Modeling and Experimental Validation of the Data Handover API

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modeling and Experimental Validation of the Data Handover API

Soumeya Leila Hernane — Jens Gustedt — Mohamed Benyettou

N° 7493

december 2010

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7493--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Modeling and Experimental Validation of the Data Handover API

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

Domaine : Réseaux, systèmes et services, calcul distribué
Équipe-Projet AlGorille

Rapport de recherche n° 7493 — december 2010 — 14 pages

Abstract: *Data Handover*, DHO, is a general purpose API for an efficient management for locking and mapping data. Through objects called *lock handles*, it enables to control resources in a distributed setting. Such handles ease the access to data for client code, by ensuring data consistency and efficiency at the same time. This paper explores DHO as it was presented in [1]. We model the phases that a lock handle crosses to achieve a DHO locking/mapping life cycle. The Grid Reality And Simulation (GRAS) environment of SimGrid is used as a support of an implementation of DHO and a series of tests and benchmarks of that implementation is presented. GRAS has the advantage of allowing the execution in either the simulator or on a real platform. For that purpose, we exploited a cluster of Grid'5000. The experiments that carried out cover various scenarios of sequences to lock a resources (inclusive or exclusive locking only, or combinations of both) and of combining different architectural factors. The tests demonstrate the ability of DHO to provide a robust and scalable framework. The good evaluation of the present work is consistent with an analysis of the expected behavior done by queuing theory.

Key-words: clustering coefficient graph network random generation

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Modelisation et évaluation expérimentale du API Data Handover

Résumé : *Data Handover*, DHO, est un interface générique de gestion pour verrouiller et mapper des données. Il permets le contrôle de ressources reparties par des objets baptisés *lock handles*. Ces objets facilitent l'accès au données par des codes clients et assurent à la fois la cohérence et l'efficacité. Ce papier explore DHO comme il a été présenté dans [1]. Nous modélisons les phases qu'un *lock handle* poursuit pendant un cycle de vie de DHO. L'environnement GRAS de SimGrid (*The Grid Reality And Simulation*) est utilisé comme support d'une implantation de DHO et d'une série de tests de cette implantation est présentée. D'un coté, GRAS permet une exécution du code dans le simulateur et de l'autre sur plate-forme réelle. Pour ce dernier, nous utilisons une grappe de Grid'5000. Les expériences que nous avons menés couvrent différents scénarios de verrouillage d'une ressource (inclusive ou exclusive, ou une combinaison des deux) et de combinaison de différents facteurs architecturales. Les testes montrent l'aptitude de DHO de fournir un cadre robuste qui passe à l'échelle. La bonne évaluation du travail présent est cohérent avec une analyse par théorie des filles d'attentes du comportement en moyenne.

Mots-clés : coefficient regroupement graphique réseaux génération aléatoire

1 Introduction and Overview

While large-scale distributed systems bring the advantages of an economy of scale, their heterogeneous structure limits the efficient use of system resources. Efficient data utilization in such systems relates to models and paradigms that emanate from two classes of architectures: shared and distributed memory. Both models handle data access and data consistency. However, they operate differently: threads are typically deployed on shared memory based programming environments (*e.g* OpenMP, [2]), while communication libraries based on the MPI API are widely used on distributed memory architectures[3]. Both paradigms have their powers and weaknesses. In this paper, we explore the Data handover (DHO) API that had been presented in [1]. Its goal is to be able operate efficiently in both paradigms as mentioned above.

DHO is designed to handle data on different types of systems such as multi-core machines, mainframes or in a distributed setting as clusters or grids. It enforces consistency of data and allows the application to handle individual parts (called ranges) of large objects. With this approach DHO extends known locking and mapping strategies.

DHO has not yet been implemented completely, the presented implementation is a first step towards this goal. [4, 5] introduced *ordered read-write-locks*, ORWL, for applications with iterative computations and parallel tasks. This model comes close to the locking strategy of DHO if we omit the capacity to lock parts of objects. A multi-threaded version of ORWL for shared memory were added to parXXL [6]. In [7], authors have extended classical distributed lock strategies to provide a byte-range locking in exclusive mode only.

In the following, the term resource is used for an instance of data, usually of large size that forbids an atomic access. The current implementation combines the acquisition of the lock of a resource with the provisioning of a copy of the data. The access to individual ranges of such a data object is not yet implemented.

The main objective of this paper is to measure the ability for an increasing number of requests to easily share a single resource in both *concurrent read* (cr) and *exclusive write* (ew) mode. The rest of this paper is organized as follows. Sec. 2 describes the core features of DHO. We provide an execution pattern by means of a deterministic finite state automaton (DFSA) in Sec. 3. Sec. 4 then presents the evaluation criteria and the experimental framework. Results are reported and discussed in Sec. 5.

2 The DHO Interface

DHO introduces an abstraction level between resource and memory through *lock handles* and according to an access control policy. Application processes (*clients*) attempt to regain access to the resource locally. A resource owner (*server*) deals with requests, manages the resource and ensures its consistency. Clients may hold several handles to the same resource, e.g to regulate a sequence of accesses.

Through the handle, the clients request the instantiation of the resource in their local memory. This triggers events and crosses various states from request insertion until resource release. The state of knowledge about an acquisition (or not) that a client process has is denoted with capitalized names such as *Idle* for its initial state, see Fig. 1. States of the handle itself (that might be hidden to the client) are denoted in all minuscules like *invalid* for the initial state of the handle, see Fig. 2. The first action on a handle has always to be the `dho_create` function. The client attempts to open a socket to a server that holds the resource and waits for a reply. After that reply, the handle switches to the *valid* state.

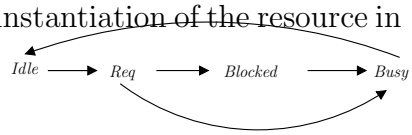

Figure 1: State diagram of a DHO client

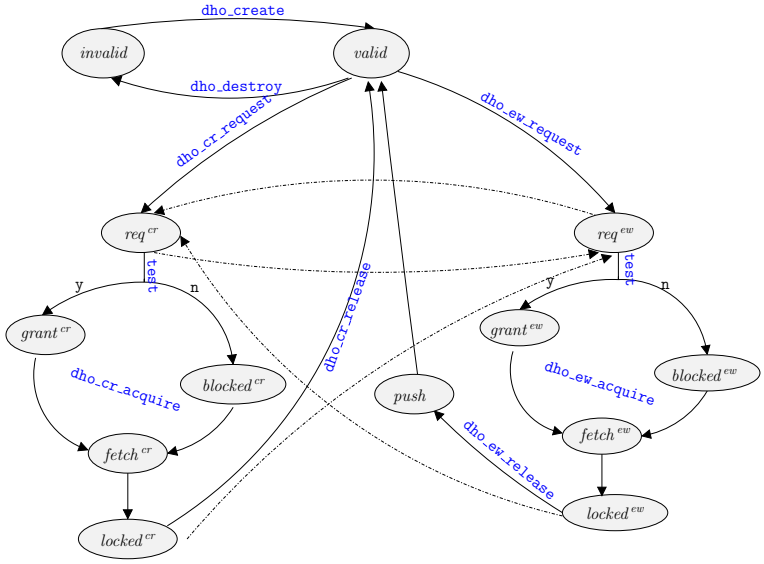

Figure 2: State diagram of the DHO handle implementation

2.1 The DHO life cycle

The main phases of DHO and states through which a client and a handle go are designed to form a cycle (Fig. 2), the life cycle of a DHO locking and mapping request, a *dho cycle* for short. DHO distinguishes two access modes, namely a concurrent read mode (cr) for multiple simultaneous readers and an exclusive write mode (ew) for a single writer. A FIFO strategy assigns the resource to a handle in the same order as the requests are issued.

Phase 1 After an idle time $T_{(\text{idle})}$, the client requests read access to the resource by calling the function `dho_cr_request` with argument `handle`. If a previous ew locking/mapping event is already present for the resource, the request is not serviced immediately, but is placed in a waiting queue. An `ACK` is returned by the function that confirms the registration of the request in the queue. The client switches to the *Req* state in non blocking mode and the handle becomes req^{cr}. A similar action is triggered by `dho_ew_request` for an ew access. In that case, the handle becomes thus req^{ew}.

Phase 2 When the resource is released from all previous read or write locking/mapping that inhibited the access in Phase 1 the handle switches to $grant^{\text{cr}}$ respectively $grant^{\text{ew}}$.

By $T_{(\text{WaitGrant})}$, we will denote the time to achieve this state. The client may at any moment ask to achieve the locking by calling `dho_cr_acquire` respectively `dho_ew_acquire`. If the lock is not yet obtained, the client is *Blocked* until this is the case. Then, the handle becomes $blocked^{\text{cr}}$ or $blocked^{\text{ew}}$, respectively. The time that the client waits until that call will be denoted $T_{(\text{WaitBlocked})}$. As an alternative to blocking, a call to the function `dho_test` can be used to know if access has been granted.

Phase 3 Once access has been granted, the handle switches to the states $fetch^{\text{ew}}$ or $fetch^{\text{cr}}$, respectively. These are intermediate states during which the mapping is effectively done. $T_{(\text{fetch})}$ is the time required for this operation.

In both cases, the handle then switches to the $locked^{\text{cr}}$ respectively $locked^{\text{ew}}$ state and instantiates the resource in local memory. The function returns a pointer to this local copy of the resource and the client becomes *Busy*. We will denote the times before a call to `dho_test` or that the client is *Blocked* $T_{(\text{WaitGrant})}$ and $T_{(\text{blocked})}$.

Phase 4 After an application dependent lock time denoted by $T_{(\text{locked})}$, the client calls `dho_cr_release` or `dho_ew_release`. If the request had been for writing, the eventual modification of the data is propagated to the resource during a transitory *push* state using a time $T_{(\text{push})}$.

Then the resource is notified of the release of the lock and the handle becomes *valid*, again. On return from the call, the client switches again to the *Idle* state.

Thereafter, the client may again call `dho_cr_request` or `dho_ew_request` for the same handle. Listing 1 presents an overview of testing the DHO API.

Listing 1: A simple example of using the DHO API

```
char const* name;
dho_t *a;
double DELAY, T_WaitBlocked, T_Lock;
dho_create(name, &a);
do {
  dho_cr_request(a);
  sleep(T_WaitBlocked);
  dho_test(a);
  dho_cr_acquire(a);
  sleep(T_Lock);
  dho_cr_release(a);
} while(time < DELAY);
dho_destroy(a);
```

At any time it isn't *Blocked*, the client can call `dho_test` to determine the state of the handle. The handle becomes *invalid*, again, by calling the function `dho_destroy` with a duration $T_{(\text{destroy})}$. It important that to note that $T_{(\text{WaitBlocked})}$ and $T_{(\text{locked})}$ are parameters imposed by the application and not observations.

2.2 Other specific cases

In addition to the above, DHO foresees all possible combinations of calls to its interfaces and acts accordingly. A client can call any function at any time and may not respect the logical order of the cycle as we have described above. The effect of calling the functions 'out of order' are that the handle first is returned to the state of *valid* and that all priorities and locks are lost. Then the desired state is achieved as if the necessary other functions had be called previously. If for example the client has issued a read access that is not yet served or if it calls `dho_cr_request` while the process holds a lock and thereafter calls the function `dho_ew_request`, the lock or the old request is immediately canceled by the handle and the new request is inserted with a new `ACK`. The dotted lines in Fig. 2 illustrate such behavior.

3 Formal specification and modeling

The resource can be only in three different states, namely that no request has been issued and that an EW or CR request has been granted. For the states of an DHO handle we already have implicitly introduced a model as Deterministic finite-state automaton (DFSA) [8]. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a 5-tuple, consisting of:

1. A finite set of states Q of a handle, where:

$$Q = \{\mathit{valid}, \mathit{invalid}, \mathit{blocked}^{\mathrm{cr}}, \mathit{fetch}^{\mathrm{cr}}, \mathit{grant}^{\mathrm{cr}}, \mathit{locked}^{\mathrm{cr}}, \mathit{req}^{\mathrm{cr}}, \mathit{blocked}^{\mathrm{ew}}, \mathit{fetch}^{\mathrm{ew}}, \mathit{grant}^{\mathrm{ew}}, \mathit{locked}^{\mathrm{ew}}, \mathit{req}^{\mathrm{ew}}, \mathit{push}\}$$

2. A finite set of input symbols Σ corresponding to the DHO functions, where:

$$\Sigma = \{\texttt{create}, \texttt{destroy}, \texttt{test}, \texttt{request}^{\mathrm{cr}}, \texttt{acquire}^{\mathrm{cr}}, \texttt{release}^{\mathrm{cr}}, \texttt{request}^{\mathrm{ew}}, \texttt{acquire}^{\mathrm{ew}}, \texttt{release}^{\mathrm{ew}}\}$$

3. A start state $q_0 \in Q$ where $q_0 = \mathit{invalid}$

4. A set of accept states $F \subseteq Q$, where $F = \{\mathit{invalid}, \mathit{valid}\}$

5. A transition function $\delta : Q \times \Sigma \mapsto Q_\delta$ is defined by the state transition Table 1.

Let $T_{(\mathrm{DHO})}$ denote the time of the whole DHO cycle. According to Fig. 2, it is easy to point out two possible paths for each access type, through which the handle goes. For example:

$$\{\mathit{req}^{\mathrm{cr}} \rightarrow \mathit{grant}^{\mathrm{cr}} \rightarrow \mathit{fetch}^{\mathrm{cr}} \rightarrow \mathit{locked}^{\mathrm{cr}} \rightarrow \mathit{valid}\}$$

This path occurs when the handle remains non blocking and often with the first requests or in such scenarios with many successive read accesses. The whole DHO cycle time is then:

$$T_{(\mathrm{DHO})} = T_{(\mathrm{WaitGrant})} + T_{(\mathrm{grant})} + T_{(\mathrm{fetch})} + T_{(\mathrm{locked})} + T_{(\mathrm{idle})}. \quad (1)$$

When multiple requests are ahead of current write request, the handle crosses following states:

$$\{\mathit{req}^{\mathrm{ew}} \rightarrow \mathit{blocked}^{\mathrm{ew}} \rightarrow \mathit{fetch}^{\mathrm{ew}} \rightarrow \mathit{lock}^{\mathrm{ew}} \rightarrow \mathit{push} \rightarrow \mathit{valid}\}$$

The corresponding DHO cycle time is:

$$T_{(\mathrm{DHO})} = T_{(\mathrm{WaitBlocked})} + T_{(\mathrm{blocked})} + T_{(\mathrm{fetch})} + T_{(\mathrm{locked})} + T_{(\mathrm{push})} + T_{(\mathrm{idle})}. \quad (2)$$

	valid	req^{ew}	$grant^{\text{ew}}$	$blocked^{\text{ew}}$	$fetch^{\text{ew}}$	$locked^{\text{ew}}$	*push*	*invalid*
`create`	*valid*	*valid*	*valid*	–	–	*valid*	–	*valid*
`request`$^{\text{ew}}$	req^{ew}	req^{ew}	req^{ew}	–	–	req^{ew}	–	*invalid*
`test`	*valid*	req^{ew} | $grant^{\text{ew}}$	$grant^{\text{ew}}$	–	–	$locked^{\text{ew}}$	–	*invalid*
`acquire`$^{\text{ew}}$	*valid*	$blocked^{\text{ew}}$	$fetch^{\text{ew}}$	–	–	$locked^{\text{ew}}$	–	*invalid*
`release`$^{\text{ew}}$	*valid*	*valid*	*valid*	–	–	*push*	–	*invalid*
`destroy`	*invalid*	*invalid*	*invalid*	–	–	*invalid*	–	*invalid*

Table 1: State transition table. For readability, only the exclusive states are listed.

4 Experimental Parameters and the Simulation Environment

There are quite a lot parameters to be tuned before evaluating DHO. To facilitate the study, in all experiences we assume $T_{(\text{idle})}$ to be zero, or in other words, the client launches a new request as soon as the previous lock is released. An M/M/1 queuing model formalizes DHO's mechanism, namely a single-server queue model corresponding to a queued (FIFO) access to regulate the access to a resource, see [9]. In our context, two kind of requests are inserted in the queue. We take N_c clients as sources for arrivals of requests. Let $T_{(\text{Wait})}$ denote the waiting time of a request in queue, *i.e* the time between the call to request and the return from fetching into state *locked*:

$$T_{(\text{Wait})} = T_{(\text{WaitGrant})} \mid T_{(\text{WaitBlocked})} + T_{(\text{grant})} \mid T_{(\text{blocked})} + T_{(\text{fetch})}. \quad (3)$$

Let $T_{(\text{Blocking})}$ denote the time the client is blocking before acquiring the resource, *i.e.* the time between the call to acquire and the return from the fetching into state *locked*:

$$T_{(\text{Blocking})} = T_{(\text{blocked})} + T_{(\text{fetch})}. \quad (4)$$

Experiments to measure these waiting times were run under the *Grid Reality And Simulation environment*, GRAS, [10]. GRAS is socket based and is a powerful API that gives the opportunity to implement distributed applications on top of real platforms. Indeed, with GRAS, we can deploy experiments on real platforms without even modifying or recompiling the code. We

just have to relink the program. Under simulation mode, we exploited a description of a realistic platform, namely *gdx* which was a subset of Grid'5000. The simulation uses two nodes of that virtual cluster for its own purpose, for the server process and another to compute the different delays. The remaining 118 nodes had a role as clients ($N_c = 118$). Here is the description format:

```
<cluster id="gdx" prefix="gdx-" suffix=".orsay.grid5000.fr"
radical="1-120" power="3.185E9" bw="1.25E8" lat="1.0E-4"
bw_bb="1.25E9" laT_bb="1.0E-4"/>
```

As usual, latencies accumulate by their sum, whereas the bandwidth of a chain of links is their minimum. We use the improved SimGRID model [11] for communication to guarantee a good accuracy for messages greater than 100 KiB:

$$T = \alpha \cdot L + \frac{S}{\min(\beta \cdot bw, \gamma')}, \tag{5}$$

where $\alpha = 10.4$ and $\beta = 0.92$ are two correction factors. γ' sets the window size (γ_{TCP}) in TCP connections, $\gamma' = \gamma_{\text{TCP}}/2L$. L is the accumulated latency ($L = 2\ell + \ell_{bb}$). So, each flow really starts after αL. The TCP flows achieve 92% of the physical bandwidth. We use this improved model, since we are testing different sizes. We set the $\gamma_{\text{TCP}} = 10^7$.

5 Simulation Results and Analysis

Dho was run successfully as well in reality as in simulation. However, this section relates to a series of benchmark we have conducted in the simulation framework and is summarized as follows:

We evaluate fetching and updating phases and the DHO cycle time. We focus at first, upon write locks only by making variation of the lock time and other delays (see Sec. 4). Thereafter, we look for the impact of the other application delay $T_{(\text{WailBlocked})}$. We plan then, scenarios of read accesses and of both. An analysis is done finally through the queuing system.

As described in Sec. 2, the creation is the first phase that allows the handle to negotiate a *valid* state. Conversely, the destruction phase allows the handle to return to the *invalid* state. We study the influence of the latency on the times $T_{(\text{create})}$ and $T_{(\text{destroy})}$. We took three values both for ℓ and ℓ_{bb}: 10^{-5}, $5. \cdot 10^{-5}$ and 10^{-4}(s). $T_{(\text{create})}$ is at least $145.6 \cdot 10^{-5}$ and at most $145.6 \cdot 10^{-4}$. We also made similar variations to measure the destruction phase. The delays are between $72.8 \cdot 10^{-5}$ and $72.8 \cdot 10^{-4}$. We observed the

following relationship:

$$T_{(\text{create})} = \alpha \cdot (2L + \ell + 2\ell_{bb}) \quad (6)$$
$$T_{(\text{destroy})} = \alpha \cdot (L + \ell). \quad (7)$$

α is the parameter as above in the network model. These results are consistent since destruction involves less communication between client and server.

Next, we study the time during which the mapping/locking $T_{(\text{fetch})}$ and the updating $T_{(\text{push})}$ is done. We varied latency and bandwidth as previously as well as the resource size from 100 KiB to 1 GiB. We took the lock time $T_{(\text{locked})}$ in the range $0 \ldots 10$. Experiments were performed with clients that run 100 cycles.

Despite all variations, we observed that the push delay is exactly the communication time T as explained earlier in Sec. 5. Also, the fetch time is close to the push time:

$$T_{(\text{fetch})} = T_{(\text{push})} + \alpha \cdot L. \quad (8)$$

These results clearly demonstrate the quality of the model since all the bandwidth is consumed.

We aim at evaluating the DHO cycle time and to measure its adequacy in extreme circumstances. A first series of experiments were done with ew requests only. To better analyze the behavior of the API, we first had $T_{(\text{WaitBlocked})} = 0$ such that the handle switches to the blocking state as soon as the request is issued. The clients launched 100 write accesses and 100 executions were performed. In Table 5 we observe first that the waiting time is roughly equal to the blocking time. This is consistent with the formulas in Sec. 4. Secondly, we observe that the gap between durations for small resources (100 KiB and 10 MiB) is minimal. This is explained by the fact that, the mapping ($T_{(\text{fetch})}$) and update ($T_{(\text{push})}$) times are negligible (formulas 3 and

Resource size	$\overline{T_{(\text{locked})}}$	$\overline{T_{(\text{Wait})}}$	$\overline{T_{(\text{Blocking})}}$	$\overline{T_{(\text{DHO})}}$
100 KiB	0-10	$5.833 \cdot 10^2$	$5.833 \cdot 10^2$	$5.883 \cdot 10^2$
	10-20	$17.449 \cdot 10^2$	$17.4491 \cdot 10^2$	$17.599 \cdot 10^2$
	20-30	$29.164 \cdot 10^2$	$17.4491 \cdot 10^2$	$29.414 \cdot 10^2$
10 MiB	0-10	$6.024 \cdot 10^2$	$6.024 \cdot 10^2$	$6.075 \cdot 10^2$
	10-20	$17.67 \cdot 10^2$	$17.67 \cdot 10^2$	$17.821 \cdot 10^2$
	20-30	$29.336 \cdot 10^2$	$29.336 \cdot 10^2$	$29.587 \cdot 10^2$
100 MiB	0-10	$7.942 \cdot 10^2$	$7.942 \cdot 10^2$	$8.002 \cdot 10^2$
	10-20	$19.583 \cdot 10^2$	$19.583 \cdot 10^2$	$19.742 \cdot 10^2$
	20-30	$31.282 \cdot 10^2$	$31.282 \cdot 10^2$	$31.541 \cdot 10^2$
1 GiB	0-10	$27.569 \cdot 10^2$	$27.569 \cdot 10^2$	$27.712 \cdot 10^2$
	10-20	$39.20 \cdot 10^2$	$39.444 \cdot 10^2$	$39.444 \cdot 10^2$
	20-30	$50.86 \cdot 10^2$	$50.86 \cdot 10^2$	$51.204 \cdot 10^2$

Table 2: Measured times in seconds for BW $= 1.25 \cdot 10^8$ (B/s), L $= 10^{-5}$ s.

4). Also, durations are longer when the resource size and the lock time are larger. The difference of delays is not significant for the big sizes. Whereas the cycle delay is $31.541 \cdot 10^2$ for the resource of 100 MiB for example, it is $51.204 \cdot 10^2$ for a size 10 times larger. The lock time dominates the DHO cycle in these cases. The queue length($\overline{q}$) is about 116 in all cases. Thus, the DHO cycle time depends on the lock times $T_{(\text{locked})}$, $T_{(\text{fetch})}$, $T_{(\text{push})}$ and the queue length:

$$\overline{T_{(\text{DHO})}} = [T_{(\text{locked})} + T_{(\text{fetch})} + T_{(\text{push})} + \alpha \cdot \delta(3\ell + \ell_{bb})](\overline{q} + 1) \tag{9}$$

where $\delta \simeq 7$. The service time noted $1/\mu_1$ can be deduced from 9:

$$\frac{1}{\mu_1} = T_{(\text{locked})} + T_{(\text{fetch})} + T_{(\text{push})} \tag{10}$$

From formula 9, we can infer the mean time of the cycle for solely cr accesses:

$$\overline{T_{(\text{DHO})}} = [T_{(\text{fetch})} + \alpha \cdot \delta(3\ell + \ell_{bb})](\overline{q} + 1) + T_{(\text{locked})}. \tag{11}$$

Are counted neither lock time of previous requests nor update time $T_{(\text{push})}$ in such a cr mode. Similar executions on the basis of read accesses only, were conducted and confirmed. The service time $(1/\mu_2)$ is approximately $T_{(\text{fetch})}$. Results obtained so far have demonstrated the scalability of the DHO model, regardless of simulation time, resource size and the large number of requests to handle.

Let us recall that up to now, we assumed that $T_{(\text{WaitBlocked})} = 0$ in all our experiences. In asynchronous computations, $T_{(\text{WaitBlocked})}$ is the delay the client takes before launching the DHO function. We now study the influence of this parameter with set of experiences that fix a $T_{(\text{locked})}$ delay and a resource size of 100 MiB. As shown in Fig. 3, the time

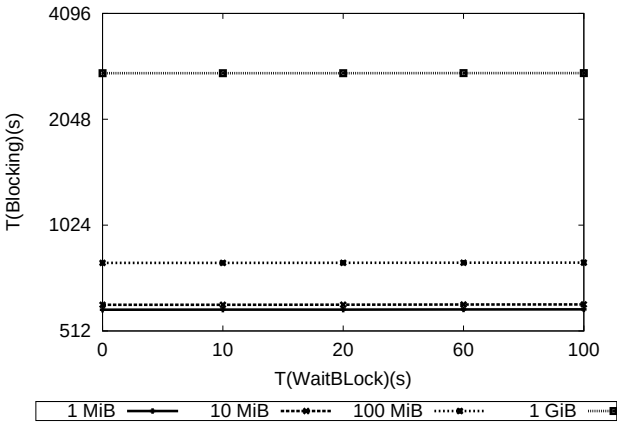

Figure 3: DHO cycle vs $T_{(\text{WaitBlocked})}$ ($T_{(\text{locked})}$ =5, size= 100 MiB)

remains almost the same, despite the different delay parameters we chose. In fact, the gap between $T_{(\text{Wait})}$ wait and $T_{(\text{Blocking})}$ extends. Thereby, if $T_{(\text{WaitBlocked})}$ increases, the blocking time decreases and vice versa. This is interesting because no additional overhead is measured in the cycle. Actually, applications that use the API expect a delay before switching to blocking state.

The last scenario relates to a random sequence of both types of requests. We took $T_{(\text{locked})}$ and $T_{(\text{Blocking})}$ within the range $0 \dots 10$ s. For a size of 100 MiB, the measurements reported as the cycle delay, the waiting and the blocking times are respectively 446.35, 441,024 and 412.92. The queue length is approximately 114. Since our system is modeled as an M/M/1 queuing system, we analyze the occupancy of the model(ρ). It is defined as the average arrival rate(λ) divided by the average service rate(μ) such that, $\mu = \frac{\mu_1+\mu_2}{2}$. With *Little's* law [12] and formulas 11 and 9 is:

$$\rho = \frac{(\overline{q}+1)\mu}{T_{(\text{DHO})}}. \tag{12}$$

ρ reached $\simeq 0,4$ which means that the system has reached a stable state.

6 Conclusion

This paper presents a study of the DHO model according to different aspects. The life cycle of DHO has been detailed by a elaborated modeling. From measurements of different scenarios in which the two kind of accesses were issued, we conclude that DHO is a stable system. What is interesting, is that our model allows to issue asynchronous locks and then provides overlap between computation and control. Also, the queuing formalism has allowed us to confirm good performance of the current implementation.

In spite of successful results, more studies are needed in the future. Soon, we plan to compare simulated results with those obtained within Grid'5000. In ongoing work we investigate the locking of ranges of the resource by different handles. Also, we are looking for a formal modeling and a validation of a distributed model so as to improve DHO mechanism in terms of the DHO cycle time.

References

[1] J. Gustedt, "Data handover: Reconciling message passing and shared memory," in *Foundations of Global Computing*, ser. Dagstuhl Seminar Proceedings, J. L. Fiadeiro, U. Montanari, and M. Wirsing, Eds., no. 05081, Dagstuhl, Germany, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/297

[2] "The OpenMP API specification for parallel programming," www.openmp.org.

[3] P. Arquet, "Introduction à MPI – Message Passing Interface," 2001, www2.lifl.fr/west/courses/cshp/mpi.pdf.

[4] P.-N. Clauss and J. Gustedt, "Iterative computations with ordered read-write locks," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010.

[5] ——, "Experimenting iterative computations with ordered read-write locks," in *18th Euromicro International Conference on Parallel, Distributed and network-based Processing*, M. Danelutto, T. Gross, and J. Bourgeois, Eds. Italy Pisa: IEEE, 2010, pp. 155–162.

[6] J. Gustedt, S. Vialle, and A. De Vivo, "The parXXL environment: Scalable fine grained development for large coarse grained platforms," in *PARA 06*, ser. Lecture Notes in Computer Science, B. Kågström *et al.*, Eds., vol. 4699. Umeå, Sweden: Springer, 2007, pp. 1094–1104. [Online]. Available: http://hal-supelec.archives-ouvertes.fr/hal-00280094/en/

[7] M. Quinson and F. Vernier, "Byte-range asynchronous locking in distributed settings," in *17th Euromicro International Conference on Parallel, Distributed and network-based Processing - PDP 2009*, Weimar, Germany, 2009. [Online]. Available: http://hal.inria.fr/inria-00338189/en/

[8] M. Sipser, *Introduction to the Theory of Computation*. Boston: PWS, 1997.

[9] W. Smith, V. Taylor, and I. Foster, "Using run-time predictions to estimate queue wait times and improve scheduler performance," in *Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1999, pp. 202–219.

[10] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a generic framework for large-scale distributed experiments," in *10th IEEE International Conference on Computer Modeling and Simulation - EUROSIM / UKSIM 2008*. Cambrige, UK: IEEE, 2008. [Online]. Available: http://hal.inria.fr/inria-00260697/en/

[11] P. Velho and A. Legrand, "Accuracy study and improvement of network simulation in the SimGrid framework," in *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. Brussels, Belgium: ICST, 2009, pp. 1–10.

[12] J. D. Little, "A proof for the queuing formula:l=λw," in *Operations Research*. INFORMS, 1961, pp. 383–387.

[13] M. Quinson, "GRAS: a research and development framework for grid and P2P infrastructures," in *The 18th IASTED International Conference on Parallel and Distributed Computing and Systems*. IASTED, 2006. [Online]. Available: http://hal.inria.fr/inria-00108389

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399