, Lowenthal D, Pan F, Kappiah N, Springer R, Rountree B, Femal M (2007) Analyzing the energy-time trade-off in high-performance computing applications. IEEE Trans Parallel Distrib Syst 8:1575–1590
3. Freeh VW, Lowenthal DK (2005) Using multiple energy gears in MPI programs on a power-scalable cluster. In: PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 164–173. doi:10.1145/1065944.1065967
4. Geimer M, Wolf F, Wylie BJN, Abraham E, Becker D, Mohr B (2010) The Scalasca performance toolset architecture. Concurr Comput 22(6):277–288
5. Gerndt M, Ott M (2010) Automatic performance analysis with periscope. Concurr Comput 22:736–748. doi:10.1002/cpe.v22:6
6. Hotta Y, Sato M, Kimura H, Matsuoka S, Boku T, Takahashi D (2006) Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In: IPDPS ’06: proceedings of the 20th international parallel and distributed processing symposium (2006). doi:10.1109/IPDPS.2006.1639597
7. Hsu CH, Feng WC (2005) A power-aware run-time system for high-performance computing. In: SC ’05: proceedings of the 2005 ACM/IEEE conference on Supercomputing. IEEE Computer Society, Washington, pp 1. doi:10.1109/SC.2005.3
8. Huang S, Feng W (2009) Energy-efficient cluster computing via accurate workload characterization. In: CCGRID ’09: proceedings of the 2009 9th IEEE/ACM international symposium on cluster computing and the grid. IEEE Computer Society, Washington, pp 68–75. doi:10.1109/CCGRID.2009.88
9. Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller MS, Nagel WE (2008) The Vampir performance analysis tool-set. In: Tools for high performance computing, proceedings of the 2nd international workshop on parallel tools. Springer, Berlin, pp 139–155
10. Lofstead J, Klasky SKS, Podhorszki N, Jin C (2008) Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). http://www.adiosapi.org/uploads/clade110-lofstead.pdf
11. Lofstead J, Zheng F, Klasky S, Schwan K (2009) Adaptable, metadata rich IO methods for portable high performance IO. In: Proceedings of IPDPS’09, May 25–29, Rome, Italy. Springer, Berlin

12. Minartz T, Knobloch M, Ludwig T, Mohr B (2011, will be published) Managing hardware power saving modes for high performance computing
13. Minartz T, Kunkel J, Ludwig T (2010) Simulation of power consumption of energy efficient cluster hardware. Comput Sci Res Dev 25:165–175. doi:10.1007/s00450-010-0120-6
14. Minartz T, Molka D, Knobloch M, Krempel S, Ludwig T, Nagel W, Mohr B, Falter H (2011, will be published) eeClust—Energy-efficient cluster computing
15. Minh TN, Wolters L (2010) Using historical data to predict application runtimes on backfilling parallel systems. In: Euromicro conference on parallel, distributed, and network-based processing, pp 246–252. http://doi.ieeecomputersociety.org/10.1109/PDP.2010.18
16. Rountree B, Lowenthal DK, Funk S, Freeh VW, de Supinski BR, Schulz M (2007) Bounding energy consumption in large-scale MPI programs. In: SC '07: proceedings of the 2007 ACM/IEEE conference on supercomputing. ACM, New York, pp 1–9. http://doi.acm.org/10.1145/1362622.1362688
17. Shende SS, Malony AD (2006) The tau parallel performance system. Int J High Perform Comput Appl 20(2):287–311. http://doi.acm.org/10.1007/s00450-011-0193-x
18. Smith W, Foster IT, Taylor VE (1998) Predicting application run times using historical information. In: Proceedings of the workshop on job scheduling strategies for parallel processing. Springer, London, pp 122–142. http://portal.acm.org/citation.cfm?id=646379.689526

Julian M. Kunkel received his M.Sc. degree in computer science at the University of Heidelberg in 2007. Employed at the German High Performance Computing Centre for Climate- and Earth System Research he conducts research to improve performance of parallel applications. His interests cover parallel file systems, MPI middleware, and modeling of cluster systems' performance.

Timo Minartz received his M.Sc. degree in computer science at the University of Heidelberg in 2009. Now he is a research scientist at the University of Hamburg and contributes to the eeClust project. His major research interests are energy-efficiency aspects in high performance computing.

Michael Kuhn received his M.Sc. degree in computer science at the University of Heidelberg in 2009. Currently, he is employed at the University of Hamburg and works towards his Ph.D. His research interests are in high performance input/output, file systems and distributed systems in general.

Thomas Ludwig became Professor at the Ruprecht-Karls-Universität Heidelberg in 2001 and lead the research group Parallel and Distributed Systems. Since 2009 he is Professor at the university of Hamburg and CEO of the German High Performance Computing Centre for Climate- and Earth System Research. His major research interests are high performance storage and energy efficiency in HPC.