Steering of Sequential Jobs with a Distributed Shared Memory Based Model for Online Steering [1]

Daniel Lorenz [a] Peter Buchholz [a] Christian Uebing [a] Wolfgang Walkowiak [a]
Roland Wismüller [a]
[a] *University of Siegen, Germany*

Abstract

Online steering systems allow to retrieve application data like intermediate results for visualization, and to modify parameters during the runtime of the application. While most steering systems use a client/server paradigm, online steering can favorably be modeled as a distributed shared memory with concurrent access by the application and the online steerer. In this paper, this idea is formalized, focusing on the exploration of the consistency models and protocols for the distributed shared memory. The behavior of the steering system is described by consistency models, which also guarantee the data integrity of the application, both within a single process and between multiple application processes. Depending on the integrity requirements, applications can choose the proper model and protocol. The performance of our protocols is evaluated with a synthetic workload, which shows that the newly developed delayed weak consistency is faster than the special weak consistency. Furthermore, the results prove that the invalidate protocols of both consistency models are able to adapt themselves to the workload.

Key words: online steering, distributed shared memory, consistency model, RMOST

1. Introduction

In recent times, scientific simulations increased both in complexity and in the amount of data they produce. Often the simulations run on batch systems in clusters or computational Grids and do not support interactivity during the runtime of the simulation. Online steering of an application enables the visualization of intermediate results, performance data, or other application data, and the invocation of actions, e.g. modification of parameters by the user at runtime of the job. The user can interactively explore parameter realms, debug a program, or optimize performance. Because the user sees results earlier, the user can evaluate results earlier and react before the job has finished. Thus, online steering accelerates scientific research and saves ressources.

In this work application means the steered program. A steerer is distinguished from a steering system. A *steerer* is the interface to the user which visualizes data and offers the user the possibility to enter commands, e.g. modifications of a parameter. A *steering system* comprises all extentions to the application, external components, specialized steerers, and extentions to offline visualization tools to enable steering.

Until now, various steering systems have been developed [3,5,8–11,18,20] for supporting the scientist with interactive control over his simulation. Existing systems provide means to retrieve data from the application and invoke actions in the remote application. Typically, the application is instrumented with

[1] This work is partly funded by the *Bundesministerium für Bildung und Forschung* (BMBF) as part of the German e-Science Initiative (contract 01AK802E, HEP-CG)

calls to a steering library to enable the sending of data to a remote visualizer, or to apply commands issued by the user. The steering system manages the data transport to a customized user interface. One of the reasons why steering systems are not widely used is the required effort to instrument a large application for steering.

In this paper another approach is used, which views steering similar to distributed shared memory (DSM). Any steering is basically the change of state of an application and a state change corresponds to a change in memory. Thus, steering can be modeled as a case for DSM because all steering actions can be reduced to memory access operations.

The consistency model used in a DSM system defines the order in which each process sees memory operations. This defines the value which a read operation must return. A DSM system is consistent according to a consistency model if all read operations return the values defined by the consistency model.

Every application process assumes that its data is consistent to some application specific consistency model. Based on this assumption, semantic relations between data objects can exist which are essential for correct execution of the application and meaningful results. Data *integrity* means the consistency and the correctness of semantic relationships between data object inside the application.

Data integrity is an important prerequisite to obtain correct results from the application. This means a steering system should ensure that the displayed data is consistent in itself, and any modifications must preserve the integrity of the data within the application. If a data object is modified in a running application without any synchronization with the execution of the applications, severe errors may occur and the data integrity can be broken. To protect the integrity, rules are needed which define the order of access operations applied to the shared data. The necessary rules define a consistency model for the steering system. In this paper, consistency is related to the interoperability between application and steering system while integrity is related to the correctness inside the application only.

A *protocol* for a consistency model is a description of the communication and algorithms which implements the consistency protocol. The two main strategies used for consistency protocols are the invalidate and the update stategy.

Though various steering systems exist, until now no consistency model for online steering exists. In most cases the integrity problem is not addressed or left to the user.

Based on the DSM-based steering model, the new online steering system RMOST (Result Monitoring and Online Steering Tool) [14] was developed. It demonstrates the ability of the DSM-based approach to easily apply steering to existing applications and even use existing offline visualization tools for online visualization.

By abstraction to a consistency model, the used data exchange protocol implementation is hidden. For a single consistency model several protocol implementations are possible which have all the modeled behavior but may use different strategies. Thus, the protocol implementations can be exchanged transparently, e.g. switching from a push strategy to a pull strategy.

2. Formalism for the DSM Based Model for Online Steering

In online steering, the application and the steerer both access the same data. If the steerer and the application run in the same address space, this is a trivial task. But if the steerer and simulation belong to different address spaces, e.g. if they are located on different machines, a mechanism to access the remote data is needed. Because two processes access the same data, online steering can be modeled as DSM. The advantages of the DSM model are that the complexity of distributed data is hidden from the user of the steering system, and it appears to be accessing only local data for the steerer and the application. The steering system handles the communication and it supports the programmer with the consistency guarantees to maintain data integrity.

In the DSM based model of online steering, two kinds of processes exist with different roles and properties. Firstly, n application processes $p_1, ..., p_n$ exist. The application may synchronize $p_1, ..., p_n$ with any mechanism, e.g. via messages, or shared memory. However, the synchronization within the application is out of the scope of this work. Secondly, m steering processes $p_{n+1}, ...p_{n+m}$ exist, each representing a steerer in a collaborative environment. The data objects which can be visualized or steered reside in the distributed shared memory. Each data object x has a home location $H(x) \in p_1, ..., p_n$ which is one of the application processes. The steering processes are not chosen for home locations, because the steerers may detach and thereby causing the home location to become inaccessible.

Three kinds of memory operations exist: read operations r, write operations w, and synchronization operations s. Read and write operations are denoted as $o(p, x, v)$ where $o \in \{w, r\}$ specifies the operation type, p is the process that performs the operation, x is the memory location, and v is the value that is written or read. Synchronization operations are denoted as $s(p)$. A process p_i is viewed as a sequence of memory operations $S_i = \{o_1, o_2, ...\}$ with $o_i \in \{w, r, s\}$. A process *sees* a write operation w if a current read operation would return the value written by w. A write operation w is *visible* to a process p if p can see w.

Each application process p_i is associated with a logical clock T_i, which indicates the progress of the process. T_i is incremented at synchronization operations. An *epoch* is the interval between two consecutive clock increments. The time $T_{min} = \min(T_1, ..., T_n)$ is the minimum time of all application processes. The time $T_{max} = \max(T_1, ..., T_n)$ is the maximum time of all application processes.

3. Data Integrity Conditions

In this section, the effects that might affect the data integrity are analyzed which lead to two integrity conditions. The first one is the *intra-process* condition, and the second one is the *inter-process* condition.

3.1. *The Intra-Process Condition*

The intra-process condition requires that the data in the application must not be modified externally during certain operation intervals, and that the write operations of the application to shared objects become only visible if the data is in a well-defined state. For example, assume one formula is computed where one variable x appears at different places in the formula. The result can only be correct if the value of x stays the same during the whole computation. Another case could be a numerical n-body simulation. While it is allowed to modify parameters between each simulated time step, the value should stay the same inside each time step.

Also the modifications by the application to shared data should become visible only at well defined places. Imagine several properties of different input objects are computed. If the object is visible and displayed after the computation of the first few properties while the other properties stem from another input, the displayed result is propably incorrect and can be misleading. Thus, to preserve the intra-process condition, changes of the application must only become visible at well defined points, and changes by the steerer must only be applied by the application at well-defined *synchronization* points (SP). Typically, one epoch is bounded by two SP, which implies that SPs match the incrementations of the logical clock.

3.2. *The Inter-Process Condition*

The inter-process condition considers differences in progress of different processes. Firstly, it requires that write operations of the steering processes must be seen in all processes at the same time step. Secondly, values of displayed data objects must stem from the same epoch.

For example, suppose a parallel simulation iterates over several time steps and each process computes a part of the overall result. If changing a boundary parameter, one would like to change this parameter for all processes in the same epoch. If a steerer changes the value of the parameter in the DSM, the system must ensure that the modification is viewed by all processes at the same epoch.

Another case occurs if a steerer wants to display a distributed object which is modified by several processes, and each process computes a part of the whole object. The steerer must only see the write operations of all processes up to T_{min} to retrieve an internally consistent data set.

4. Consistency Models

To ensure the integrity of the data in online steering, each process must view access operations to the shared memory according to certain rules. For each given set of access operations, a consistency model is defined through the possible orders in which each process is allowed to see the memory accesses [19]. Thus, a consistency model can be used to maintain data integrity. In this section, consistency models are evaluated which fulfill the requirements for data integrity in online steering. One consistency model will not satisfy all cases, because not all data objects require both integrity requirements analyzed in Sec. 3. Some data objects require no integrity conditions and can be treated completely asynchronous, some data objects require only the intra-process condition, and some data objects require both conditions.

Thus, different consistency models are appropriate in each of these cases. The case that data objects require only the inter-process condition is not considered, because the inter-process condition implies the existence of epoches. The transition points between two epoches define the SPs where values may be read or modified.

4.1. *Consistency for the Intra-Process Condition*

In case the application uses message passing, each application-process has its own memory. If the application uses shared memory, it must have its own consistency model which can not be changed by the steering system. The shared memory is accessed through one process. Because in the intra-process condition no requirements for synchronization between application processes must be considered, communication happens only between one application-process and one steerer. In case multiple processes exists, each application process and the steerer have a shared memory disjunct from the shared memory of another application process and the steerer.

The intra-process condition allows the application and distribution of updates only at specified SPs. The desired existence of specified SPs leads to two new consistency models which are similar to weak consistency [6]. While the application should always see its most recently written value, two possibilities exist for the behavior of the steerer, which do both implement the intra-process condition. Formally the difference occurs in the following case:

Let p_1 be an application process and let p_2 be a steering process that viewed the accesses $w(p_1, x, 2)$, and $s(p_1)$. Now, p_2 executes $s(p_2), w(p_2, x, 1), r(p_2, x, ?)$ before it sees another $s(p_1)$. Which value should $r(p_2, x, ?)$ return?

4.1.1. *Special Weak Consistency*

$r(p_2, x, 1)$ returns the value recently written by the same process. This leads to weak consistency with the modification that updates are applied exactly at the next synchronization operation, instead of *latest* at the next synchronization operation. Many implementations do fullfill this stricter definition, because of performance reasons (aggregation of modifications). If two modifications by a steering process and a application process conflict, the modification of the steerer is given priority. This model is called *special weak consistency* (SWC).

4.1.2. *Delayed Weak Consistency*

$r(p_2, x, 2)$ returns the current value of the application. The modeled behavior is the one of a steerer which only issues a command to change a value to the application and sees its own modification only after the application confirmed the modification. In this case, the modification by the steerer is distributed at the next SP of the steerer. After the application received the modification command, it applies the modification at its next SP and returns a confirmation to the steerer. The steerer applies the new value at the next SP after receiving the confirmation of an application process. Instead of confirming single modifications, the application can confirm all modifications distributed by a SP at once. The execution order of access operations performed by p_2 until it reads $r(p_2, x, 1)$ would be:

$$w(p_1, x, 2), s(p_1), s(p_2), w(p_2, x, 1), r(p_2, x, 2),$$
$$s(p_2), s(p_1), s(p_2), r(p_2, x, 1)$$

The SP needs to only be Pipelined RAM consistent [13], because the only request to the consistency points is that other processes see the consistency points of one process in the order they occur in that process. The SPs of different processes may be seen in different orders by different processes.

This model delays the visibility of the write operation after the next synchronization operation, thus it is called *delayed weak consistency* (DWC).

Interestingly, the sequential consistency [12] is too strong for the intra-process condition. Sequential consistency requires that all processes see all write operations in the same order. In most DSM systems, the usage of relaxed consistency models is driven by the better performance of the relaxed models compared to strong consistency models, but the programmer wants his program to behave as if sequential consistency [12] would be used. In the case of online steering, strong consistency would not provide the desired behavior.

4.2. *Consistency for Both Conditions*

In this case it must be ensured that the steerers retrieve all values from the same epoch, and all application processes apply all modifications at the same epoch. At every given time, each epoch can be assigned to one of the following three groups:

– The *past* are those epoches T that are finished by all application processes: $T < T_{min}$.

– The *future* are those epoches T that are not yet entered by any application process: $T > T_{max}$.
– The *presence* are the epoches T that do neither belong to the past nor to the future: $T \in [T_{min}, T_{max}]$.

Each write operation w is tagged with a time stamp $T(w)$. Write operations of an application process p will be tagged with the timestamp of the process $T(w) = T(p)$. Write operations of steering processes will be tagged with $T_{max} + 1$. Thus, each data object has a schedule of values assigned to it. A read operation of data object x by a steering process will always return the most recent value v from the past. Read operations of an application process p at time step $T(p)$ will always return the most recent value v from the viewpoint of the process.

This consistency model is called *schedule consistency*. Steerers can only write to the future and read from the past. It has the effect, that modifications are not seen immediately, but after a delay which depends on the length of the presence. The delayed weak consistency (DWC) is a special case of the schedule consistency with the presence comprising only one epoch. Formally, this effect is caused by an reordering of write and read operations in the steering processes. Write operations that occur before a read operation in the program order may be seen later than the read operation. This consistency model is based on a algorithm from CUMULVS [17] but was not defined as a consistency model before.

4.3. *Consistency with no Integrity Conditions*

Besides parameters or results which probably have the intra-process or inter-process condition, data objects with a producer-consumer access pattern exists which require none of the integrity conditions. These data objects have one producer, which is the only process writing to these data objects, and one or more consumer processes who read this data object. For example, processor load or other monitoring data has neither the intra-process nor the inter-process condition. For those data the update intervals or delays implied by the weak or schedule consistency may be inappropriate. These data objects are independent from other data objects by definition, thus Pipelined RAM consistency [13] should be sufficient. Pipelined RAM consistency ensures that all processes see the write operations of a process p in the order they are executed by p.

5. Consistency Protocols

The development of the protocols focuses on the consistency models for the intra-process condition. In the intra-process condition the shared memory exists between the steerer and one application process.

In all described protocols, each process has a local copy of the shared data object in its local memory. Furthermore, it is assumed, that all messages are received in the order they are sent.

5.1. *Update Protocol for the SWC*

In the SWC, updates for a modified data object are distributed at the next SP. Thus, a write operation sets a modified flag for the data object and stores the new value in the local copy.

At a SP, the new values of all received updates are stored in the local copy. Then the current values of all modified data objects are distributed and the modified flag is cleared. If a process receives an update, the value is buffered until the next SP, where the new value is applied. Read operations return always the current value of the local copy.

A conflict occurs in a SP if a process has modified a data object x in the last epoch and received an update (or invalidate) message for x in the same epoch. If a conflict occurs in a SP of the steering process, updates are discarded. If a conflict occurs in an application process, the update is applied and the modification flag is cleared, preventing the update for this data object to be sent.

To ensure the order of the SPs, a synchronization token is exchanged between the processes. A process can only execute a SP if it possesses the synchronization token else it will send a request for it and block until the token is received.

5.2. *Invalidate Protocol for the SWC*

In the invalidate protocol, a write operation on x sets the modification flag for x and stores the new value. At the next SP, an invalidate notification is sent if the current state indicates that the remote process has not invalidated its local copy. Furthermore, the current value of x is copied to $LS(x)$. $LS(x)$ is a memory location which contains the value of x at the last SP.

The invalidate protocol uses the same mechanism for the ordering of the SPs as the update protocol. If

a request for the synchronization token is received, $LS(x)$ is copied to $AKT(x)$ which represents the most recent modification of this process which is visible by the other process.

If the application process receives an invalidation, the local value of the data object is marked as invalid. When a read operation wants to read the invalid data object, it requests an update from the other process and blocks until it receives a response. If the steerer receives a request, it sends the value of $AKT(x)$ to the application. If the steering process receives an invalidation the reaction is analogous.

5.3. *Update Protocol for the DWC*

In the DWC, no total ordering of SPs exists, but each process p has a counter T_p of its local SPs, and $T_p(s)$ is the timestamp of the SP s.

The application and the steerer act differently on a write operation. If the job writes a data object x, the new value is stored in the local copy, and the modification flag is set. At the next SP, the modified data objects are distributed like in the SWC.

If the steering process p modifies an object x, the new value is not stored in the local copy of x, but stored in $LW(x)$ which contains the most recent modification of x. At the next SP s_1, $LW(x)$ is copied to $LS(x, T_p(s_1))$ which contains the value of x at s_1, and $LW(x)$ is sent to the application process q together with $T_p(s_1)$. Because one process may execute several SPs before the modification is applied, multiple values from different SPs can be buffered.

If q receives the updates of s_1, it applies either all updates of s_1 at a single SP or none. If the updates of s_1 are applied, q returns a confirmation which contains $T_p(s_1)$. When p receives the confirmation it knows that q applied all updates made by p until s_1. At its next SP it copies the value from $LS(x, T_p(s_1))$ to the local copy of x.

Conflicts can only occur in the application process when the application process has modified an object and receives an update from the steerer in the same epoch. In this case the update is applied and the modification of the application is discarded.

5.4. *Invalidate Protocol for the DWC*

Because no total ordering of the SPs exist, each process p has its own clock T_p which is increased at each SP of p. At the end of each SP, a `SyncEnd` message is sent which contains the current value of T_p. If the `SyncEnd` message is received by the remote process q, it stores the received value of T_p in R_q. To acknowledge a `SyncEnd` message, q appends R_q to its next `SyncEnd` message in addition to T_q.

In the invalidate protocol for the DWC, each process p must remember the state $VL_p(x)$ of the value of the local copy of x, and $VR_p(x)$ which is the state of the local copy at the remote process of x.

First, it is described when the steering process p performs a write operation to a data object x. In this case, a modification flag is set and the new value is stored in $LW(x)$ like in the update protocol. At the next SP s_1, the value of $LW(x)$ is stored in $LS(x, T_p(s_1))$ which is still the same procedure like in the update protocol. If $VR_p(x)$ is valid, an invalidate message is sent and $VR_p(x)$ is set to invalid.

If the application q receives an invalidate message, it sets $VL_q(x)$ to invalid and $VR_q(x)$ to valid at the next SP s_2 after receiving all data from s_1. At the end of s_2, q acknowledges s_1 by sending a `SyncEnd` message.

When p receives the acknowledgment of s_1, it knows that q has either applied the invalidations from s_1, or the data objects were already invalidated before, and that q sees the values which were written in the epoch before s_1. Because the modifications of p are only transferred on a read operation of q, p must remember $AKT(x)$ which is the last modification q sees. Thus, p stores $LS(x, T_p(s_1))$ in $AKT(x)$. At the next SP after receiving the acknowledgment of s_1, p applies the value of $LS(x, T_p(s_1))$.

A read operation of q returns the local copy of x if $VL_q(x)$ is valid else a request to p is sent which returns $AKT(x)$, and set $VR_p(x)$ to valid. After the response has been received by q, the local copy is updated, $VL(x)$ is set to valid, and the new value is returned. If a process answers to an update request, further modifications can be stored in $LS(x, *)$ from a later SP. For these modifications, no invalidate messages have been sent so far. Thus, if another modification is buffered in LS, an invalidate message with the timestamp of its required application is appended to the update message.

If the application q modifies a data object x, the application stores the new value in the local copy of x. In addition, the modification flag is set, and $VL_q(x)$ is set to valid.

If an invalidate message is received in the current epoch, any modification flag for x is removed and $VL_q(x)$ is set to invalid at the next SP s_3. This ensures the priorization of the steerer in case of a conflict. Else, the current value of x is stored in

$LS(x, T_q(s_3))$. If $VR_q(x)$ is valid, an invalidate message is sent and $VR_q(x)$ is set to invalid. If p receives the invalidation, p sets $VL_p(x)$ to invalid at its next SP after receiving all data from s_3. When q receives the acknowledgment for s_3, it stores $LS(x, T_q(s_3))$ in $AKT(x)$ which is returned on an update request.

Suppose q performs a write operation on x, $VL_q(x)$ was invalid and the next SP is s_5. Then, p could perform another write operation on x and afterwards a SP s_4. In s_4 no invalidate message was sent for x because $VR_p(x)$ was already invalid. If the `SyncEnd` message of s_4 is received by q before q executes s_5, $VL_q(x)$ should be invalidated again, but the invalidate message was not sent.

As a solution, q sends a request for validation approval to p if $VL_q(x)$ is invalid and q modifies x. If p receives an approval request, it sets $VR_p(x)$ to valid, checks for already buffered modifications, and appends the invalidation time if a modification is buffered. If approval requests were sent in an epoch, the next SP blocks until all approvals are received.

The complete protocol is listed in Listing 1. It shows the action on each possible event (read, write, synchronization, and receipt of messages). It is assumed that the reaction procedures of one process are mutual exclusive, except at the `wait` statements where messages can be received and processed.

Listing 1. invalidate protocol for the delayed weak consistency

```
onRead(x):
  if VL(x) = invalid
    send Request(x)
    wait until VL(x) = valid
  return Value(x)
onWrite(x, val):
  if Job
    if VL(x) = invalid
      set VL(x) := valid
      send ApprovalRequest(x)
      set needApproval := true
    set Value(x) := val
  set LW(x) := val
  set Modified(x) := true
onSync:
  T ++
  if Job
    for all x
      wait until needApproval(x) = false
  for all x with Apply(x) = true
    set Apply(x) := false
    set Value(x) := AKT(x)
    set VL(x) := true
  for all (x,t) in BufInv with t <= R
    remove (x,t) from BufInv
    if not (Steerer and Modified(x))
      set Modified(x) := false
    set VL(x) := invalid
    set VR(x) := valid
  for all x with Modified(x) = true
    set LS(x,T) := LW(x)
    if VR(x) = valid
      send Invalidate(x,T)
      set VR(x) := invalid
  send SyncEnd(T,R)
onRecvSyncEnd(SyncNum, DoneSyncs):
  set R := SyncNum
  for all x and t <= DoneSyncs
    if exist LS(x,t)
      set AKT(x) := last LS(x,t)
      delete all LS(x,t)
      if Steerer
        set Apply(x) := true
onRecvInvalidate(x,t):
  add (x,t) to BufInv
onRecvRequest(x):
  if exists LS(x,t)
    send Invalidate(x,t)
  else set VR(x) := valid
  send Update(x, AKT(x))
onRecvUpdate(x, val):
  set Value(x) := val
  set VL(x) := valid
onRecvApprovalRequest(x):
  if exists LS(x,t)
    send Invalidate(x,t)
  else set VR(x) := valid
  send ApproveValidate(x)
onRecvApproveValidate(x)
  needApproval(x) := false
```

6. The RMOST Steering System

RMOST (Result Monitoring and Online Steering Tool) [2] [14] is an online steering system for Grid jobs of the High Energy Physics (HEP) experiment ATLAS [1], which is developed as part of the HEPCG [3] project of Germany's D-Grid initiative [2,15]. It consists of an application independent implementation

[2] RMOST is available at http://hep.physik.uni-siegen.de/grid/rmost

[3] High Energy Physics Community Grid

of the presented DSM approach for online steering, and a thin integration layer into the ATLAS software. Through the DSM-based approach it is possible to enable steering of Grid jobs in the ATLAS experiment without modification of the source code. Currently, SWC and DWC with one steering process are supported.

Its architecture consists of four main layers:

(i) The communication layer realizes a communication channel between the application and the steerer. The Grid communication channel of RMOST is described in [16].

(ii) The data consistency layer implements a DSM system which supports multiple consistency protocols. Currently, protocols for the SWC, DWC, pipelined RAM consistency, and on demand data exchange exist. New protocols can be added by developing a class which implements reactions to possible events like read, write, SP, or the receipt of a message. Each data object must be registered and the used protocol can be defined individually for each data object.

(iii) The data processing layer is a placeholder for any data processing performed by the steering system like filtering, or automated evaluation.

(iv) The data access layer provides tools for data access. For example, in RMOST a preloaded library replaces standard library calls in order to observe file accesses. Another (not yet implemented) possibility is to monitor method calls by modifying a classes' virtual table.

The ATLAS [1] experiment is part of the Large Hadron Collider (LHC) at CERN. Besides many other, the most prominent goal of the ATLAS experiment is to search for the Higgs particle which is responsible for the masses of particles.

The experiment's software framework Athena [7] was created for the reconstruction of the data. The processed data consists of collision events which can be computed independently. In general, the desired results consists of, e.g. histograms with statistics over several thousands events.

The user composes his job from different components, provided by the Athena framework [7]. The two main types of components are called algorithms and tools. The composition is defined in a so called job options file. Furthermore, Athena can be extended with customized components contained in a shared library.

The ROOT toolkit [4] is commonly used for offline visualization of physics results. It provides an interface to extend ROOT with new classes which are located in a shared library and loaded dynamically.

The ROOT toolkit and the Athena framework are huge collections of source code and are already installed at all ATLAS Gridsites. These software is still under development. Source code instrumentation would put a large additional effort on the developers of these packages. Because the software is preinstalled on all ATLAS Gridsites, they do not want another group make modifications for research which could break their production system. Furthermore, general reservations against Online-Steering on a batchsystem in the Grid exist preventing RMOST from being integrated in the Athena framework. For the integration of RMOST in the Athena framework a new component `RM_Spy` was developed which can be applied to the Grid job by editing the job options file.

Steering is made available to ROOT by dynamically loading a library with interface classes for ROOT to RMOST. It allows to adjust steerable parameters, or view progress information from the job. By preloading the RMOST file access library, the steering system intercepts file accesses and fetches or updates the according parts of the file. Thus, ROOT which was intended to be a post-mortem analysis tool, can be used for online visualization of intermediate results and steering without any modifications to the source code.

7. Performance Evaluation

The performance of the described protocols is measured with a synthetic workload. A test application and the steerer registered 5 data objects of the same size and has performed 1000 SPs. In each epoch, the data objects are written or read with a probability of 0.75. If one process has finished its 1000 iterations before the other process has finished, it continues to process SPs without write or read operations until the other process has finished. This experiment was performed for data objects with 10 Bytes and 100 000 Bytes size. The results are shown in Tab. 1. Additionally, the performance of the update protocol of the pipelined RAM consistency is measured which simply transfer the data on each write operation.

Next, the results for 10 Bytes long data objects are discussed. The DWC protocols are significantly faster than the SWC protocols, due to the stricter ordering of the SPs in the SWC. The updated proto-

Protocol	10 Bytes	100000 Bytes
SWC update	117.34 s	412.25 s
SWC invalidate	61.84 s	68.51 s
DWC update	24.16 s	395.82 s
DWC invalidate	46.71 s	45.70 s
PRAM C. update	22.30 s	249.84 s

Table 1
Measured run-time for 1000 iterations of the test program with data objects of 10 Bytes and 100 000 Bytes size.

col of the SWC exchanges the synchronization token between each SP and must wait until it has received the token again. Thus, the processing of all SPs is sequential, and additional time for 1000 roundtrips is contained in the measured time. If the application, and the steerer would have longer intervals between the SPs than 20 ms as for the test programs, the execution of a SP had more overlap with the computing of the code between the SPs of the other process which might reduce the difference between the SWC and DWC update protocols.

The SWC invalidate protocol uses nearly half the time of the SWC update protocol though it uses the same synchronization mechanism. In the invalidate protocol, the read operations block until an answer is received. In this time, no synchronization tokens are requested. Thus, the other process can process multiple SPs in a row without waiting for the synchronization token. In contrast to the update protocol, the invalidate protocol executes sequences of several SPs between each synchronization exchange. The roundtrip times saved are larger than the waiting time for an answer, reducing the run-time.

In the DWC, different processes can not block each other for synchronization. The performance of the DWC update protocol is close to the performance of the PRAM consistency protocol. The invalidate protocol takes nearly twice as long, because of waiting times for read operations.

When the size of the data objects is increased, the run-time of the update protocols also increases due to the higher amount of data transferred. The increase of the data size has only little effect on the performance of the SWC invalidate protocol and the DWC invalidate protocol is not affected at all. Larger data objects imply longer transfer times leading to longer waiting times for read operations. In between, the other process can perform more iterations, of which only the last modification leads to a new invalidation. Thus, invalidations, and blocking update requests are less frequent.

As a consequence invalidate protocols dynamically adapt to the available network bandwidth and data volume, by reducing the update frequency. In case of online steering, where a user wants to see the current state of an application, the user must not see every intermediate step, but gets independent of the data volume and available network bandwidth, the best possible update frequency.

8. Related Work

In this paper a new model for online steering was presented and implemented in RMOST. RMOST is the first DSM-based steering system. However, some steerers provide tools to support the user to maintain the integrity of the data.

Most similar to the presented work is the Pathfinder [10] steering system. Steering actions and the program's execution are both viewed in terms of atomic transactions. They address the issue of consistently applying steering actions to a parallel message passing program. A steering action is consistent if it is applied in a consistent snapshot of the parallel program. An algorithm is presented which detects inconsistent steering actions. The major issue is to define points in a parallel application where steering actions can be consistently applied. As a result a total ordering of all transactions exists, which leads to sequential consistency.

CUMULVS [9,17] is a steerer which allows to make checkpoints of a parallel program. An algorithm is presented that captures distributed data objects consistently by stopping processes that have already processed ahead until all processes reached an equal progress. While this algorithm is similar to the presented schedule consistency, CUMULVS has no DSM-based model for steering.

EPSN [8] requires a description of the structure of the application. For each steerable data object, areas are defined where the data object may be read or changed. The source code of the application must be instrumented with markers to the abstract structure. VASE [5] follows similar principles. The integrity problem is brought to an abstract level which can simplify the problem for the user. However, the decision where a data object may be accessed without disrupting integrity remains with the user. Both have no DSM-based approach for steering.

In RealityGrid [11] a client/server based steering system was developed. The steering library only informs the application on events which must be han-

dled by the user. The user may use predefined library calls to react on events, but a DSM like mechanism does not exist. The steering actions are performed in a single steering library call to reduce the effort of instrumentation. Because events are processed in a single function, by default, weak consistency is implicitly realized.

9. Conclusions and Future Work

In this paper, online steering is viewed as accesses to a distributed shared memory. The behavior of the protocols are defined through the used consistency model. It provides guarantees for data integrity, the possibility to easily apply online steering to existing applications, and to enable offline visualization tool for online visualization.

For each model, several protocol implementations are possible which can be exchanged transparently for the application. For sequential jobs, the delayed weak consistency (DWC) and the special weak consistency (SWC) were developed and protocols with a pull and a push strategy are implemented and evaluated. Hereby, the DWC showed the better performance. The invalidate protocols dynamically adapt to the network bandwidth and to the data volume. In existing event-based steerers the overloading of the bandwidth can be problem which is solved with invalidate protocols.

Because exchanging the protocol for the same consistency model is transparent, it offers the possibility to optimize the performance by selection of the best protocol. E.g., the DWC update protocol has a smaller overhead for small data volumes while the invalidate protocol perfrom better if the data volume is large. This is a possible area of further reseach.

Until now, the focus was on the communication between a steerer and an application process, but the model can also be applied to multi-process applications and collaborative steering. DSM-based steering with multiple processes and evaluation of the schedule consistency is a topic for future research.

References

[1] G. Aad et al. The ATLAS experiment at the CERN large hadron collider. *JINST 3 S08003*, Aug. 2008.

[2] S. Borovac et al. Overview over the high energy physics community Grid in Germany's D-Grid initiative. In *Proc. of the XI Int. Workshop on Advanced Computing and Analysis in Physics Research*, POS(ACAT25)12, Amsterdam, the Netherlands, Apr. 2007.

[3] K. Brodlie et al. Visualization in grid computing environments. In *Proceedings of IEEE Visualization 2004*, pages 155–162, 2004.

[4] R. Brun et al. ROOT - an object oriented data analysis framework. In *Proceedings of AIHENP'96 Workshop*, number A 389 in Nuclear Instruments and Methods in Physics research (1997), pages 81–86, Sept. 1996.

[5] J. D. Brunner et al. VASE: the visualization and application steering environment. In *Proc. of the 1993 ACM/IEEE Conf. on Supercomp.*, pages 560–569, 1993.

[6] M. Dubois et al. Memory access buffering in multiprocessors. In *ISCA '86: Proc. of the 13th annual Int. Symp. on Comp. arch.*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Comp. Society Press.

[7] G. Duckeck (ed.) et al. ATLAS computing: Technical design report. CERN-LHCC-2005-022.

[8] A. Esnard et al. A time-coherent model for the steering of parallel simulations. In *Proc. of Euro-Par 2004*, volume 3149 of *LNCS*, pages 90–97. Springer, 2004.

[9] G. A. Geist et al. CUMULVS: Providing fault-tolerance, visualization and steering of parallel applications. *Int. J. of High Performance Computing Applications*, 11(3):224–236, Aug. 1997.

[10] D. Hart and E. Kraemer. Consistency considerations in the interactive steering of computations. *Int. J. of Parallel and Distributed Systems and Networks*, 2(3):171–179, 1999.

[11] S. Jha, S. Pickles, and A. Porter. A computational steering API for scientific Grid applications: Design, implementation and lessons. In *Workshop on Grid Application Programming Interfaces*, Sept. 2004.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Comp.*, C-28(9):690–691, Sept. 1979.

[13] R. J. Lipton and J. S. Sandberg. PRAM: A scaleable shared memory. Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

[14] D. Lorenz et al. Online steering of HEP Grid applications. In *Proc. of the Cracow Grid Workshop '06*, pages 191–198, Cracow, Poland, July 2007. Academic Computer Centre CYFRONET AGH.

[15] D. Lorenz et al. Job monitoring and steering in D-Grid's high energy physics community Grid. *Future Generation Computing Systems*, 2008.

[16] D. Lorenz et al. Secure connections for computational steering of Grid jobs. In *Proc. of the 16th Euromicro Int. Conf. on Parallel, Dist. and network-based Processing*, pages 209–217, Toulouse, France, Feb. 2008. IEEE.

[17] P. M. Papadopoulos, J. A. Kohl, and B. D. Semeraro. CUMULVS: Extending a generic steering and visualization middleware for application fault-tolerance. In *Proc. of the 31st Hawaii Int. Conf. on System Sciences (HICSS-31)*, Jan. 1998.

[18] R. L. Ribler et al. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.

[19] R. C. Steinke et al. A unified theory of shared memory consistency. *Jour. of the ACM*, 51(5):800–849, 2004.

[20] J. S. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. of the 11th Int. Symp. on Parallel Processing*, pages 128–134. IEEE, 1999.